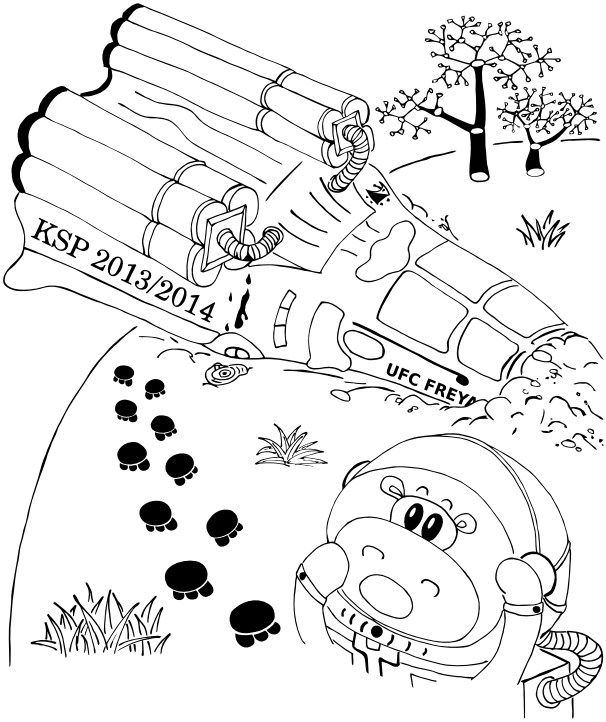
JIŘÍ SETNIČKA A KOLEKTIV

Korespondenční seminář z programování

XXVI. ročník – 2013/2014



matfyzpress
VYDAVATELSTVÍ
MATEMATICKO-FYZIKÁLNÍ FAKULTY
UNIVERZITY KARLOVY V PRAZE

JIŘÍ SETNIČKA A KOLEKTIV

Korespondenční seminář z programování

XXVI. ročník – 2013/2014

matfyzpress

Praha 2014

Vydáno pro vnitřní potřebu fakulty.
Publikace není určena k prodeji.

ISBN 978-80-7378-273-3

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož dvacátý šestý ročník se vám dostává do rukou, patří k nejznámějším aktivitám pořádaným MFF pro zájemce o informatiku a programování z řad studentů středních škol. Řešením úloh našeho semináře získávají středoškoláci praxi ve zdolávání nejrůznějších algoritmických problémů, jakož i hlubší náhled na mnohé disciplíny informatiky.

Ročník *KSP* je obvykle rozdělen do pěti *sérií*, neboli kol. Během každé rozešleme řešitelům zadání většinou osmi úloh okořeněné příběhem. Poslední úloha je tvořena tzv. *seriálem*, což je povídání o nějakém zajímavém informatickém tématu prolínající celý ročník. V této ročence je seriál vyčleněn z ostatních úloh a uveden pohromadě.

Na sepsání řešení v klidu domácího krbu a odevzdání přes naše stránky nebo poštou bývá několik týdnů. Poté vše opravíme, výsledkovou listinu se vzorovými řešeními vystavíme na Internet a pošleme poštou s další sérií.

Závěrečným bonbónkem je pak týdenní *soustředění* nejlepších řešitelů semináře, konané obvykle na začátku následujícího ročníku. Účastníci soustředění zažijí bohatý program – aktivity ryze odborné (především přednášky na různá zajímavá témata), ryze neodborné (kupříkladu hry a soutěže v přírodě) i různé kombinace obojího. Pro začínající řešitele tradičně pořádáme o trochu kratší jarní soustředění, kam může jet kterýkoliv středoškolák se zájmem o programování či informatiku, i když třeba ještě nic nevyřešil.

I letos jsme pro řešitele připravili několik novinek: podařilo se vedle hlavní kategorie vytvořit i kategorii začátečnickou, kde si na své přijdou začínající řešitelé. Pro začátečníky také pokračujeme v tradici zveřejňovat lehčí, mnohdy návodné, varianty úloh v hlavní sérii. Dále byla spuštěna programátorská encyklopedie,[1] kde lze vedle programátorských kuchařek nalézt i kratší články na různá informatická témata.

Mimo jiné jsme opět zorganizovali on-line soutěž Kasiopea.[2] Také jsme zvládli v průběhu roku přednášet na středních školách při Putovních přednáškách.[3]

[1] `http://ksp.mff.cuni.cz/encyklopedie/`

[2] `http://ksp.mff.cuni.cz/akce/kasiopea/`

[3] `http://ksp.mff.cuni.cz/akce/putovni-prednasky/2013/`

(Nejen) u úloh v této knize lze zahlédnout tyto značky označující typ úlohy:


Takto označenou úlohu považujeme za řešitelnou i pro začátečníky, zkušení řešitelé ji jistě zvládnou levou zadní. Pro její vyřešení by neměly být potřeba žádné speciální znalosti.


Aby si i pokročilí přišli na své, zařazujeme někdy do zadání těžkou úlohu, která se může stát leckomu noční můrou. Na její pokoření jsou často potřeba hlubší znalosti algoritmů a datových struktur, odměnou je však vyšší bodový zisk.


Této úloze říkáme *praktická*, jelikož není potřeba popsat algoritmus, jen ho naprogramovat a odevzdat přes Internet. Bližší informace naleznete přímo v jejím zadání.


V každém ročníku KSP rozebíráme na pokračování nějaké zajímavé informatické téma do hloubky. Poslední úloha série je pokračováním takového *seriálu* – obsahuje kromě samotného zadání ještě text, ve kterém se můžete dozvědět o tématu něco nového. Jelikož díly seriálu na sebe navazují, vyplatí se mít nastudované i předchozí série.


Protože chápeme, že k „uvaření“ řešení jsou často potřeba znalosti základních algoritmů a datových struktur, obvykle též přikládáme do každé série tzv. *kuchařku,* ze které se můžete takové věci naučit. Často je také v zadání úloha, již lze řešit algoritmem z kuchařky. A pozor – další kuchařky najdete na našich webových stránkách.


Dále tímto symbolem označujeme místa, jejichž pochopení může vyžadovat větší zamyšlení, případně nějaké předchozí znalosti.

Chcete-li se na cokoliv zeptat, ať už ohledně semináře, studia na naší fakultě nebo nějakého informatického či programátorského problému, neváhejte a napište nám na diskusní fórum na stránce *http://ksp.mff.cuni.cz/forum/* nebo na naši poštovní adresu:

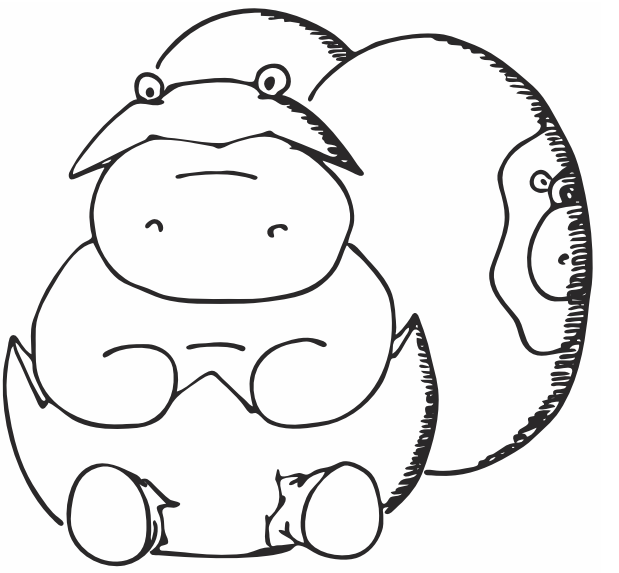
Korespondenční seminář z programování
KAM MFF UK
Malostranské náměstí 25
118 00 Praha 1

e-mail: *ksp@mff.cuni.cz*
www: *http://ksp.mff.cuni.cz/*

Obsah

KSP-Z

Začátečnická kategorie KSP



Zadání úloh KSP-Z

První série

Úlohy označené piktogramem počítače byly zadány jako praktické. To znamená, že na webu byly umístěny sady vstupů a soutěžním úkolem bylo odevzdat správné výstupy. Generování výstupů bylo čistě na volbě účastníků.

26-Z1-1 Kevin a magnety — 8 bodů

Kevin má v krabičce N magnetů – obyčejných magnetických tyček, které mají na jednom konci pól `+` a na druhém `-`. Jednoho dne se rozhodl, že si v nich udělá pořádek.

Jeden po druhém magnety vyndaval z krabičky a pokládal je do řady na stůl. Někdy se spojily k sobě (pokud se setkaly opačnými póly), jindy se mezi nimi zase vytvořila mezera.

Kevina by zajímalo, do kolika částí se mu magnety spojily.

Tvar vstupu: Program na vstupu dostane na prvním řádku číslo N ($1 \leq N \leq 1\,000\,000$). Každý z dalších N řádků bude obsahovat buď řetězec `+-`, nebo `-+` udávající, kterým směrem Kevin magnet položil.

Tvar výstupu: Na výstup vypište jediný řádek obsahující jedno celé číslo: počet oddělených částí, do kterých se magnety spojí.

Ukázkový vstup:

```
4
+-
+-
-+
+-
```

Ukázkový výstup:

```
3
```

26-Z1-2 Piškvorky — 10 bodů

Kevin sedí na hodině dějepisu. Bitvy, letopočty, děsná nuda. Raději si hraje se svým novým smartphonem a píše si s kamarádkou Sárou, která se ve vedlejší třídě nudí na hodině matiky. Aby si zkrátili dlouhou chvíli, rozhodli se hrát piškvorky.

Jejich aplikace na piškvorky ale není dokonalá a neumí poznat, jestli už někdo vyhrál. To si Kevin se Sárou uvědomili až po nějaké době, a tak by je nyní zajímalo, kolikrát kdo z nich vyhrál. Kevin má křížky a Sára kolečka.

Tvar vstupu: Na vstupu dostanete čísla S a R udávající šířku a výšku hrací plochy ($1 \leq R, S \leq 1\,000$). Na dalších R řádcích bude vždy S znaků `X`, `O` nebo `.`, kde `X` znamená křížek, `O` kolečko a `.` znamená prázdné políčko hrací plochy.

Tvar výstupu: Na výstup vypište jediný řádek se dvěma celými čísly oddělenými mezerou: počtem pětic křížků a počtem pětic koleček. Pětice mohou ležet v libovolném řádku, sloupci, nebo úhlopříčce.

Ukázkový vstup:

```
6 6
XXXXXO
.X..OX
..XO.X
..OX.X
.O..XX
OOOOOX
```

Ukázkový výstup:

```
4 3
```

26-Z1-3 Zamilovaný dopis — 10 bodů

Kevin působí jako bubeník ve studentské rockové kapele *Velká tlama.* Kapela během svého prvního veřejného koncertu sklidila ohromný úspěch. Další den Kevin objevil ve schránce K dopisů. Mezi všemi nudnými úředními dopisy jeden vyčníval. Byl navoněný a na obálce měl nakreslené růžové srdíčko. Ten si okamžitě přečetl. Byl od fanynky!

Takové dopisy Kevin nedostává každý den. Obsah dopisu si tak hned slovo od slova zapamatoval. Bohužel pak při cestě do školy pršelo a dopisy Kevinovi zmokly. Některé se staly téměř nečitelnými. I tak se ale snažil a přečetl z nich, co jen šlo. Posloupnosti těchto znaků si zapsal. Nyní by chtěl vědět, který z těchto dopisů by mohl být oním dopisem se srdíčkem.

Tvar vstupu: Na prvním řádku vstupního souboru bude zadán text původního dopisu, jehož délka bude nejvýše 300 000 znaků. Na druhém řádku bude zadáno číslo K udávající celkový počet doručených dopisů ($1 \leq K \leq 50$). Na každém ze zbylých K řádků bude posloupnost maximálně 300 000 znaků, které se Kevinovi z daného dopisu podařilo přečíst. Všechny znaky dopisů budou složeny z malých a velkých písmen anglické abecedy a znaků `.`, `,`, `_`, kde `_` odděluje jednotlivá slova.

Tvar výstupu: Vypište K řádků výstupu. Na i-tý řádek umístěte řetězec `ANO`, pokud i-tý dopis může být oním dopisem se srdíčkem. To znamená, že i-tý dopis lze z původního dopisu vytvořit vynecháním některých znaků. V opačném případě vypište řetězec `NE`.

Ukázkový vstup:

```
Horoucne_Te_miluji,_Kevine.
5
Horoucne_Te_milujeme,_Kevine.
nemiluji
MILUJI.
Hor___eve.
Horce_Te_miluji,Kene.
```

Ukázkový výstup:

```
NE
ANO
NE
ANO
ANO
```

26-Z1-4 Hroch v jezeře — 12 bodů

Kevin po odpoledních hraje počítačovou hru Hroch v jezeře. Ve hře plave s hrochem ve velkém jezeře, ze kterého se tu a tam vynořuje ostrůvek. Některé ostrůvky jsou pusté, na jiných se nachází hroší laskominy – jídlo všeho druhu. Cílem hry je doplavat s hrochem na ostrůvek s co největším množstvím jídla a mezitím nevylézat z vody.

Ostrov je souvislá oblast políček pevniny, kde políčka považujeme za spojená, pokud se dotýkají hranou (nikoliv pouze rohem). Hroch může plavat nahoru, dolů, doleva a doprava.

Tvar vstupu: Na prvním řádku vstupu dostanete čísla S a R udávající šířku a výšku mapy hry. Na každém z dalších R řádků bude S znaků, přičemž:

- `.` znamená, že na políčku je voda,
- `#` znamená, že na políčku je pevnina bez jídla,
- `J` je pevnina s jídlem,
- `H` je políčko, kde hroch začíná.

Zaručujeme, že na vstupu bude právě jedno políčko se znakem `H`.

Tvar výstupu: Na výstup napište jediné celé číslo udávající největší množství jídla na ostrově, ke kterému se hroch může ze své startovní pozice doplavit povolenými způsoby bez toho, aby mezitím vystoupil z vody.

Ukázkový vstup:

```
10 8
.J##..#.J#
..###.J#.J
#JJJ#..#.J
..###..J.J
.......#.J
.J.H.###.J
.J...#JJ.#
.J#..#JJ.J
```

Ukázkový výstup:

```
6
```

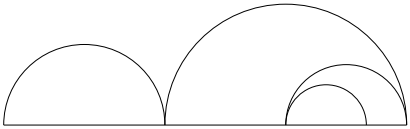
Zadání úloh KSP-Z – 1. série

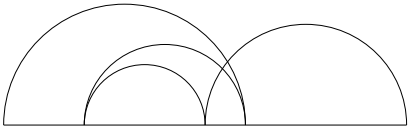
Zbývající úlohy (bez piktogramu počítače) byly zadány jako teoretické. U nich bylo nutné vymyslet algoritmus a slovně jej popsat.

26-Z1-5 Úkol z geometrie — 12 bodů

Kevin a jeho spolužáci dostali na hodině geometrie následující úkol. Mají zadaných N bodů na ose x. Je třeba každé dva po sobě jdoucí body spojit horní půlkružnicí. Kevin se podíval na posloupnost ze zadání a začal přemýšlet, jestli se některé dvě půlkružnice protnou. Navrhněte co nejefektivnější algoritmus, který pro zadanou posloupnost N bodů odpoví `ANO`, pokud se některé půlkružnice protínají, a `NE`, pokud nikoliv.

Na prvním obrázku je příklad pro posloupnost $0, 8, 20, 14, 18$ a na druhém pro posloupnost $0, 12, 4, 10, 20$. V prvním se kružnice neprotínají, v tom druhém protínají.





26-Z1-6 Nezbední skřítci — 14 bodů

Kevin byl na vánočních trzích a co nesehnal – opravdové, živé vánoční skřítky! Tomu nemohl odolat a hned si jich N koupil. Dokonce byl každý jinak velký.

Aby mu neutekli, vyrobil si pro ně doma řadu N klícek a každého do jedné zavřel. Zavření skřítci ale dělali příšerný kravál a neustále si stěžovali, že vedle sebe nemají podobně velké kamarády, se kterými by si mohli povídat.

Tak se Kevin rozhodl, že je v klíckách seřadí podle velikosti. Pro pořádek vždy vyndá jen dva skřítky a vymění jejich pozice.

Navrhněte algoritmus, který má Kevin použít, aby skřítky seřadil podle velikosti od nejmenšího po největšího na co nejméně prohození.

Příklad: Pokud máme tři skřítky zavřené v klíckách v pořadí: $3, 1, 2$ (čím větší číslo, tím větší skřítek), tak je můžeme seřadit na dvě prohození. Nejdříve prohodíme skřítka 3 se skřítkem 1 a poté skřítky 3 a 2.

Druhá série

26-Z2-1 Had z domina — 8 bodů

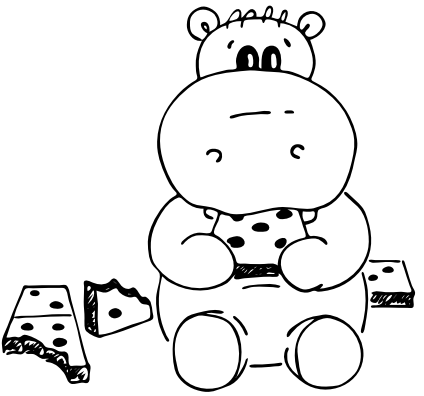
Kevinova malá sestra Zuzka dostala k Vánocům domino. Zatím ho neumí hrát, tak kostky jenom skládá do různých tvarů. Úplně nejraději z nich staví tlustého hada – vezme N kostek, postaví je do jedné dlouhé řady a přikládá je k sobě delší stranou.

Se svým výtvorem se přišla pochlubit Kevinovi. Toho ale víc než had zajímala samotná čísla na něm. V hadovi by chtěl otočit právě jednu kostku o 180° tak, aby první i druhý řádek hada měly sudý součet. Kolika způsoby toho může dosáhnout?

Tvar vstupu: Na prvním řádku dostanete číslo N udávající délku hada, kterého Zuzka vytvořila ($1 \leq N \leq 1\,000$). Na dalších dvou řádcích bude na každém N čísel z rozmezí 0 až 9 oddělených mezerou.

Tvar výstupu: Na výstup vypište jedno celé číslo K udávající, kolik je možností, jak otočit právě jednu kostku tak, aby byl součet v obou řádcích sudý.

Ukázkový vstup:	*Ukázkový výstup:*
6 1 2 4 9 2 3 3 7 3 2 1 3	4
5 2 3 1 3 4 2 1 2 7 8	0



26-Z2-2 SADO — 10 bodů

Společně s Kevinem, Sárou a Petrem jdete na týmovou matematickou soutěž jménem SADO. Hned na začátku jste dostali následující úlohu a ostatní po vás chtějí její řešení.

Kolik celých čísel z uzavřeného intervalu $[a, b]$ je dělitelných číslem x a zároveň číslem y?

Tvar vstupu: Na vstupu dostanete na jednom řádku čtyři celá čísla a, b, x, y oddělená mezerou ($1 \leq a \leq b \leq 10^{15}$, $1 \leq x, y \leq 10^9$).

Tvar výstupu: Na výstup vypište jedno celé číslo, které bude odpovědí na výše položenou otázku.

Ukázkový vstup:	*Ukázkový výstup:*
`1 90 3 5`	`6`

26-Z2-3 Šifrovaná zpráva — 10 bodů

Nezbednice Sára už zase lelkuje ve škole. Místo toho, aby dávala pozor, píše šifrovanou zprávu pro Kevina. Nejdříve z písmen anglické abecedy a bez mezer napíše původní zprávu, pak každé písmeno nahradí jiným. Stejná písmena jsou vždy nahrazena stejně a různá písmena jsou naopak vždy nahrazena různými písmeny. Takové šifře se obvykle říká jednoduchá substituce.

Po vás by chtěla zkontrolovat, zda někde neudělala chybu. Dostanete text původní zprávy a jeho zašifrovanou podobu a máte ověřit, že každé dva stejné znaky se zašifrují na stejný znak a každé dva různé na různý znak.

Tvar vstupu: Na prvním řádku vstupu bude číslo N udávající délku zprávy ($1 \leq N \leq 10^6$). Na druhém řádku dostanete původní zprávu složenou z N velkých písmen anglické abecedy. Na třetím, posledním, řádku dostanete zašifrovanou zprávu složenou také z N velkých písmen anglické abecedy.

Tvar výstupu: Pokud zpráva není zašifrovaná správně, vypište na jediný řádek výstupu slovo `NE`. Pokud je zašifrovaná správně, na první řádek vypište slovo `ANO` a na druhý vypište bez mezer 26 znaků, které budou udávat, na co se zašifrují která písmena ze vstupu.

První bude udávat, na co se zašifruje písmeno `A`, druhý, na co se zašifruje písmeno `B` a tak dále, až poslední bude udávat, na co se zašifruje písmeno `Z`. U písmen, která se v původní zprávě nevyskytují, si můžete vybrat libovolně. Stále však musíte dodržet, že se různá písmena zašifrují různě.

Ukázkový vstup:	*Ukázkový výstup:*

`18`
`AHOJKEVINEJAKSEMAS`
`XOBNKYWZCYNXKAYFXA`
	`ANO`
	`XDEGYHIOZNKJFCBLPQARSWMTUV`

26-Z2-4 Životně důležitá úloha — 12 bodů

Kevinovi se po silvestrovské noci zdál fakt divný sen. Byl vsazen do přísně střeženého žaláře a odsouzen k vyhladovění. Krutý to osud! Naštěstí ale svítala naděje, v rohu byl malý nenápadný poklop vedoucí ke svobodě. K jeho otevření ale bylo potřeba zadat na číselném zámku odpověď na následující úlohu.

Máme zadanou posloupnost N celých čísel. Prvek má v posloupnosti aritmetický výskyt, pokud se v ní vyskytuje pravidelně. To znamená, že pokud si vezmeme indexy jeho výskytů v posloupnosti, tak tyto indexy tvoří aritmetickou posloupnost, tj. rozdíl každých dvou po sobě jdoucích indexů je stejný. Tomuto rozdílu se říká *diference*.

Vypište ve vzestupném pořadí všechny prvky, které mají v posloupnosti aritmetický výskyt, a navíc ke každému z nich i jeho diferenci. Pokud se prvek v posloupnoti vyskytuje pouze jednou, má aritmetický výskyt s diferencí 0.

Tvar vstupu: Na vstupu na prvním řádku dostanete délku posloupnosti N ($1 \leq N \leq 10^5$). Na dalších N řádcích budou jednotlivé prvky posloupnosti v rozmení -10^9 až 10^9.

Tvar výstupu: Na první řádek výstupu vypište počet prvků K, které mají v posloupnosti aritmetický výskyt. Na dalších K řádků vypište ve vzestupném pořadí jednotlivé prvky. Každý řádek bude obsahovat hodnotu prvku, mezeru a jeho diferenci.

Ukázkový vstup:

```
6
1
2
3
2
3
3
```

Ukázkový výstup:

```
2
1 0
2 2
```

26-Z2-5 Nedopité skleničky — 12 bodů

Kevin se celý vyděšený probudil, opláchl se vodou a šel do kuchyně. Tam na stole našel řadu N nedopitých skleniček šampaňského. Je to divné, ale skleničky byly seřazeny podle toho, kolik v nich ještě zbývalo, a v lahvi navíc bylo dalších K mililitrů šampaňského. To by chtěl Kevin nalít právě do jedné ze skleniček tak, aby se hladinou co nejvíce přiblížil jiné skleničce. Do které skleničky jej má nalít?

Jinými slovy: Vaším úkolem je v setříděné posloupnosti najít dvě čísla (nemusí být nutně těsně za sebou), jejichž rozdíl se co nejvíce blíží číslu K. Na-

vrhněte co nejefektivnější algoritmus, který tato čísla najde. Pokud existuje více takových dvojic, stačí nalézt libovolnou z nich.

26-Z2-6 Čtecí hlavy — 14 bodů

Kevin od tatínka dostal dlouhou magnetickou pásku s několika vyznačenými místy a k tomu čtecí hlavy. Každá čtecí hlava jezdí nad páskou a dokáže z ní číst informace. Hlava v každém kroku může buď stát na místě, popojet o jedno políčko doleva, anebo popojet o jedno políčko doprava. Pak přečte informaci z políčka, kde zrovna je, a celý postup se opakuje.

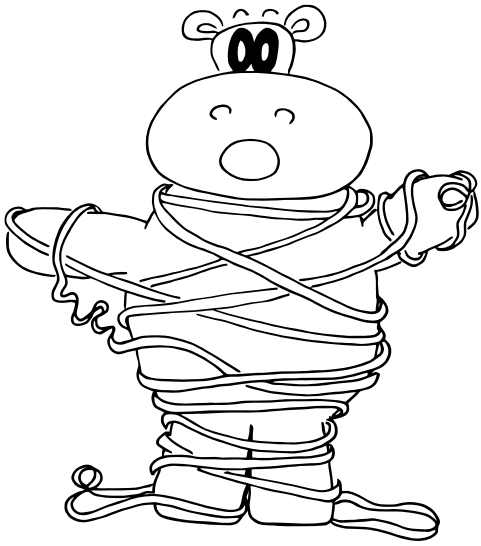
Zároveň je na pásce N míst, která chceme pomocí hlav přečíst. Máte zadané souřadnice míst k přečtení a počáteční souřadnice hlav. Na kolik nejméně kroků lze všechny zadané segmenty přečíst?

Úkol 1 [5b]: Čtete pomocí jedné hlavy.

Úkol 2 [9b]: Čtete pomocí dvou hlav.

Navrhněte co nejefektivnější algoritmus na spočítání minimálního potřebného počtu kroků. Efektivitu algoritmu počítejte vzhledem k hodnotě N, případně k velikosti pásky. Můžete předpokládat, že políčka, kde hlavy stojí na začátku, jsou již přečtené.

Poznámka: Tato úloha je velmi podobná úloze 26-2-7 z hlavní kategorie KSP, kde se však řešila pro obecný počet hlav. Pro jednu nebo pro dvě hlavy existuje jednodušší a efektivnější algoritmus na spočítání minimálního počtu kroků, než je ten popsaný v řešení odkazované úlohy.



Třetí série

26-Z3-1 Zámky labyrintu — 8 bodů

Kevin se společně se svými přáteli vydal na Matějskou pouť. Jako první atrakci si vyhlédli Labyrint snů. Mělo to ale jeden malý háček – hned při vstupu bylo potřeba otevřít několik číselných zámků.

Na každém zámku jsou nastavena tři celá čísla a, b, c a cílem je změnit právě jedno z těchto čísel o nějaké celé nezáporné r tak, aby čísla a, b, c v tomto pořadí tvořila aritmetickou posloupnost.[4] Najděte nejmenší takové r, které je třeba k jednomu z čísel přičíst nebo odečíst, abychom dostali aritmetickou posloupnost.

Tvar vstupu: Na vstupu na prvním řádku dostanete číslo N udávající počet zámků ($1 \leq N \leq 10\,000$). Na dalších N řádcích bude vždy trojice čísel a, b, c oddělených mezerou ($-1\,000\,000 \leq a, b, c \leq 1\,000\,000$).

Tvar výstupu: Na výstup vypište N řádků, kde i-tý z nich bude obsahovat právě jedno číslo r udávající, o kolik nejméně musíme jedno z čísel a, b, c z odpovídajícího řádku vstupu změnit.

Ukázkový vstup:

```
3
3 12 7
1 4 7
-5 25 51
```

Ukázkový výstup:

```
7
0
2
```

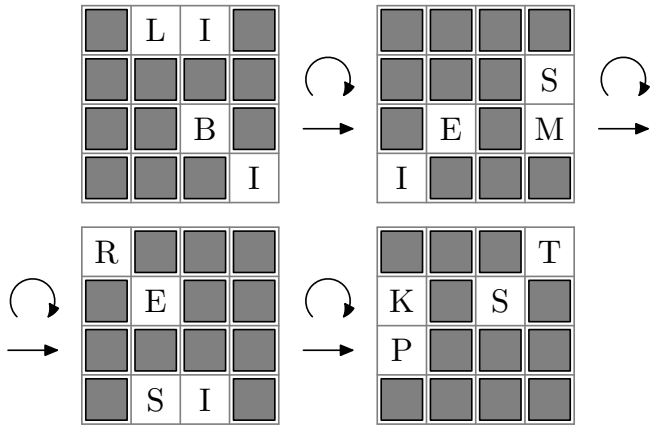
26-Z3-2 Čarodějova šifra — 10 bodů

Otevřením zámků to ale nekončilo. Hned za dveřmi stál člověk oblečený jako čaroděj a říkal, že do Labyrintu snů nepustí jen tak někoho. Pro vstup musíte vyřešit ještě další jeho úlohu.

Čaroděj Kevinovi a jeho kamarádům podal tajemnou čtvercovou destičku o rozměrech $K \times K$ políček (K je sudé), kde několik políček bylo proděravěných. Díry byly rozmístěny tak, že když budeme destičku postupně otáčet o 90° do

[4] Tj. aby platilo, že $b - a = c - b$.

celkem čtyř různých otočení, tak se na každé pozici objeví díra při právě jednom z nich.



Čaroděj po vás chce, abyste mu pomocí této destičky pomohli zašifrovat text. Šifrování provedete tak, že destičku nejprve přiložíte na tabulku $K \times K$ a v místech, kde jsou v destičce díry, zapíšete část textu (řádek po řádku, zleva doprava). Pak destičku otočíte o 90° po směru hodinových ručiček a budete postup ještě třikrát opakovat (destička se tak otočí do všech čtyř možných otočení). Pokud vám v průběhu šifrování dojde text, doplňte na jeho konec opakování písmen „KSP“.

Tvar vstupu: Na prvním řádku dostanete text k zašifrování o maximální délce 1 000 000 znaků obsahující pouze velká písmena anglické abecedy. Na druhém řádku bude číslo K udávající velikost destičky (platí $2 \leq K \leq 1000$). Slibujeme, že K bude vždy dost velké, aby se text do tabulky vešel.

Na dalších K řádcích bude vždy K znaků `#` nebo `.`, kde každá `.` udává díru a `#` plné políčko destičky. Destička splňuje všechny podmínky ze zadání, to nemusíte kontrolovat.

Tvar výstupu: Na výstup vypište výslednou tabulku pro zadaný text a destičku, tedy K řádků o K znacích.

Ukázkový vstup:

```
LIBISEMIRESITKSP
4
#..#
####
##.#
###.
```

Ukázkový výstup:

```
RLIT
KESS
PEBM
ISII
```

26-Z3-3 Hádanka — 10 bodů

Čaroděj se usmál: „Výborně! A teď pro vás mám už opravdu poslední hádanku.“ Při těchto slovech napsal na papír dlouhou řadu cifer a otazníků. Po dopsání si pohladil vousy a povídal: „Namísto otazníků v tomto čísle doplňte cifry tak, aby vzniklo co nejmenší číslo v desítkové soustavě, které je dělitelné devíti. Pozor! Toto číslo nesmí obsahovat žádné zbytečné nuly na začátku.“

Tvar vstupu: Na prvním řádku dostanete číslo N udávající délku řady ($1 \leq N \leq 1\,000\,000$). Na druhém řádku dostanete N znaků (cifer nebo otazníků). Na vstupu bude pokaždé alespoň jeden otazník.

Tvar výstupu: Na výstup vypište číslo s doplněnými otazníky, které splňuje čarodějovo zadání.

Ukázkový vstup:

```
6
6?34?7
```

Ukázkový výstup:

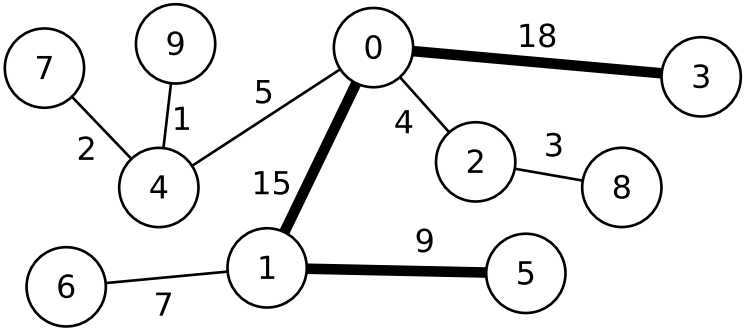
```
603477
```

26-Z3-4 Tvar labyrintu — 12 bodů

„Konečně uvnitř!“, vykřikl Kevin. Hned se s kamarády rozeběhli do všech stran a začali kreslit mapu labyrintu. Byl to opravdu zvláštní labyrint – obsahoval N křižovatek (kde konec slepé chodby také považujeme za křižovatku) a právě $N - 1$ chodeb. Nikde nebyly žádné cykly a prakticky se v něm nedalo zabloudit.

Informatici by takový labyrint mohli nazvat *grafem* a křižovatky a chodby poté *vrcholy* a *hranami* tohoto grafu. Více o grafové terminologii můžete zjistit v grafové kuchařce,[5] pro řešení této úlohy to však není nutné, stačí vám představa labyrintu z obrázku na následující straně.

[5] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`



„Bloudit se tu nedá, tak si dáme alespoň závod!“, navrhl Petr. „Najdeme tu nejdelší trasu a tu poběžíme!“ „No jo, ale která to je?“, zeptala se Sára. A to je úkol pro vás.

Tvar vstupu: Na prvním řádku dostanete číslo N udávající celkový počet křižovatek v labyrintu ($2 \leq N \leq 333\,333$). Křižovatky jsou očíslovány čísly $0, \ldots, N-1$, přičemž kamarádi začínají na křižovatce 0.

Na dalších $N-1$ řádcích jsou vždy dvě celá čísla k_i a d_i, kde k_i udává číslo křižovatky, ze které jsme došli do křižovatky i, a d_i udává délku chodby mezi nimi (platí $0 \leq k_i < i$ a $1 \leq d \leq 10\,000$).

Tvar výstupu: Na výstup vypište jedno celé číslo udávající vzdálenost dvou nejvzdálenějších křižovatek v labyrintu.

Ukázkový vstup:

```
10
0 15
0 4
0 18
0 5
1 9
1 7
4 2
2 3
4 1
```

Ukázkový výstup:

```
42
```

26-Z3-5 Ceny na střelnici — 12 bodů

Labyrintu už měli kamarádi dost, a tak se šli podívat i na jiné atrakce. Kevin zamířil na střelnici, že by třeba mohl vyhrát něco pěkného. Na střelnici stála v řadě spousta cen. Kevin se rozhodl, že se pokusí vyhrát nějaký souvislý úsek cen stojících vedle sebe. Zároveň ale chce takový úsek, kde se žádná cena nebude opakovat (co by také dělal se dvěma čepicemi). Pomozte mu takový najít.

Na vstupu budete mít posloupnost cen (každá cena je zapsána jako číslo, které určuje, co za typ ceny to je). Musíte najít nejdelší úsek, ve kterém se žádný typ nevyskytuje vícekrát.

Úkol 1 [4b]: Předpokládejte, že různých cen je řádově 50 a jsou očíslovány čísly 1–50.

Úkol 2 [8b]: Předpokládejte, že různých cen může být až tolik, jak je dlouhá řada, což může být klidně i několik milionů.

Navrhněte co nejefektivnější algoritmus. Pro obě varianty můžete (ale nemusíte) použít tentýž algoritmus.

26-Z3-6 Horská dráha — 14 bodů

A to nejlepší z celé Matějské nakonec, horská dráha! Kamarádi na ní jeli snad dvacetkrát. Na poslední jízdu si Kevin s sebou vzal výškoměr, který vyhrál na střelnici, a zapisoval si výšky na trase dráhy. Vždy, když se objevil na nějakém lokálním vrcholu nebo údolí, zapsal si výšku, a někdy si ji zapsal i mezi tím. Celkem si zapsal N čísel. Nyní by ho zajímalo:

Úkol 1 [4b]: Jakou výšku si do seznamu zapsal nejvíckrát?

Úkol 2 [10b]: V jaké výšce se na trase dráhy nejvíckrát vyskytl? To obvykle nebude jen jedna výška, ale celý interval. Vám bude stačit, když naleznete libovolnou výšku, pro kterou toto platí.

Pomozte mu a navrhněte co nejefektivnější algoritmus, který úlohu vyřeší. Obě varianty řešte zvlášť.

Čtvrtá série

26-Z4-1 Vražedná čísla — 8 bodů

Kevin utíkal lesem. Pronásledovalo ho N vražedných čísel. Už ho skoro měla. Rychle prokličkoval houfem stromů a prudce zatočil. Při tomto obratu zakopl o kořen, upadl a vyvrtl si kotník.

Byl ztracen, vražedná čísla se na něj hrnula ze všech stran. „Kolik je mezi námi dvojic, jejichž součet je násobkem naší královny Q?“ ptala se čísla neustále dokola. „Řekni kolik, kolik jich je?“

Když byla čísla až u něj a začala se po něm sápat, probudil se na hodině matematiky. Nahlas si oddechl, čímž upoutal pozornost paní učitelky, byl vyvolán k tabuli a dostal zadán onen příklad vražedných čísel ze snu. Měl říci, kolik je mezi N zadanými čísly dvojic se součtem rovným $k \cdot Q$ pro pevně zadané Q a libovolné celé k.

Tvar vstupu: Na vstupu na prvním řádku dostanete čísla N a Q oddělená jednou mezerou ($1 \leq N, Q \leq 1\,000\,000$). Na druhém řádku pak bude N vražedných čísel $x_1, \ldots, x_N$ ($-1\,000\,000 \leq x_i \leq 1\,000\,000$).

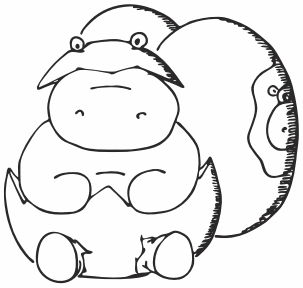
Tvar výstupu: Na výstup vypište jedno celé číslo udávající počet dvojic, jejichž součet je násobkem čísla Q. Ve dvojici nemůže být dvakrát to samé vražedné číslo, ale dvojice může obsahovat dvě čísla se stejnou hodnotou.

Ukázkový vstup:	*Ukázkový výstup:*
`5 6` `3 3 2 4 9`	`4`

Hledané dvojice jsou: $3 + 3$, $2 + 4$ a dvakrát $3 + 9$.

26-Z4-2 Sbírání vajíček — 10 bodů

Na velikonoční neděli byl u Zuzky a Kevina doma velikonoční zajíček a nechal jim na zahradě N velikonočních vajíček. Zahrada má tvar jedné dlouhé nudle a vajíčka na ní leží na souřadnicích $v_1, \ldots, v_N$. Zuzka a Kevin by chtěli všechna vajíčka sesbírat do košíku.



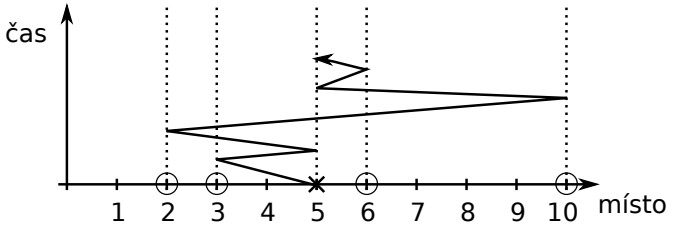
Kevin bude stát na místě a hlídat košík, zatímco Zuzka bude chodit pro vajíčka. Jelikož je malá, může vždy nést maximálně jedno vajíčko. Kam si má Kevin s košíkem stoupnout, aby se Zuzka co nejméně nachodila?

Tvar vstupu: Na vstupu na prvním řádku dostanete číslo N ($1 \leq N \leq 100\,000$). Na druhém řádku bude N celých čísel $x_1, \ldots, x_N$ udávajících souřadnice vajíček (platí pro ně $0 \leq x_1, \ldots, x_N \leq 10^9$).

Tvar výstupu: Na výstup vypište jedno desetinné číslo udávající optimální pozici, kam se má Kevin s košíkem postavit. Číslo zaokrouhlete na tři desetinná místa. Pokud existuje více řešení, vypište libovolné z nich.

Ukázkový vstup:	*Ukázkový výstup:*
`4` `3 2 6 10`	`5`

Příklad je znázorněn na následujícím schématu:



Kolečka představují vajíčka, křížek Kevina s košíkem. Klikatá čára znázorňuje jednu z mnoha možných cest Zuzky pro jednotlivá vajíčka. V tomto případě celkem naběhá $2 \cdot 2 + 2 \cdot 3 + 2 \cdot 5 + 2 \cdot 1 = 22$ metrů. Když si Kevina zkusíme posunout kamkoliv jinam, zjistíme, že výhodnější pozice než tato není.

26-Z4-3 Hra Othello — 10 bodů

Kevin hraje s Petrem hru Othello, kterou můžete znát také pod jménem Reversi. Hrají podle těchto zjednodušených pravidel:

1. Hraje se na desce $D \times D$.
2. Hráči se střídají pravidelně po jednom tahu.
3. Hráč ve svém tahu umístí kámen své barvy na libovolné prázdné pole (což je rozdíl oproti normálním pravidlům, kde je umisťování kamenů omezeno). Pak v každém z osmi směrů provede následující:

   Od svého kamene jde po kamenech soupeřovy barvy, dokud nenarazí na svůj kámen, prázdné pole, nebo na kraj desky. Pokud jako na první dojde na svůj kámen, tak všechny soupeřovy kameny, po kterých k němu došel, změní na své. Pokud narazí na prázdné pole a nebo na kraj desky, nic se v tomto směru nestane.

Kevin je zrovna v situaci, kdy jsou na desce právě tři volná políčka a je na tahu. Kolika nejvíce kamenů může na konci hry dosáhnout za předpokladu, že on i Petr hrají optimálně a snaží se maximalizovat počet svých kamenů na konci hry?

Tvar vstupu: Na prvním řádku bude číslo D udávající rozměr desky ($2 \leq D \leq 100$). Na dalších D řádcích bude vždy D znaků `K`, `P`, nebo `.`, kde `K` značí Kevinův kámen, `P` Petrův kámen a `.` (tečka) značí prázdné pole. Je zaručeno, že na vstupu se budou vyskytovat právě tři prázdná pole.

Tvar výstupu: Na výstup vypište, kolik kamenů bude mít Kevin na konci hry za předpokladu, že oba hráči hrají optimálně.

Ukázkový vstup:	*Ukázkový výstup:*

`4`
`PK.P`
`KKPK`
`.PKK`
`PKK.`
	`10`

První tah musí Kevin vést doprava dolů – i když tím nezíská žádný Petrův kámen, zabrání tak, aby v dalším půltahu přišel o mnohem víc. Zbylé dva půltahy už jsou symetrické a ve výsledku tak může hrací deska vypadat třeba takto:

```
PPPP
KKPK
KKKK
PKKK
```

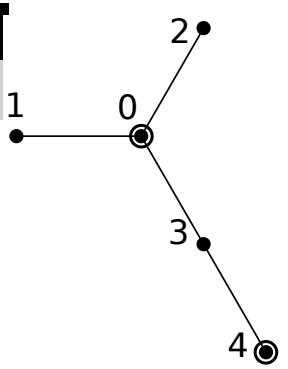
26-Z4-4 Hlídači v labyrintu — 12 bodů

Pamatujete si na Labyrint, ve kterém byli Kevin a jeho kamarádi v minulé sérii? Takové zvláštní bludiště tvořené křižovatkami a chodbami, kde nebyl vůbec žádný cyklus (informatik by takové speciální bludiště mohl nazvat *stromem*).

V novinách se psala děsivá zpráva, že zrovna noc po jejich návštěvě byl Labyrint vykraden. Aby se podobná událost již více neopakovala, rozhodli se majitelé do křižovatek postavit hlídače. Z křižovatky hlídač vidí právě do přilehlých chodeb. Hlídači budou na křižovatky rozestaveni tak, aby do každé chodby viděl alespoň jeden z nich. Kolik nejméně hlídačů musí majitelé najmout?

Tvar vstupu: Na prvním řádku dostanete číslo N udávající celkový počet křižovatek v Labyrintu ($2 \leq N \leq 333\,333$). Křižovatky jsou očíslovány čísly $0, \ldots, N-1$.

Umístíme 0. křižovatku a pomocí dalších $N-1$ řádků popíšeme uspořádání zbylých $N-1$ křižovatek. Na i-tém řádku bude jedno celé číslo k_i udávající číslo již existující křižovatky, ke které je i-tá křižovatka připojena ($0 \leq k_i < i$).



Tvar výstupu: Na výstup vypište jedno celé číslo udávající, kolik nejméně hlídačů musí v Labyrintu být, aby jej uhlídali.

Ukázkový vstup:	*Ukázkový výstup:*
5 0 0 0 3	2

Labyrint odpovídající tomuto vstupu vidíme na obrázku vlevo (zvýrazněné křižovatky označují jedno z možných optimálních rozmístění hlídačů).

26-Z4-5 Podávání rukou — 12 bodů

Velikonoce, to je důvod k oslavám. To se tak Kevinovi vždycky sejde celá N-členná rodina, shromáždí se u velkého kulatého stolu a každý s každým si podá ruku. Jelikož mají starou pověrčivou babičku, tak se žádné dva páry rukou nesmí křížit.

Podávání rukou funguje tak, že si v jednom taktu vždy některé dvojice podají ruku. Kolik nejméně taktů je potřeba, aby si každý podal ruku s každým? Svá řešení podrobně zdůvodněte.

26-Z4-6 Překreslení obrázku — 14 bodů

Když byl Kevin o Velikonocích vyšupat Sáru, dostal od ní takový zvláštní černobílý obrázek velký $S \times R$ políček. Obrázek ho celkem zaujal a rozhodl se, že si jej obkreslí. Bohužel má k dispozici pouze štětec velký $K \times K$ políček. Tím chce překreslit alespoň to, co půjde.

Štětec je možné přiložit na libovolné místo papíru a obarvit tak příslušných $K \times K$ políček na černo. Jedno políčko je možné obarvit vícekrát. Kevin ale nikdy nechce obarvit žádné políčko, které v původním obrázku bylo bílé. Kolik nejvíce černých políček může vybarvit, aniž by přitom začernil jakékoliv bílé?

Nalezněte co nejefektivnější řešení vzhledem k hodnotám S, R a K. Můžete předpokládat, že $K \leq \min(R, S)$. Vše pečlivě zdůvodněte.

⇧ **Lehčí varianta (za 6 bodů):** Pokud si nebudete vědět rady, můžete úlohu zkusit vyřešit jen v jednom rozměru, tedy pro obrázek velký $S \times 1$ a štětec velký $K \times 1$.

Vzorová řešení KSP-Z

26-Z1-1 Kevin a magnety

Magnety se přitahují tehdy, když mají naproti sobě opačné póly, a naopak se odpuzují, jsou-li umístěné stejnými póly k sobě. Částem, do kterých se magnety spojily, říkejme třeba *komponenty*.

Každé dva po sobě následující magnety v jedné komponentě se přitahují. Komponenta začíná buď prvním magnetem, nebo místem, kde se magnety odpuzují. Obdobně končí posledním magnetem, nebo místem, kde se magnety odpuzují, čili tam, kde začíná jiná komponenta.

Každá hranice mezi komponentami se skládá z dvojice po sobě následujících odpuzujících se magnetů. Dají se tedy jednoduše spočítat, ale z toho nedostaneme hned počet komponent. K jeho získání musíme k počtu hranic ještě přičíst jedničku: máme-li k sobě stlučených 10 prken, je mezi nimi 9 mezer.

Mohli bychom nejdříve do pole načíst orientace všech magnetů, a pak spočítat, kolik po sobě následujících dvojic se odpuzuje. Načítání a počítání odpuzujících se dvojic ale můžeme dokonce spojit do sebe: budeme si pamatovat pravou stranu posledního načteného magnetu, a když načteme další, podíváme se, jestli se s ní jeho levá strana přitahuje, nebo odpuzuje. Jestli narazíme na magnety, které jsou stejnými póly k sobě, prostě přidáme jedničku k počtu nalezených hranic mezi komponentami.

Nakonec stačí vypsat počet hranic plus jedna a dostaneme funkční řešení. Jeho časová složitost je $\mathcal{O}(N)$, protože načteme celý vstup a každý magnet porovnáme s jeho předchůdcem, což pro každý trvá konstantní čas. Paměti spotřebujeme jen konstantně mnoho.

Program (C): `http://ksp.mff.cuni.cz/viz/26-Z1-1.c`

Michal Pokorný

26-Z1-2 Piškvorky

K vyřešení této úlohy nebyl potřeba žádný „trik“, stejně jsme se museli na každou pětici v programu podívat. Nejjednodušší způsob, jak s piškvorkovou hrací plochou pracovat, je použít dvourozměrné pole, „pole polí“. Do něj načteme celý vstup.

K samotnému počítání můžeme zvolit dva přístupy. První je velmi jednoduchý na napsání, ačkoli o trochu pomalejší, než ten druhý. V praxi na tom ale nezáleží, protože vzhledem k velikosti hrací plochy bude doba jejich běhu růst stejně rychle – jsou asymptoticky stejně složité.

Nejprve se podíváme na první způsob. Všimněme si, že přestože je 8 směrů pohybu z jednoho políčka, z dvou protilehlých musíme kontrolovat vždy jen jeden, jinak každou pětici započítáme dvakrát – jednou popředu a jednou pozadu.

Abychom nemuseli psát každý směr zvlášť (představte si, že kontrolujete 42-tice místo pětic), napíšeme si na to funkci `zkontroluj`. Ta přijme čtyři parametry: souřadnice startovního políčka `x` a `y` a směr zadaný jako `vx` a `vy` – směr pohybu v obou osách s hodnotami $-1/0/1$. Například pro kontrolu směrem doleva dolů z políčka $[5, 6]$ zavoláme funkci jako `zkontroluj(5, 6, -1, 1)`.

Tato funkce v cyklu projde všechna políčka počínaje původním a v každém kroku provede odpovídající posun. Jakmile narazí na jiný symbol než na začátku, skončí. Pokud budou všechny stejné, zvýší odpovídající počítadlo pětic.

Tuto funkci zavoláme pro každé možné počáteční políčko pro všechny směry. Jenom si musíme ověřit, že hledaná pětice celá leží na hrací ploše. Protože je to snazší dělat pro konkrétní směry než obecně, nebudeme to dělat uvnitř funkce `zkontroluj`, ale ještě předtím, než ji zavoláme.

Program (C): `http://ksp.mff.cuni.cz/viz/26-Z1-2.c`

Takto se ale na každé políčko podíváme $5\times$ z každého směru (celkově tedy $20\times$), což je zbytečně mnohokrát. Pokud bychom chtěli tento počet snížit, můžeme zkusit druhý přístup. Pro každý sloupec hrací plochy půjdeme shora dolů a budeme počítat délku souvislého úseku. V každém kroku ji zkontrolujeme, a pokud bude větší rovna 5, víme, že přibyla další pětice. Toto provedeme i pro řádky a diagonální směry, tam to už ale začne být trochu divoké. Takto se na každé políčko podíváme pouze $4\times$, jednou z každého směru.

Asymptotická časová složitost obou algoritmů je lineární k velikosti vstupu, tj. $\mathcal{O}(RS)$.

Ondra Hlavatý

26-Z1-3 Zamilovaný dopis

Kevinův problém okolo zamilovaných dopisů si nejprve zavedeme o něco formálněji, popíšeme si přirozený hladový algoritmus a na závěr dokážeme jeho správnost.

Označme posloupnost znaků milostného dopisu jako sekvenci $m_0, \ldots, m_{M-1}$. Poslopnost znaků zmoklého dopisu, o kterém chceme zjistit, zda by mohl být oním zamilovaným dopisem, si označme jako $z_0, \ldots, z_{Z-1}$. Naším cílem je zjistit, zda existuje posloupnost indexů $i_0 < i_1 < \ldots < i_{Z-1}$ taková, že pro každé $k < Z$ platí $z_k = m_{i_k}$.

Hladový algoritmus

Jako *hladové* označujeme takové algoritmy, které se ve svém rozhodování neohlíží na budoucnost, nýbrž si volí možnost, která se v danou chvíli jeví býti tou

nejlepší možnou. Takový algoritmus bývá snadné vymyslet, u úloh obtížnějších než ta naše však obvykle nefunguje, či nebývá snadné jeho správnost dokázat.

Jako příklad pravděpodobně nejznámějšího hladového algoritmu jmenujme *Kruskalův algoritmus pro hledání nejmenší kostry grafu*. Nyní už se zabývejme naším hladovým algoritmem, který je o něco jednodušší.

Budeme postupně procházet znaky zmoklého dopisu a hledat pro každý znak jemu odpovídající znak v původním zamilovaném dopisu.

Hlavní trik, díky kterému bude algoritmus efektivní, spočine v tom, že do posloupnosti znaků zamilovaného dopisu umístíme jakousi zarážku. Ta bude oddělovat zpracované a nezpracované znaky. Pozici zarážky budeme značit jako α. Na začátku položíme $\alpha = 0$.

Pozici ve zmoklém dopise bude reprezentovat k, na počátku nastavené na nulu a po každém kroku algoritmu ho zvýšíme. Skončíme, když k přesáhne délku zmoklého dopisu. V průběhu algoritmu budeme zachovávat následující invariant: Pro znaky $z_0, \ldots, z_{k-1}$ jsme již našli požadované indexy $i_0, \ldots, i_{k-1}$ a znaky před zarážkou jsou již „použité“, tedy i_k hledáme mezi znaky zamilovaného dopisu na pozicích $\alpha, \alpha + 1, \ldots$.

V jednom kroku algoritmu pak pro aktuální k hledáme index i_k takto: Posunujeme zarážku doprava (navyšováním o jedna) tak dlouho, dokud znak za ní není totožný se znakem z_k. Pokud takový znak existuje (nechť je na pozici p v zamilovaném dopise) nastavíme index i_k roven p. Dále zvýšíme k o jedna a zarážku posuneme o ještě jednu pozici doprava (to, aby znak m_p nemohl býti znovu použit).

Pokud znak nalezen není, můžeme rovnou prohlásit, že tento zmoklý dopis nemůže odpovídat původnímu dopisu. Algoritmus tedy skončí buď zamítnutím z důvodu toho, že se zarážkou dojel až na konec zamilovaného dopisu, či zkonstruuje celou posloupnost indexů a pak odpoví `ANO`.

Jak dokážeme, že náš algoritmus funguje? Pokud odpoví `ANO`, pak i zkonstruoval korektní posloupnost indexů, která dokazuje správnost odpovědi. Pokud však odpoví `NE`, víme zatím pouze to, že náš algoritmus neumí posloupnost indexů zkonstruovat. Nevíme ještě, že taková posloupnost vůbec neexistuje.

Důkaz správnosti

Pro ověření správnosti algoritmu postačí dokázat toto tvrzení: Existuje-li posloupnost indexů $j_0 < \ldots < j_{Z-1}$ taková, že pro každé $k < Z$ platí $z_k = m_{j_k}$,

pak i náš algoritmus nějakou takovou posloupnost nalezne.

Předpokládejme pro spor, že taková posloupnost $j_0 < \ldots < j_{Z-1}$ existuje, ale náš algoritmus odpověděl `NE`. Označme jako k krok, ve kterém tak algoritmus učinil. Algoritmus tedy zkonstruoval pouze posloupnost $i_0, \ldots, i_{k-1}$.

Všimněme si, že pro každé $\ell < k$ platí $i_\ell \leq j_\ell$. To vyplývá z povahy našeho algoritmu – vždy volí nejmenší možný index.

Protože $j_{k-1} < j_k$ a $i_{k-1} \leq j_{k-1}$, platí $i_{k-1} < j_k$. Tady tvrdíme něco strašného – algoritmus se zastavil, protože za pozicí i_{k-1} nenalezl znak z_k, přitom však tento znak se vyskytuje na pozici j_k, kde $j_k > i_{k-1}$, tedy leží až za i_{k-1}.

To je spor. Popsaný případ nemůže nastat, a tedy platí naše tvrzení. Správnost algoritmu byla dokázána.

Analýza efektivity

Náš algoritmus opakuje nejvýše Z-krát hlavní cyklus. Krom posouvání zarážky proběhnou všechny operace pouze jednou v rámci jedné iterace cyklu. Zarážka se sice může posunout daleko během jedné iterace, za celý běh algoritmu se však zarážka posune o nejvýše M pozic. Proto časová složitost algoritmu je $\mathcal{O}(M + Z)$, kde M je délka milostného dopisu a Z délka zmoklého dopisu. Paměťová složitost je rovněž $\mathcal{O}(M + Z)$.

Všech K dopisů pak zvládneme otestovat s časovou složitostí $\mathcal{O}(KM + Y)$, kde Y je součet délek všech zmoklých dopisů. To je pro případy, kdy je délka zmoklých dopisů zhruba stejně velká jako délka milostného dopisu, optimální. Rozmyslete si, jaké řešení zvolit v případě, že místo relativně malého počtu dlouhých zmoklých dopisů, obdržíte spoustu krátkých. Kde v tomto případě algoritmus mrhá časem?

Program (C): `http://ksp.mff.cuni.cz/viz/26-Z1-3.c`

Lukáš Folwarczný

26-Z1-4 Hroch v jezeře

Nejdříve si přeformulujme, co po nás vlastně chce zadání. Máme určit, kolik nejvíc znaků `J` obsahuje některý z ostrovů sousedící s vodní plochou, ve které se nachází hroch. Abychom se dostali k této informaci, tak ve výpočtu určitě musíme provést tyto kroky.

1. Odhalit vodní plochu, ve které se nachází hroch.
2. Identifikovat ostrovy, které s touto vodní plochou sousedí.
3. Spočítat počet znaků `J` na jednotlivých ostrovech.

Na bod 1 použijeme algoritmus, kterému se říká *prohledávání do šířky*. Algoritmus funguje tak, že na začátku dáme do fronty[6] políčko, kde začíná hroch.

Pak opakujeme následující: vyndáme první políčko z fronty, podíváme se na všechna jeho sousední políčka a z nich vybereme všechna políčka vody, která jsme ještě nenavštívili, označíme je jako navštívené a přidáme je na konec fronty. Až nám políčka ve frontě dojdou, tak máme označenou celou vodní plochu, ve které se hroch nachází, a skončíme.

Řešení bodu 2 a 3 spojíme. Budeme postupně procházet všechna políčka. Když narazíme na nějaké neoznačené políčko pevniny sousedící s označeným políčkem vody, tak z něj opět pustíme prohledávání do šířky – tentokrát na pevninu – a označíme tak jeden souvislý ostrov. Během odhalování ostrova budeme počítat, na kolik znaků `J` jsme narazili, a budeme si pamatovat maximum, kterého jsme dosáhli.

Celý algoritmus má časovou složitost $\mathcal{O}(SR)$, protože každé políčko maximálně jednou přidáme do fronty a maximálně ze čtyř různých směrů se na něj podíváme. Pak v algoritmu procházíme všechna políčka a ověřujeme, zda se nejedná o pevninu vedle označené vody – to nám také pro každé políčko zabere pouze konstantní čas.

Paměťová složitost algoritmu je rovněž $\mathcal{O}(SR)$. Pro každé políčko si musíme pamatovat, jakého je typu (pevnina, jídlo, voda, hroch) a jestli je či není označené.

Na závěr ukážeme ještě alternativní možnost, jak prohledat souvislou plochu. Jedná se o algoritmus *prohledávání do hloubky.* Tento algoritmus se od prohledání do šířky liší pouze v tom, že políčka neukládá do fronty, ale na zásobník.[7]

Postup s prohledáváním do hloubky můžete vidět ve vzorovém programu. Tam jsou dokonce všechny tři body spojeny. Prohledáváme do hloubky vodní plochu, hned jak narazíme na ostrov, tak spustíme další prohledávání na ostrov, při kterém spočítáme počet `J`, a pak zas pokračujeme v prohledávání vodní plochy.

[6] *Fronta* je datová struktura, do které můžeme přidávat prvky a zase je odebírat. Prvky jsou odebírány ve stejném pořadí, v jakém jsme je do fronty přidali.

[7] *Zásobník* je datová struktura, která vydává prvky v opačném pořadí, než jsme je do ní přidávali. Tedy první jde ven ten, co jako poslední přišel.

KSP-Z

Algoritmus s prohledáváním do hloubky má také časovou i paměťovou složitost $\mathcal{O}(RS)$.

Program (C++): `http://ksp.mff.cuni.cz/viz/26-Z1-4.cpp`

Karel Tesař

řešení

26-Z1-5 Úkol z geometrie

Ze zadání úlohy nebylo jasné, jestli se čísla v posloupnosti mohou opakovat. Stalo se tak kvůli naší chybě, a tak jsme přijímali řešení pro obě varianty. Zamýšlena byla posloupnost bez opakovaných čísel a řešení tohoto problému jsme oceňovali až 12 body. Varianta s opakováním neměla tak pěkné řešení, bylo na ní možné aplikovat pouze jednodušší a pomalejší postup, proto jsme za taková řešení rozdělovali jen po 10 bodech.

Pomalejší řešení pro opakované body

Příklad posloupnosti: $1, 10, 5, 10, 15$

Postupně čtu body posloupnosti ze vstupu a zapamatovávám si je. Řekněme, že jsem zrovna přečetl k-tý bod – B_k. Pro každý takový bod počínaje čtvrtým zkontroluji následující: Půlkružnice mezi bodem B_{k-1} a B_k nesmí protínat jinou půlkružnici mezi už přečtenými body. Proto pro každé ℓ mezi 0 a $k-2$ ověřím, jestli kružnice mezi body B_ℓ a $B_{\ell+1}$ nekříží kružnici mezi B_{k-1} a B_k (všimněte si, že první číslo z posloupnosti má index 0 a ne 1 – tak už to mají programátoři ve zvyku). To zjistíme rychle – jeden (ale ne oba) z okrajových bodů první kružnice leží mezi dvěma body druhé. Pokud tato podmínka platí, vypíšeme `ANO` a skončíme. Jinak po takovémto zpracování všech bodů vypíšeme `NE`.

Tento algoritmus má pro N bodů časovou složitost $\mathcal{O}(N^2)$, neboť pro každý z nich projde až N předchozích kružnic a pro každou z nich provede několik (konstantně mnoho) porovnání. Pamatuje si maximálně N bodů, takže má paměťovou složitost $\mathcal{O}(N)$.

Řešení pro posloupnost bez opakování

Tento případ se od předchozího liší několika věcmi. Například pokud se posledním bodem dostaneme pod nějakou půlkružnici, už z ní nemůžeme „ven“. Proto nám stačí vědět, pod jakou nejmenší půlkružnicí zrovna jsme, a okolní body mimo ni už nás nemusí zajímat. Podobně pokud jsou v posloupnosti body seřazené za sebou (např. $2, 3, 4, 5$), žádný další bod už nesmí padnout do intervalu 2 až 5. Proto nám stačí znát jeho okraje a vnitřek nás opět nemusí zajímat. Původní algoritmus pro posloupnost bez opakování tedy sice funguje, ale díky těmto pozorováním ho můžeme výrazně zrychlit.

V následujícím algoritmu si budeme pamatovat krajní body poslední půlkružnice (označíme levý P_ℓ a pravý P_r) a až dva „zakázané“ intervaly, do kterých už žádný bod nelze umístit, jinak by došlo k překřížení kružnic. První dva bo-

dy můžeme umístit kamkoli. Pro každý další bod nejdříve zkontrolujeme, jestli neleží v zakázaném intervalu. Pokud ano, vypíšeme `ANO` a skončíme.

Dále budeme rozlišovat, jestli je přidávaný bod uvnitř nebo vně poslední kružnice. Pokud je uvnitř, přidáme k intervalům $(-\infty; P_\ell\rangle$ a $\langle P_r; \infty)$. Pokud je venku, přidáme k intervalům $\langle P_\ell; P_r\rangle$. Takto budeme pokračovat, dokud nepřidáme všechny body posloupnosti. Pokud během toho nenarazíme na problém, vypíšeme `NE`.

Je důležité si uvědomit, že přidáním intervalu k zakázaným intervalům nám vždy opravdu vzniknou jen dva intervaly. To je dáno tím, že kružnice na sebe navazují a hraničními body intervalů jsou vždy okrajové body poslední kružnice. A jelikož jeden z hraničních bodů poslední kružnice bude vždy jedním z hraničních bodů předposlední kružnice, nikdy nám nevznikne „díra“, která by intervaly „rozpojila“ na tři nebo více.

Analýza efektivity

Díky tomu, že jsou intervaly jen dva, umíme v konstantním čase zkontrolovat, jestli bod leží v zakázaném intervalu, nebo intervaly v konstantním čase rozšířit. Přidáním každého z N bodů tedy strávíme jen konstantní čas a celková časová složitost tedy bude $\mathcal{O}(N)$. Pamatujeme si dva intervaly (tj. čtyři čísla, která je ohraničují) a předposlední půlkružnici (dvě čísla), takže paměťová složitost je $\mathcal{O}(1)$.

Jan „Oggy“ Škoda

26-Z1-6 Nezbední skřítci

Poslední úloha první série KSP-Z byla trochu záludnější. Hlavním cílem bylo minimalizovat počet prohazování skřítků. Jakmile máme algoritmus, který provede nejmenší možný počet prohození, měli bychom se snažit urychlit i ostatní operace.

Zadání však neříkalo úplně přesně, jak vypadá vstup. Příklad naznačoval, že dostaneme jednotlivé skřítky očíslované podle velikosti. Pokud bychom však měli pouze jejich velikosti, tak se celá úloha trochu komplikuje.

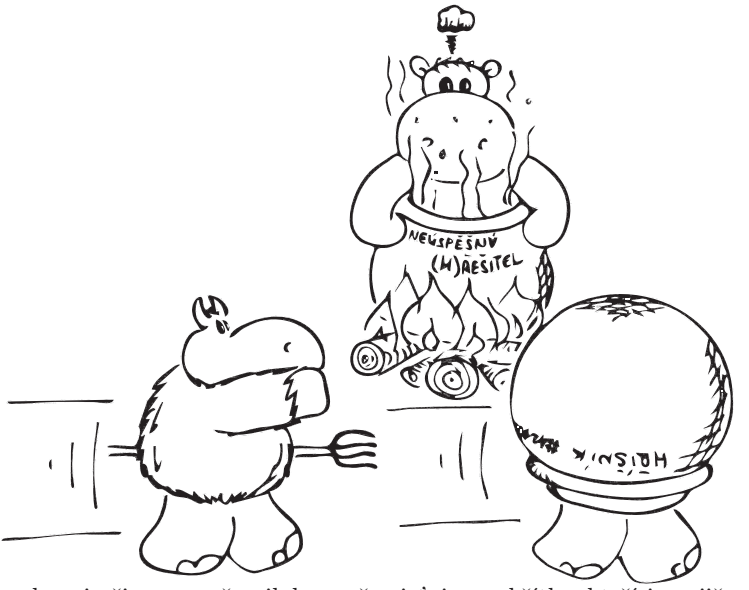
Očíslovaní skřítci

Pokud máme skřítky očíslované, můžeme celou úlohu vyřešit poměrně elegantním způsobem. O každém skřítkovi totiž okamžitě víme, kolikátý je v pořadí, a tedy i do jaké klícky patří.

Stačí nám projít postupně všechny klícky. Pokud narazíme na skřítka, který je špatně umístěný, umístíme jej do správné klícky. Tím jsme jej však vyměnili za nějakého jiného skřítka, který nemusí patřit do aktuálně zpracovávané klece. Proto budeme tento krok opakovat, dokud nenajdeme toho správného skřítka.

Program (C): `http://ksp.mff.cuni.cz/viz/26-Z1-6-poradi.c`

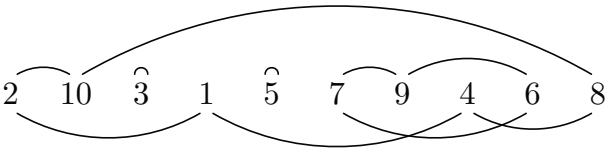
Nyní by vás mohly napadat různé otázky. Podaří se nám tímto způsobem skřítky seřadit? Nemůže se algoritmus zacyklit? Provede nejmenší možný počet prohození?



Snadno si všimneme, že nikdy nepřemisťujeme skřítky, kteří jsou již ve své klícce. Pokud tedy algortimus skončí, tak předtím musel projít postupně všechny klece. Na další se přesunul pouze tehdy, když v dané kleci byl správný skřítek.

Každou výměnou umístíme vždy alespoň jednoho skřítka do své klece. Proto nemůžeme provést více než N prohození. Algoritmus tedy musí nutně skončit, a to dokonce v čase přímo úměrném počtu skřítků $\mathcal{O}(N)$.

Rozdělme si skřítky do jednotlivých *cyklů*. Cyklus pro nás bude minimální skupina skřítků, kteří jsou promícháni pouze mezi sebou. Skřítek umístěný ve správné klícce tedy sám tvoří nejmenší možný cyklus. Názornější bude obrázek:



Co se s cykly stane, když prohodíme dva skřítky? Pokud byli ve stejném cyklu, tak tento cyklus rozdělíme na dva. Pokud naopak byli v různých cyklech, tak jejich prohozením cykly spojíme do jednoho.

Seřazení skřítkové tvoří N cyklů. Skřítky tedy lze seřadit nejlépe pomocí $N - K$ prohození, kde K je počáteční počet cyklů.

Toho však dosáhne i náš algoritmus. Vždy totiž prohazuje pouze skřítky ve stejném cyklu. Provede tedy nejmenší možný počet prohození.

Známe pouze výšky skřítků

Úloha se mírně komplikuje, pokud bychom měli na vstupu pouze velikosti jednotlivých skřítků (tedy třeba čísla z velkého rozsahu, i když samotných skřítků bude málo). Nejjednodušší bude si skřítky očíslovat a převést tím úlohu na předchozí případ.

Načteme si tedy do pole jejich výšky, a ty seřadíme libovolným rychlým algoritmem v čase $\mathcal{O}(N \log N)$. O třídicích algortimech se dočtete v tematicky zaměřené kuchařce.[8]

Tím jsme si vytvořili pomocné pole, ve kterém *binárním vyhledáváním*[9] rychle najdeme pořadí skřítka podle jeho výšky. Stačí tedy použít předchozí algoritmus s tím rozdílem, že ke skřítkovi klícku najdeme pomocí tohoto pole. Hledání nás sice trochu zbrzdí, ale celé to stihneme opět v čase $\mathcal{O}(N \log N)$.

Důležité je ještě poznamenat, že klasickými třídicími algoritmy nemůžeme třídit přímo skřítky v klíckách, protože bychom je mezi klíckami moc často prohazovali. Použitelné by ještě trochu mohlo být *třídění výběrem – SelectSort.* Tento algoritmus je však pomalejší, třídí v čase $\mathcal{O}(N^2)$.

Program (C): `http://ksp.mff.cuni.cz/viz/26-Z1-6-vysky.c`

Jenda Hadrava

[8] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

[9] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy`

KSP-Z

26-Z2-1 Had z domina

řešení

Které kostky může Kevin otočit, aby součet čísel v obou řadách hada byl sudý? Nabízí se snadné dřevorubecké řešení: vyzkoušet všecky možnosti. To v tomto případě znamená zkoušet otáčet každou kostku a průběžně kontrolovat součty. Budou-li po otočení kostky součty teček v obou řadách sudé, přičteme k výsledku jedničku.

Takto jistě nalezneme správný výsledek, ale pro každou kostku musíme sečíst N čísel v horní i dolní řadě, a těch kostek je N. Asymptotická časová složitost je tedy $\mathcal{O}(N^2)$.

Had v zadání měl největší délku 1000, takže bylo potřeba nanejvýš řádově milion výpočtů. I tímto postupem se dalo dosáhnout plného počtu bodů. S tím se ale nespokojíme.

Představte si, že by Zuzka měla milion kostek místo tisíce, pak by takový výpočet na běžném počítači trval určitě více než čtvrt hodiny. A s většími vstupy ještě déle. Souvisí to s rychlostí různých algoritmů, dočtete se o nich v naší kuchařce o složitosti.[10]

Zkusme se na to podívat jinak. Zjistíme, že skácet celý les není potřeba, nemusíme počítat všechno. Úloha se dá řešit i rychleji.

Sudost nebo lichost čísla nazýváme stručně jedním slovem *parita*. Využijeme pozorování, že součet posloupnosti čísel má stejnou paritu jako součet jejich zbytků po dělení dvěma. Stačí nám tedy sečíst místo čísel ty zbytky a nakonec zjistit zbytek celého součtu po dělení dvěma. K tomu se hodí operace modulo. Výsledek bude ta parita. (Také můžeme modulit už během sčítání, při načtení sudého čísla paritu zachovat, při načtení lichého otočit, to je to samé.)

To také znamená, že když Kevin otočí kostku se dvěma sudými čísly, parity součtů sloupců se nezmění. Vymění se jenom nula za nulu. Stejně tak se dvěma lichými čísly. A naopak, když Kevin otočí kostku s jedním lichým a jedním sudým číslem, parity se otočí, protože v jedné řadě bude o jedničku míň a v druhé o jedničku víc. Sudé číslo plus nebo mínus jedna je vždy liché, z lichého se stane sudé.

Parity obou řad kostek si spočítáme jedním průchodem. Pokud mají obě řady na začátku sudou paritu (stručně řečeno parita řad je sudá-sudá), budeme otáčet kostky se stejnou paritou (říkejme jim „stejné“), aby se sudá-sudá zachovala. Jako výsledek tedy vypíšeme jejich počet. U lichá-lichá naopak budeme otáčet „různé“ kostky, tím nám vznikne sudá-sudá, a to přesně chceme.

Ještě může nastat možnost lichá-sudá a sudá-lichá. V takových případech nemáme co otáčet. Otáčením kostek umíme paritu buďto zachovat, nebo otočit. Z toho sudou-sudou nevyrobíme, proto vypíšeme jako výsledek nulu.

[10] `http://ksp.mff.cuni.cz/viz/kucharky/slozitost`

Kolik kostek je „stejných“ a kolik „různých“ si můžeme spočítat během jednoho průchodu vstupu zároveň s počítáním parit. Podle nich vypíšeme výsledek.

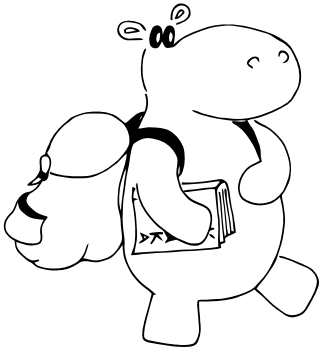
Vstup je na dvou řádcích. První řadu si tedy musíme uložit do paměti, abychom u čtení té druhé věděli, jaké bylo to první číslo na příslušné kostce. Paměťová složitost bude kvůli tomu lineární (stejně jako u dřevorubeckého řešení), časová ale jen $\mathcal{O}(N)$ místo $\mathcal{O}(N^2)$. Stačí nám totiž jen jeden průchod.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/26-Z2-1.py`

Program (C): `http://ksp.mff.cuni.cz/viz/26-Z2-1.c`

Dominik Macháček

26-Z2-2 SADO



Nejprve si připomeneme některé užitečné matematické pojmy. Nejmenší společný násobek čísel a a b je nejmenší číslo takové, že ho lze beze zbytku vydělit a i b. Největší společný dělitel je naopak největší číslo takové, které beze zbytku dělí obě čísla. Jak na ně ale v programu přijít? Na výpočet největšího společného dělitele se používá *Euklidův algoritmus*. Kdo jej nezná, nechť se podívá do ukázkového kódu k této úloze. Jak funguje a jak je rychlý, se můžete dočíst v kuchařce o teorii čísel.[11]

Na výpočet nejmenšího společného násobku podobný algoritmus neexistuje, protože není potřeba. Platí totiž vztah $a \cdot b = \mathrm{nsd}(a,b) \cdot \mathrm{nsn}(a,b)$. Na nejmenší společný násobek tedy přijdeme vydělením součinu čísel x a y na vstupu jejich největším společným dělitelem.

Kdo ovšem Euklidův algoritmus neznal, mohl na nejmenší společný násobek přijít jednoduše postupným zkoušením násobků čísla x, zda náhodou nejsou dělitelné y. To pro rychlé řešení úlohy bohatě stačilo.

A jak to celé souvisí se zadanou úlohou? Všechna čísla dělitelná zároveň x a y musí být nějakým násobkem nejmenšího společného násobku. A kolik takových čísel nalezneme v intervalu $[a,b]$ zjistíme třeba tak, že spočítáme počet násobků v intervalu $[0,b]$ a odečteme jejich počet v $[0,a-1]$. Obě hodnoty spočítáme jednoduchým dělením.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/26-Z2-2.py`

Ondra Hlavatý

[11] `http://ksp.mff.cuni.cz/viz/kucharky/teorie-cisel`

KSP-Z

26-Z2-3 Šifrovaná zpráva

řešení

Ke zdárnému vyřešení úlohy si budeme muset zkonstruovat nějakou *šifrovací tabulku*, která každému písmenu z rozsahu `A-Z` přiřadí odpovídající písmeno v zašifrovaném dopisu. Technicky to můžeme lehce zrealizovat pomocí jednoho pole o 26 prvcích – i-tý prvek bude vyjadřovat, na co se přeloží i-tý znak abecedy.

V ukázkovém programu využíváme toho, že písmena jsou v ASCII tabulce[12] umístěna za sebou a převod písmene na nějaké číslo uděláme jednoduše (bez nutnosti vědět, jaké přesně číslo v tabulce má) tím, že od něj odečteme hodnotu znaku `A`. Samotné `A` tak dostane hodnotu 0, `B` hodnotu 1 a tak dále. Jinou metodou je například použití *asociativního pole*, které nám poskytuje jazyk Python.

Když už máme technické detaily za sebou, stačí nám jen znak po znaku projít původní i zašifrovaný text a ke každému znaku původní zprávy uložit do šifrovací tabulky znak, na který je původní znak zašifrovaný. Toto by skvěle fungovalo, kdybychom měli slíbeno, že zpráva je vždy zašifrována správně. Jak však poznat chyby?

Důležité je uvědomit si, jaké chyby se nám mohou vyskytnout. Možné chyby jsou dvě. První z nich nastane, pokud stejné písmeno zašifrujeme dvakrát na písmena různá. To ale poznáme snadno, stačí nám přidat jednoduchou kontrolu u vyplňování naší šifrovací tabulky: Pokaždé, když budeme zpracovávat nějaký znak, se podíváme, jestli jsme mu už náhodou nepřiřadili nějaký znak dříve. Pokud ano a znaky jsou stejné, je vše v pořádku. Pokud se však znaky liší, zahlásíme chybu.

Druhá chyba nastane, pokud se dva různé znaky původní zprávy pokusíme zašifrovat na stejný znak zašifrované zprávy. V tomto místě nám pomůže další kontrola, ke které ale už budeme potřebovat druhou tabulku, říkejme jí třeba *tabulka použitých znaků*.

Vždy, když budeme přidávat nový záznam do šifrovací tabulky, poznačíme si do tabulky použitých znaků, že tento znak je již zabraný. Pokud pak narazíme na to, že námi chtěný znak již byl použit dříve, nezbude nám nic jiného, než také nahlásit chybu.

Mohlo by se zdát, že tím máme úlohu už zdárně vyřešenou a zašifrovanou zprávu zkontrolovanou. Ale ouha, zbývá nám ještě jedna drobnost na závěr – doplnit zbytek abecedy.

To je však už maličkost. Jen postupně projdeme šifrovací tabulku a u každého znaku, který doposud nemá přiřazený svůj zašifrovaný ekvivalent, najdeme

[12] `http://cs.wikipedia.org/wiki/ASCII`

první nepoužitý znak v tabulce použitých znaků. Ten přiřadíme a stejným způsobem doplníme celou abecedu.

Práce, kterou náš program musí udělat, je jednou projít původní i zašifrovaný text od začátku do konce a pak ještě celou abecedu. To, pokud si délku textu označíme jako N a velikost abecedy budeme brát jako malou konstantu, vede na celkovou časovou složitost $\mathcal{O}(N)$.

Program (C): `http://ksp.mff.cuni.cz/viz/26-Z2-3.c`

Jirka Setnička

26-Z2-4 Životně důležitá úloha

Připomeňme, že zadáním úlohy bylo v zadané posloupnosti o N číslech $A = a_0, a_1, \ldots, a_{N-1}$ nalézt všechny prvky s aritmetickým výskytem. Výskyt prvku x je aritmetický, pokud je posloupnost pozic, na kterých se prvek x v posloupnosti A nachází, aritmetická.

Prvky s aritmetickým výskytem mají být vypsány ve vzestupném pořadí a má u nich být uvedena i diference příslušné aritmetické posloupnosti. (Pokud má prvek jediný výskyt v celé posloupnosti A, chápeme jej jako prvek s aritmetickým výskytem o diferenci 0.)

Zdá se přirozené pevně si zvolit jeden prvek posloupnosti a ověřit, zda je jeho výskyt aritmetický. Pokud si však posloupnost A nepředzpracujeme, můžeme pro každý prvek projít celou posloupnost A (nebo její významnou část), což povede k řešení se složitostí $\mathcal{O}(N^2)$. S tou se nespokojíme.

Hodilo by se seskupit prvky posloupnosti A se stejnou hodnotou vedle sebe, abychom „aritmetičnost“ výskytů ověřovali rychle. Pokud ale posloupnost A rovnou setřídíme, ztratíme informaci o pozicích prvků a nebudeme schopni aritmetičnost výskytu ověřit.

Tento problém snadno obejdeme – nebudeme třídit pouhé prvky posloupnosti A, ale budeme třídit posloupnost dvojic. K tomu účelu si zaveďme posloupnost $S = s_0, \ldots, s_{N-1}$, v níž $s_i = (a_i, i)$. Tedy i-tá dvojice udává hodnotu prvku na i-té pozici společně s touto pozicí.

Dvojice posloupnosti setřídíme podle slovníkového uspořádání (známé také pod cizojazyčným ekvivalentem *lexikografické*). Dvojice porovnáme podle první složky a v případě shody dále podle druhé. Například setříděním posloupnosti dvojic $(1,2), (0,1), (1,0), (1,1)$ lexikograficky bychom získali posloupnost $(0,1), (1,0), (1,1), (1,2)$.

Uvědomme si, že v okamžiku, kdy definujeme vlastní funkci pro porovnávání dvojic, neliší se třídění posloupnosti dvojic nijak od třídění posloupnosti čísel. Pokud s třídicími algoritmy nejste obeznámeni, nahlédněte do naší kuchařky.[13]

[13] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

Když je posloupnost S lexikograficky setříděna, máme už výskyty každého prvku hned za sebou a ve vzestupném pořadí. Přímým průchodem pak ověříme, zda jsou posloupnosti výskytů jednotlivých prvků aritmetické.

Časová složitost třídění je $\mathcal{O}(N \log N)$, všechny ostatní části algoritmu už zvládneme v lineárním čase. Proto můžeme celkovou časovou složitost stanovit jako $\mathcal{O}(N \log N)$. Krom několika proměnných si stačí pamatovat pouze posloupnost S, proto je prostorová složitost lineární.

Závěrem si dovolíme ještě jeden mírně odlišný náhled řešení. Místo třídění posloupnosti dvojic můžeme od počátku udržovat pro každou hodnotu ze vstupní posloupnosti přihrádku. Při průchodu posloupnosti pak pro každý prvek rovnou vložíme jeho pozici do příslušné přihrádky. Na závěr zkontrolujeme, které přihrádky obsahují aritmetickou posloupnost.

Čísla v posloupnosti mohou dalece převyšovat délku posloupnosti, proto by přístup k jednotlivým přihrádkám přes pole nemusel být efektivní. Jak k nim přistupovat efektivně? K těmto účelům lze použít některou z datových struktur, které se skrývají pod souhrnným označením „asociativní pole“.

Program (C): `http://ksp.mff.cuni.cz/viz/26-Z2-4.c`

Lukáš Folwarczný

26-Z2-5 Nedopité skleničky

Cílem úlohy bylo najít v setříděné posloupnosti dvě čísla (nemusí být nutně těsně za sebou), jejichž rozdíl se co nejvíce blíží číslu K. Nejjednodušším (ale pomalým) řešením tohoto problému by bylo projít každé dvě skleničky, odečíst jejich hladiny a porovnat velikost tohoto rozdílu s K.

Jak toto realizovat? Nejdřív se hodí si skleničky očíslovat. Ve většině programovacích jazyků má první prvek pole index 0, takže budeme skleničky stejně číslovat i my (od 0 do $N-1$). Dále si stačí pamatovat, jaký rozdíl mezi dvěma skleničkami byl nejblíže ke K (označme toto číslo MIN, na začátku nechť je v něm třeba ∞) a které dvě skleničky to byly (označme S_1, S_2). Pak pro každou skleničku vyzkoušíme všech $N-1$ dalších a při zkoušení každých takových dvou skleniček porovnáme, jestli je velikost rozdílu jejich obsahů ke K bližší než MIN. Pokud ano, aktualizujeme MIN i skleničky S_1 a S_2. Řekněme, že do S_1 uložíme číslo skleničky s menším obsahem šampaňského. Až projdeme všechny dvojice skleniček, bude v S_1 řešení, tedy číslo skleničky (počítáno od nulté), do které má Kevin nalít zbytek šampaňského.

Takové řešení prochází každou dvojici skleniček a provádí na nich porovnání. Proto má časovou složitost $\mathcal{O}(n^2)$. Jak jej zrychlit? Můžeme využít seřazenosti skleniček. Všimneme si, že pokud porovnáme i-tou a j-tou skleničku a rozdíl je už moc velký, tak rozdíl mezi i-tou a $(j+1)-$ní skleničkou bude ještě větší. Nemá tedy smysl porovnávat i-tou s $(j+1)$-ní a jakoukoliv další a můžeme místo toho

i posunout o jedna dál a porovnávat $(i+1)$-ní a j-tou. Obdobně, jen obráceně, to funguje i v případě, že rozdíl je menší než K.

K realizaci použijeme dvě čísla (řekněme A a B), která budou označovat dvě aktuální pozice v posloupnosti skleniček. Tyto dvě skleničky označené čísly budeme v každém kroku porovnávat. Pokud budou skleničky očíslované od 0 do $N-1$, bude na počátku $A=0$ a $B=1$. Posloupnost skleniček si označíme jako s, tedy s_i bude množství tekutiny v i-té skleničce. Nyní můžeme procházet posloupnost s a hledat, pro která A a B bude $s_B - s_A$ co nejblíž ke K.

V každém kroku porovnáme $s_B - s_A$ s K. Pokud je $s_B - s_A > K$, zvětšíme A o 1, jinak zvětšíme B o 1. Zároveň pro každou dvojici kontrolujeme, jestli není lepší než doposud nejlepší řešení a upravujeme MIN, S_1 a S_2 – úplně stejně jako v pomalém řešení. Pokud $A=B$, tedy A „dohnalo“ B, posuneme B o 1 doprava. Pokud $B > (N-1)$, prošli jsme všechny vhodné dvojice skleniček a můžeme skončit. V S_1 bude opět uloženo řešení.

Toto řešení v každém kroku posune A nebo B alespoň o 1, vždy platí $B \geq A$ a při $B > (N-1)$ skončí. Maximálně tedy udělá $2N$ kroků. Když zanedbáme onu dvojku, už víme, že má lineární časovou složitost $\mathcal{O}(N)$. Co se paměti týče, ukládáme konstantní počet čísel: A, B, MIN, S_1, S_2, N. Připočteno k velikosti vstupu načteného do paměti nám tedy vychází lineární paměťová složitost $\mathcal{O}(N)$.

Program (Python 2): `http://ksp.mff.cuni.cz/viz/26-Z2-5.py`

Honza „Oggy“ Škoda

26-Z2-6 Čtecí hlavy

Řešení této úlohy se řídí myšlenkou: „Pokud existuje jen málo možností, tak je vyzkoušíme všechny a z nich vybereme tu nejlepší.“ To je v informatice poměrně častý postup. Při aplikaci tohoto pravidla si však musíme dát pozor, obecně totiž může být zkoušení všech možností velmi neefektivní. Když se nám ale povede nalézt jen málo možností, mezi kterými určitě bude i ta úplně nejlepší, máme vyhráno.

Při čtení pomocí jedné hlavy máme pouze dvě možnosti. Buď hlava nejdříve pojede směrem k nejlevějšímu segmentu a pak směrem k nejpravějšímu segmentu, a nebo naopak nejdřív k nejpravějšímu a pak k nejlevějšímu. Z těchto možností vybereme tu lepší a máme výsledek. Časová složitost tohoto výpočtu je konstantní, ale časová složitost celého algoritmu je lineární (konkrétně $\mathcal{O}(N)$, kde N je počet čtených segmentů), protože musíme načíst vstup a zjistit, který segment je nejlevější a který nejpravější.

Při čtení pomocí dvou hlav máme také jen málo možností. Všimneme si, že v optimálním řešení levá hlava přečte nějaký počáteční úsek vybraných segmentů a pravá hlava přečte zbytek vybraných segmentů, které zas tvoří nějaký koncový

úsek vybraných segmentů. Tyto dva úseky pokrývají dohromady všechny vybrané segmenty a nekříží se.

Bude mezi takovými řešeními určitě i to optimální? Ano, bude! Pokud by se obě hlavy měly po cestě křížit, tak namísto toho budeme uvažovat situaci, kdy se od sebe hlavy odrazí a to, co původně měla jet jedna hlava, pojede druhá hlava a naopak. Není tedy výhodné, aby si hlavy vyměnily pozice – levá hlava tedy celou dobu zůstane levou a pravá hlava pravou.

A co úsek čtený levou hlavou a úsek čtený pravou hlavou? Může být výhodné jejich překrytí? Také ne, protože nemá smysl, abychom nějaký segment četli dvakrát.

Stačí nám tedy vyzkoušet všechny možnosti. Pro souřadnice segmentů $s_1 < s_2 < \cdots < s_N$ vyzkoušíme možnosti, kdy levá hlava bude číst segmenty $s_1, \ldots, s_i$ a pravá hlava bude číst segmenty $s_{i+1}, \ldots, s_N$. Pozor, nesmíme zapomenout na možnosti, kdy levá resp. pravá hlava čte všechny segmenty.

Za jak dlouho hlava přečte konkrétní úsek segmentů zjistíme pomocí výpočtu pro jednu hlavu. Tedy hlava buď nejdříve pojede k nejlevějšímu a pak k nejpravějšímu segmentu čteného úseku, nebo naopak. Celý algoritmus má časovou složitost $O(N)$, máme dohromady $N + 1$ možností a výpočet každé z nich zabere konstantní čas. Pokud bychom souřadnice segmentů na vstupu nedostali setřízené, museli bychom je v čase $\mathcal{O}(N \log N)$ setřídit.

Paměťová složitost varianty pro jednu hlavu je konstantní (značíme $\mathcal{O}(1)$) – stačí nám si pamatovat pouze souřadnici nejlevějšího a nejpravějšího segmentu. Paměťová složitost varianty pro dvě hlavy je lineární (značíme $\mathcal{O}(N)$) – pamatujeme si N souřadnic segmentů a pak několik málo (konstantně) dalších pomocných proměnných.

Pro lepší představu řešení nahlédněte do okomentovaného zdrojového kódu. Zdrojový kód je v jazyce C++, jeho znalost by však neměla být nutná pro pochopení programu.

Program (C++): `http://ksp.mff.cuni.cz/viz/26-Z2-6.cpp`

Karel Tesař

26-Z3-1 Zámky labyrintu

Stejně tak, jako zadání znělo jednoduše, řešení bude též jednoduché.

Nejprve si musíme rozmyslet, na jaké hodnoty bychom potenciálně měli nastavit a, b a c.

Proměnná a musí splňovat (jak bylo uvedeno v zadání) rovnost $b - a = c - b \Rightarrow a = -(c - b - b) = 2b - c$. Pro b a c vyjdeme ze stejné rovnice a vyjde nám, že $b = \frac{c+a}{2}$ (zde se jen musí ověřit, že toto číslo je celé) a $c = 2b - a$.

Tedy stačilo spočítat tyto hodnoty, a pak zjistit, kde je v absolutní hodnotě nejmenší rozdíl s původní hodnotou.

Program (C): `http://ksp.mff.cuni.cz/viz/26-Z3-1.c`

Vojta Sejkora

26-Z3-2 Čarodějova šifra

Než jsme se mohli pustit do samotného šifrování mřížkou, bylo nutné poradit si se situací, kdy jsme měli text kratší, než kolik byla velikost mřížky ($K \times K$). Dalo by se to řešit nějakou soustavou podmínek až přímo při šifrování, ale proč si to komplikovat?

Nejjednodušší řešení jsou často ta nejlepší – prostě si na začátku doplníme text na požadovanou délku opakováním písmen „KSP“. Na technické detaily, stejně jako na detaily načítání vstupu se můžete podívat do vzorových programů na konci tohoto řešení.

Tím se dostáváme k hlavní části celé úlohy, k šifrování mřížkou. Řešme nejdříve jednodušší případ bez otáčení a pak zkusme stejný princip zkombinovat s otáčením.

Máme tedy dvourozměrné pole s mřížkou a text. Budeme si udržovat pozici v textu – jaké písmeno bychom měli zapsat dál – a budeme postupně po sloupcích a řádcích procházet dvourozměrné pole s mřížkou (vnější cyklus přes řádky, vnitřní přes sloupce, abychom šli správně zleva doprava řádek po řádku). Vždy, když narazíme na díru, zapíšeme na stejnou pozici ve výsledném dvourozměrném poli písmenko, které je zrovna na řadě (a posuneme ukazatel na další). Tak postupně projdeme celou mřížku a máme jednu čtvrtinu úkolu splněnou.

Teď by se nám líbilo mřížku otočit a to celé provést znova. Můžeme si buď celou mřížku nakopírovat a otočit, nebo můžeme transformovat souřadnice až při jejím otáčení. Je důležité si uvědomit, že když mřížku otočíme doprava (po směru hodinových ručiček), je to to samé, jako když souřadnice transformujeme na druhou stranu, doleva. Když se ptáme, co je na souřadnicích $[r, s]$ v mřížce otočené o jedna doprava, můžeme náš dotaz přeložit na ekvivalentní dotaz: co se nachází na souřadnicích otočených o jedna doleva v původní mřížce.

KSP-Z

řešení

Ať už se rozhodneme pro jeden nebo druhý způsob, budeme potřebovat nějaký přepočet souřadnic pro otočení, neboli na jaké pozici se octne to, co bylo původně na souřadnicích $[r, s]$. Ukážeme vzorce pro otočení doprava, pro otočení doleva je to stejné, jen odzadu (jedno otočení doleva je to samé jako tři otočení doprava). Indexujeme tabulku od nuly, takže máme sloupce i řádky s indexy $0, \ldots, K-1$.

1. otočení doprava: $[r, s] \to [s, K-1-r]$
2. otočení doprava: $[r, s] \to [K-1-r, K-1-s]$ (vlastně první otočení aplikované dvakrát)
3. otočení doprava: $[r, s] \to [K-1-s, r]$ (to samé, jen třikrát)

Teď již máme všechny potřebné stavební kameny. Stačí jen čtyřikrát provést zapsání přes mřížku a mezitím otáčet (ve vzorových programech používáme variantu s transformací souřadnic). Díky vlastnostem mřížky slíbeným v zadání jsme zapsali na každé políčko výsledné tabulky právě jeden znak a tím jsme splnili čarodějovo zadání, stačí mu tedy mřížku odevzdat (vypsat) a můžeme jít dovádět do Labyrintu snů.

Program (Python): `http://ksp.mff.cuni.cz/viz/26-Z3-2.py`

Program (C): `http://ksp.mff.cuni.cz/viz/26-Z3-2.c`

Jirka Setnička

26-Z3-3 Hádanka

Nejdříve si připomeneme kritérium dělitelnosti devíti, které asi všichni znají: číslo je dělitelné devíti beze zbytku právě tehdy, když ciferný součet jeho zápisu (v desítkové soustavě) je také dělitelný devíti. Například číslo 738 je dělitelné devíti právě proto, že ciferný součet $7 + 3 + 8 = 18$ devíti dělitelný je.

Než se pustíme do samotné úlohy, dovolím si drobnou otázku. Zamýšleli jste se někdy nad tím, proč toto kritérium funguje? Pokud ne, ukážeme si to. Vezmeme si naše známé číslo 738, budeme ho postupně dělit devíti a budeme sledovat přenosy mezi řády.

Rozepíšeme si číslo po řádech na $7 \cdot 100 + 3 \cdot 10 + 8 \cdot 1$ a budeme ho postupně zkoušet dělit od největšího řádu (jako bychom popořadě dělili jednotlivé členy čísly 900, 90 a 9). Dělení 900 nám nic nezmění, ale už dělení 90 je zajímavé. To nám zruší první člen a z každé stovky nám zbude právě jedna desítka, tedy přičteme 7 k desítkám (dostáváme tak $10 \cdot 10 + 8 \cdot 1$). Když budeme dál dělit 9, zmizí nám desítkový člen a z každé desítky nám zůstane právě jedna jednička, dostaneme tedy výraz $18 \cdot 1$ a u něho už dělitelnost devíti snadno poznáme.

Asi jste si už všimli jisté pravidelnosti. Je to dáno tím, že číslo v n-ární (přesněji desítkové) soustavě dělíme číslem $n-1$ (tedy devítkou). Z každého řádu se nám tak přenese do nižšího právě takové číslo, které v něm je. A všechny

tyto přenosy se pak shromáždí v jednotkovém řádu, což je přesně ekvivalentní cifernému součtu.

Potom, co jsme si dokázali dělitelnost devíti, přejděme už k řešení samotné úlohy. V podstatě nám stačí na místa otazníků doplnit taková čísla, aby nám ciferný součet vyšel dělitelný devíti. Zároveň ale chceme, aby číslo bylo co nejmenší možné, takže chceme umisťovat co nejmenší čísla do velkých řádů.

S výjimkou pozice na začátku čísla, kde musí být alespoň jednička, můžeme jinak zkusit všude místo otazníků doplnit nuly. Pokud nám potom vyjde ciferný součet dělitelný devíti, vyhráli jsme, pokud ale ne, bude ještě nutné číslo upravit.

Stačí si ale spočítat, kolik nám schází do nejbližšího dalšího čísla dělitelného devíti, to je maximálně osm. Takže nám stačí nějaké zapsané číslo (nulu nebo jedničku) zvětšit o osm. A protože chceme výsledné číslo co možná nejmenší, provedeme to u nejméně významného řádu – u nejpravější pozice s otazníkem.

Všechno, co potřebujeme, je několikrát projít všechny cifry čísla. Takže pro číslo dlouhé N cifer zabere náš postup čas $\mathcal{O}(N)$. Nahlédněte do vzorových programů.

Program (Python): `http://ksp.mff.cuni.cz/viz/26-Z3-3.py`

Program (C): `http://ksp.mff.cuni.cz/viz/26-Z3-3.c`

Jirka Setnička

26-Z3-4 Tvar labyrintu

Struktuře křižovatek a cest mezi nimi se v informatice říká *graf*, a takto speciálnímu grafu bez cyklů potom *strom.* Problém ze zadání je tedy problém nalezení nejdelší cesty (trasy) ve stromu.

Představme si křižovatky a slepé cesty jako uzlíky a cesty mezi nimi jako provázky odpovídající délky. Potom celý labyrint uchopme za křižovatku s číslem 0 a nechme provázky s uzlíky volně spadnout dolů. Pokud si to dokážete představit, můžete si všimnout následujících faktů. Nejdelší cesta bude vždy končit ve slepé chodbě – pokud do křižovatky vedou alespoň dvě chodby a jednou z nich přijdeme, tak není důvod se zastavovat, když můžeme druhou odejít. Další pozorování je, že uzlík, který spadnul nejníže (je tedy nejvzdálenější od křižovatky 0), bude určitě na kraji té nejdelší cesty. Kdybychom poté celý strom chytili za tento nejhlubší uzlík a zase nechali ty ostatní volně spadnout dolů, už bychom snadno nalezli nejdelší cestu.

Jak to ale udělat algoritmicky? Nalézt nejhlubší uzlík zvládneme průchodem grafu do hloubky, problém je ale s „přetočením“ stromu pod nejhlubší uzlík. Proto si vysvětlíme snadněji naprogramovatelný postup.

K tomu si zavedeme několik pojmů, vyjdeme u nich z představy zavěšených uzlíků. *Podstrom* začínající v nějakém uzlíku u je strom obsahující tento uzlík

a vše, co visí pod ním, uzlík u je *kořenem* tohoto podstromu. *Svislá cesta* je pak taková trasa, která v tomto zavěšení vede pouze dolů. Stojí za všimnutí, že libovolnou jinou cestu můžeme získat složením dvou svislých cest.

řešení

Při zpracování každé křižovatky uvažujeme pouze podstrom s kořenem v této křižovatce. Spočítat chceme dvě věci: jednak délku nejdelší svislé cesty, jednak délku nejdelší cesty procházející kořenem (tedy zpracovávanou křižovatkou).

Rekurzivně získáme délky nejdelších svislých cest vedoucích ze sousedních křižovatek, k nim přičteme vzdálenost k příslušné křižovatce. Maximum z těchto hodnot představuje délku nejdelší svislé cesty, součet dvou největších hodnot je pak délkou nejdelší cesty procházející přes kořen. Pro úplnost dodejme, že nejdelší svislá cesta vycházející ze slepé uličky má délku 0.

Pokaždé, když počítáme cestu procházející přes kořen, porovnáme navíc výsledek s globálně udržovaným maximem, a je-li větší, toto maximum upravíme. Po zpracování celého stromu v něm tak máme hledanou délku nejdelší cesty ve stromu.

Ještě bychom si měli rozmyslet složitost. Do každé křižovatky určitě přijdeme jen jednou (do každé vede shora maximálně jedna chodba). Jejím zpracováním strávíme jednak nějaký konstantní čas, jednak nějaký další konstantní čas za každou chodbu, která z křižovatky vede dolů. Všimněme si ovšem, že když do každé křižovatky vede nejvýše jedna chodba, je chodeb nejvýše N. Dohromady tak potřebujeme čas $\mathcal{O}(N)$. Podobně paměťová složitost je lineární.

Program (C): `http://ksp.mff.cuni.cz/viz/26-Z3-4.c`

Ondra Hlavatý & Karolína „Karryanna“ Burešová

26-Z3-5 Ceny na střelnici

Pomalu, ale jistě

Hledáme v zadané posloupnosti a co nejdelší úsek, ve kterém jsou všechny prvky různé. Zkusíme nejdřív najít nejdelší takovýto úsek, který začíná na prvním prvku, tedy zcela vlevo.

To je velice jednoduché, stačí si nějak v paměti udržovat, které všechny různé prvky jsme už potkali. Pokud budeme mít druhy cen označené přirozenými čísly, tak se na to skvěle hodí pole. Vytvořme si pole c, ve kterém na začátku budeme mít všude nuly (to znamená, že jsme zatím nic nenavštívili), a pokud potkáme cenu označenou i, napíšeme do c nějaké nenulové číslo na $c[i]$.

No a pokud budeme chtít zapsat na místo, kde už ale něco nenulového je, znamená to, že jsme právě přečtenou hodnotu potkali už podruhé. Takže úseky od začátku na místo, na které se budeme dívat, od teď budou určitě obsahovat dvojici stejných prvků.

Ale tím nesmíme ukončit hledání, ještě je možnost, že existuje nějaký vyhovující úsek začínající někde dál než na prvním prvku. Nejjednodušší způsob

by byl použít stejný algoritmus od druhého prvku, potom od třetího, potom od čtvrtého a tak dále až do N-tého. To by ale trvalo moc dlouho – algoritmus spustíme N-krát a může zkoumat až N prvků, takže časová složitost bude $\mathcal{O}(N^2)$. Pro první úkol ale stačí, protože pokaždé algoritmus skončí po nejvýše padesáti krocích (delší úsek různých cen nemůže být). Proto bude časová složitost $\mathcal{O}(50N) = \mathcal{O}(N)$.

Optimalizace

My ho ale dokážeme výrazně zrychlit jednoduchým trikem: Když budeme do pole chtít napsat, že jsme potkali nějakou cenu x na pozici i, na $c[x]$ zapíšeme hodnotu i. Tedy o druhu ceny x budeme vědět nejen, že jsme ho potkali, ale i kde to naposledy bylo.

Algoritmus bude postupovat tak, že bude procházet posloupnost cen a u každé zjistí, kde začíná nejdelší úsek různých cen, který v ní končí. Pokud prozkoumáme novou cenu, tento začátek nejdelšího úseku zůstane buď stejný jako u předchozí, nebo se posune někam směrem doprava.

Když přijdeme k nějaké ceně x na pozici i, zjistíme, jestli je její poslední výskyt před nebo za začátkem nejdelšího úseku končícího na pozici $i-1$. Pokud je před ním, můžeme ho ignorovat a začátek bude pro i stejný jako pro $i-1$. Pokud bude za ním, posune se tím začátek nejdelšího úseku doprava těsně za poslední výskyt.

Tímto způsobem projdeme postupně celou posloupnost a budeme si udržovat zatím nejlepší délku úseku, jakou jsme dosud viděli. Spolu s ní si zapamatujeme i indexy jejího začátku a konce. Ty přepíšeme, jakmile potkáme nějakou lepší.

Proč to celé funguje?

Sluší se ještě dokázat, že jsme žádnou ještě lepší validní posloupnost nepřehlédli. Na každém prvku posloupnosti jsme znali nejdelší posloupnost, která na tomto místě končí (kdybychom se pokusili její začátek posunout o jedno místo doleva, objevila by se mezi prvky posloupnosti nějaká dvojice – právě na takové místo jsme začátek posouvali).

Konec posloupnosti jsme v průběhu algoritmu posouvali postupně o jedničku, proto, ať by nejlepší validní podposloupnost končila kterýmkoliv prvkem, tak bychom přes něj museli přejít a podposloupnost bychom tak našli.

Jak dlouho to celé poběží? Uděláme N kroků, kde krok je všechno, co uděláme mezi jednotlivými posunutími konce posloupnosti. Tam ale uděláme vždy konstantní počet operací, takže časová složitost bude $\mathcal{O}(N)$.

Program (Python): `http://ksp.mff.cuni.cz/viz/26-Z3-5.py`

Martin Španěl

KSP-Z

26-Z3-6 Horská dráha

řešení

Hlavním nástrojem, který využijeme v této úloze, je třídění. O rychlých třídících algoritmech se dočtete třeba v naší kuchařce.[14]

V prvním úkolu stačí všechny hodnoty, které si Kevin zapsal, vzestupně setřídit. Poté budeme zleva doprava hodnoty procházet a udržovat si přitom počítadlo, které bude obsahovat délku souvislého úseku složeného ze stejných hodnot, ve kterém se právě nacházíme. Pokud je následující hodnota stejná jako předchozí, počítadlo o jedna zvětšíme, jinak jej vynulujeme. Zároveň si v průběhu udržujeme dosavadní maximální hodnotu počítadla, kterou stačí aktualizovat vždy před vynulováním a pak na konci.

Výpočet se skládá ze dvou fází: třídění a následný průchod. První fáze nás stojí při použití rychlého třídícího algoritmu $\mathcal{O}(N \log N)$ času, druhá fáze už jen $\mathcal{O}(N)$, takže celková časová složitost tohoto algoritmu je pak $\mathcal{O}(N \log N)$.

Budeme předpokládat, že se horská dráha chová rozumně spojitě, tedy pokud jsme přejeli z výšky a do výšky b, ocitli jsme se přitom jednou v každé výšce mezi a a b. Navíc úsek mezi každými dvěma Kevinovými zápisy je ze zadání celý z kopce nebo celý do kopce, takže každý úsek mezi dvěma zápisy můžeme popsat intervalem, jehož krajními hodnotami jsou ony zapsané hodnoty.

Rádi bychom nyní řekli, že hledaná výška je taková, která je obsažena v nejvíce intervalech. To je ovšem zrádné, protože místa, ve kterých si Kevin zapisoval výšku, jsou zachycena ve dvou intervalech. Lokální výškové extrémy tedy určitě nebudou dobrými kandidáty na hledanou nejčastěji navštěvovanou výšku. Představme si třeba obyčejnou sinusoidu, na které by se Kevin vyskytl ve výšce 0 mnohem častěji než ve výšce 1. Raději bychom proto počítali s otevřenými intervaly.

Učiníme pozorování, které nám to umožní: představme si, že jsme již nalezli onu výslednou výšku, ale vyskytuje se na seznamu, tedy je koncovým bodem některých intervalů. Bez újmy na obecnosti předpokládejme, že se tato výška vyskytuje v nejvýše tolika lokálních údolích jako vrcholech (jinak pokračujeme obráceně). Pak ovšem můžeme tuto výšku zmenšit o dostatečně malinké číslo, čímž sice už nebude v údolích, ale za každý vrchol teď budeme mít dva výskyty této výšky v jeho blízkém okolí. Jeden bude nalevo a jeden napravo. Celkový počet výskytů se tedy nezmenšil a dostali jsme výšku, která leží pouze v otevřených intervalech (volbou dostatečně malého zmenšení jsme se mohli všem ostatním lokálním extrémům vyhnout).

Nyní nám stačí pracovat s otevřenými intervaly a budeme postupovat podobně jako v první úloze. Vezmeme si všechny konce intervalů, přidáme ke každému konci příznak, zda je to konec levý nebo pravý, a všechny konce vzestupně setřídíme.

[14] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

Vzorová řešení KSP-Z – 3. série

Setříděné pole budeme procházet zleva doprava a udržovat si, v kolika jsme zrovna intervalech. Pokud narazíme na levý konec, počítadlo o jedna zvýšíme, pokud narazíme na pravý, o jedna jej zmenšíme. Přitom si udržujeme maximální hodnotu počítadla tentokrát ještě navíc s intervalem, ve kterém jí bylo dosaženo. Ten vznikne jako průnik nějakých intervalů, při výpočtu jej však snadno dostaneme jako největší levý a nejmenší pravý konec, který aktuálně máme.

Nakonec máme interval (a, b), který reprezentuje výšky, ve kterých jsme se nejčastěji objevili. Stačí tedy vypsat libovolnou z nich, třeba $(a + b)/2$, což je určitě číslo z vnitřku intervalu.

Časová složitost algoritmu je stejná jako v prvním úkolu, tedy $\mathcal{O}(N \log N)$.

Program (C++): `http://ksp.mff.cuni.cz/viz/26-Z3-6.cpp`

Mark Karpilovskij

KSP-Z

26-Z4-1 Vražedná čísla

řešení

Místo samotných vražedných čísel zaměříme svou pozornost na jejich zbytky po dělení číslem Q. Aby se nám s nimi pohodlně pracovalo, zavedeme si následující (vcelku obvyklé) značení: je-li x nějaké celé číslo, bude $x \bmod Q$ jeho zbytek po dělení Q. Tento zbytek je také celé číslo a leží mezi 0 a $Q-1$.

Nyní se podívejme na součet nějakých dvou čísel $x+y$. Kdy je tento součet dělitelný Q? Buď tehdy, jsou-li obě čísla také dělitelná Q (čili $x \bmod Q = y \bmod Q = 0$), nebo dají-li jejich zbytky dohromady Q, tedy

$$(x \bmod Q) + (y \bmod Q) = Q.$$

Počítejme nejprve dvojice prvního druhu. K tomu nám stačí spočítat, kolik z vražedných čísel je dělitelných Q. Pokud jich je z_0, můžeme utvořit přesně $z_0 \cdot (z_0 - 1)/2$ dvojic. Proč právě tolik? Představme si, že na chvíli budeme odlišovat, které číslo ve dvojici je první a které druhé. Máme z_0 možností, jak zvolit první číslo, a pak $z_0 - 1$ možností, jak doplnit druhé (o jedničku méně proto, že nesmíme totéž číslo použít dvakrát). Teď jsme ovšem každou dvojici započítali dvakrát, takže výsledek ještě vydělíme dvěma.

Pokračujme dvojicemi druhého druhu. Označme z_i počet vražedných čísel, která dávají zbytek i. Dvojici druhého druhu získáme tak, že spárujeme číslo se zbytkem 1 s číslem se zbytkem $Q-1$, nebo 2 s $Q-2$, atd. Takže napočítáme celkem

$$z_1 z_{Q-1} + z_2 z_{Q-2} + \ldots$$

dvojic. Kde se přesně zastavíme? Za $z_{Q/2} z_{Q/2}$ už pokračovat nesmíme, protože bychom opět dvojice počítali podruhé. Ale i samotný člen $z_{Q/2} z_{Q/2}$ je zákeřný – objeví se jen tehdy, je-li Q sudé číslo, ale pokud tomu tak je, jsme ve stejné situaci jako u dvojic prvního druhu, neboť kombinujeme čísla se stejným zbytkem. Takových dvojic bude $z_{Q/2} \cdot (z_{Q/2} - 1)/2$.

Pojďme shrnout, co jsme vymysleli. Pro liché Q je výsledek roven

$$\frac{z_0 \cdot (z_0 - 1)}{2} + z_1 z_{Q-1} + z_2 z_{Q-2} + \ldots + z_{\frac{Q-1}{2}} z_{\frac{Q+1}{2}},$$

pro sudé Q pak

$$\frac{z_0 \cdot (z_0 - 1)}{2} + z_1 z_{Q-1} + z_2 z_{Q-2} + \ldots + \frac{z_{Q/2} \cdot (z_{Q/2} - 1)}{2}.$$

Program pouze spočítá, kolik čísel dává který zbytek, a pak to dosadí do našeho kouzelného vzorečku. Obojí jistě stihne v lineárním čase a lineární paměti.

Dodejme ještě, že ve většině programovacích jazyků si musíme dávat pozor na zbytky po dělení záporného čísla. Často totiž vyjdou záporně: například $(-8) \bmod 7 = -1$. V programu je proto jistější psát $((x \bmod Q) + Q) \bmod Q$,

což pro kladné x neuškodí a pro záporné to výsledek posune do kladné hodnoty, která se nám hodí více.

Program (C): `http://ksp.mff.cuni.cz/viz/26-Z4-1.c`

Program (Python 3): `http://ksp.mff.cuni.cz/viz/26-Z4-1.py`

Ondřej Hlavatý & Martin „Medvěd“ Mareš

26-Z4-2 Sbírání vajíček

Kam se má Kevin s košíkem postavit, aby se Zuzka co nejméně naběhala?

Na přímce je vzdálenost bodů a a b absolutní hodnota jejich rozdílu, $|a-b|$. Třeba body 3 a 10 jsou vzdálené $|3-10|=7$.

Hledáme místo, které má nejmenší možný součet dvojnásobků vzdáleností od zadaných bodů na přímce. Proč dvojnásobků? Jednou počítáme cestu od Kevina k vajíčku a jednou od vajíčka ke Kevinovi. Obě jsou stejně dlouhé. Když chceme vybrat místo s nejmenším součtem, násobení dvěma ale vůbec nemusíme uvažovat. Tak to bude pro nás při dokazování příjemnější. Každý sčítanec v součtu je násoben dvěma, můžeme dvojku vytknout před součet. Vždy, když porovnáme dvě ušlé vzdálenosti, dvojku z nerovnice odstraníme, protože $2a < 2b$ je to samé jako $a < b$.

Pro každé místo na zahradě tedy můžeme sečíst vzdálenosti od vajíček, a vybrat to místo, kde je nejnižší součet. Toto nám dá správný výsledek, ale bude to pomalé. Možných míst je velmi mnoho.

Jak zjistit rychleji, kam postavit Kevina?

Představme si, že už máme zjištěný součet pro nějaké místo A, od tohoto místa je 5 vajíček napravo a 10 nalevo. Teď Kevina posuneme o 3 políčka doleva do bodu B, mezi A a B se bez újmy na obecnosti žádné vajíčko nenachází. (Kdyby tam bylo, za B zvolíme to nejbližší místo, kde to vajíčko je, a tak mezi A a B už žádné nebude. B už sice nebude o tři políčka od A, ale to nevadí, následující odstavec bude platit obdobně i pro jinou vzdálenost než 3.)

Jak se změní součet? Zvětší se o třikrát tolik, kolik bylo vajíček od A vpravo, to je o $3 \cdot 5$. Zároveň se ale zmenší o třikrát tolik, kolik vajíček bylo od A nalevo, to je o $3 \cdot 10$, protože k nim se Kevin přiblížil. Součet od B je tedy menší než od A, takže to je o něco vhodnější místo.

Tímto postupem tedy pro každý bod, od kterého je více vajíček na jedné straně než na druhé, umíme najít místo s lepším součtem vzdáleností. Žádné takové tedy nebude to správné.

Nejlepší je naopak místo, které má stejně vajíček nalevo jako napravo. Je-li vajíček lichý počet, je toto místo jen jedno, Kevin by si měl stoupnout přímo na prostřední vajíčko. Pokud jich je sudý počet, je to jakékoliv místo mezi dvěma prostředními vajíčky, tedy mezi vajíčky číslo $N/2$ a $N/2 \pm 1$ (plus, pokud inde-

KSP-Z

xujeme od jedné, mínus, pokud od nuly). Vybrat můžeme třeba jedno z nich. Prvek na prostřední pozici se obvykle nazývá *medián* posloupnosti.

řešení

A je to jasné. Jako řešení stačí vypsat medián. Existuje zaručeně lineární algoritmus na jeho nalezení, je popsán v kuchařce.[15] Nám ale stačí čísla uložit do pole, setřídit v čase $\mathcal{O}(N \log N)$, sáhnout doprostřed a vypsat medián.

Vzorové řešení je na pět řádků, viz ukázkový program.

Program (Python 3): `http://ksp.mff.cuni.cz/viz/26-Z4-2.py`

Dominik Macháček

26-Z4-3 Hra Othello

Pojďme si nejprve rozmyslet, jak bychom takovouto úlohu řešili s tužkou a papírem. K tomu se nám hodí nakreslit si diagram všech možných průběhů hry (viz obrázek vpravo).

Na začátku má Kevin na výběr ze tří možných tahů, poté Petr ze dvou a poslední tah už je Kevinovi předurčen. Zadání říká, že oba hráči hrají optimálně. Jak z toho poznáme, kdo kam potáhne?

Začněme jednodušší otázkou: jak by se úloha řešila, kdyby zbývaly už jen dva tahy do konce (a na řadě byl tedy Petr)? Petr má na výběr za dvou možností, kam táhnout. Vybere si pochopitelně tu, ve které získá Kevin na konci *méně* kamenů. Např. v pozici[16] C si vybere tah do pozice H, ze které Kevin musí táhnout do N a skončit tak s 6 kameny (číslo pod deskou v závorce), kdežto kdyby si vybral G, bude mít Kevin 10 kamenů. Tučné šipky v našem schématu značí vždy tah, který by si hráč z dané pozice vybral.

Nyní pro každou z pozic dva tahy před koncem (B–D) víme, kam Petr potáhne a jak hra dopadne. Proto si k těmto pozicím můžeme poznamenat *ohodnocení* $o(X)$, tedy číslo říkající, s kolika kameny Kevin skončí, vyjdeme-li z této pozice a oba hráči budou hrát optimálně. Ohodnocení pozic najdete v našem diagramu jako čísla v závorce pod deskou.

Nás ovšem zajímá výsledek celé hry, tedy ohodnocení pozice A. No ale to už je teď snadné určit! Víme, že pokud si Kevin vybere například tah do C, získá $o(\mathrm{C}) = 6$ kamenů (neb ohodnocení přesně popisuje, jak dopadne zbytek hry od C do konce). Kevin si tedy samozřejmě vybere tah s *nejvyšším* ohodnocením. Jinými slovy $o(\mathrm{A}) = \max(o(\mathrm{B}), o(\mathrm{C}), o(\mathrm{D}))$.

Tomu, co jsme právě popsali, se obvykle říká *minimaxový algoritmus*, a je to asi nejznámější herní algoritmus vůbec. Myšlenka je jednoduchá: dosažitelným

[15] `http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj`

[16] Pozicí zde rozumíme kompletní stav hry v daný okamžik, tedy umístění a barvu všech kamenů, spolu s informací, kdo je na tahu.

pozicím postupně směrem od koncových přiřazujeme ohodnocení, které říká, s jakým „skóre“ hra skončí, budou-li oba hráči hrát optimálně. Jeden hráč (v našem případě Kevin) usiluje o co nejvyšší skóre, druhý o co nejnižší. Roli skóre v naší hře zaujímá počet Kevinových kamenů.

Pozici můžeme ohodnotit ve chvíli, kdy známe ohodnocení všech pozic, na které z ní lze táhnout, a ohodnocení je buď minimem, nebo maximem (proto „minimax“) z ohodnocení těchto pozic, podle toho, který hráč je na tahu.

Nyní už zbývá jen to naprogramovat. Postup je schématicky znázorněn na obrázku na následující straně.

1. Pro každou ze tří možností prvního Kevinova tahu:
2. Pro každou ze dvou možností Petrova tahu:
3. Vytvoř novou kopii desky.
4. Odehraj na ní tyto dva tahy a nutný Kevinův poslední.
5. Spočítej Kevinovy kameny na zaplněné desce (skóre).
6. Vyber ze těchto dvou možných skóre minimum.
7. Vyber z těchto tří minim to největší a prohlas ho za výsledek.

Samotný minimax je jen několik vnořených cyklů. Dále potřebujeme umět simulovat průběh hry. K tomu si desku uložíme jako dvourozměrné pole znaků a pro každý tah ji upravujeme přímo následujíce pravidla. Ta jsou v zadání už víceméně v algoritmické podobě, pročež je stačí jen „přeložit“ do vašeho oblíbeného programovacího jazyka.

Drobný zádrhelem mohlo být například, jak zařídit „pro každý z osmi směrů proveďte následující“. Tady se hodí úplně stejný trik, jaký jsme použili v úloze o piškvorkách:[17] každý směr lze popsat dvojicí čísel `dr`, `dc` $\in \{-1, 0, 1\}$, která nám říkají, o kolik se při pohybu v daném směru změní číslo řádku a sloupce. Např. pohyb vlevo je popsán dvojicí $(0, -1)$ a šikmo vpravo nahoru $(-1, 1)$. Podrobněji ve zdrojáku.

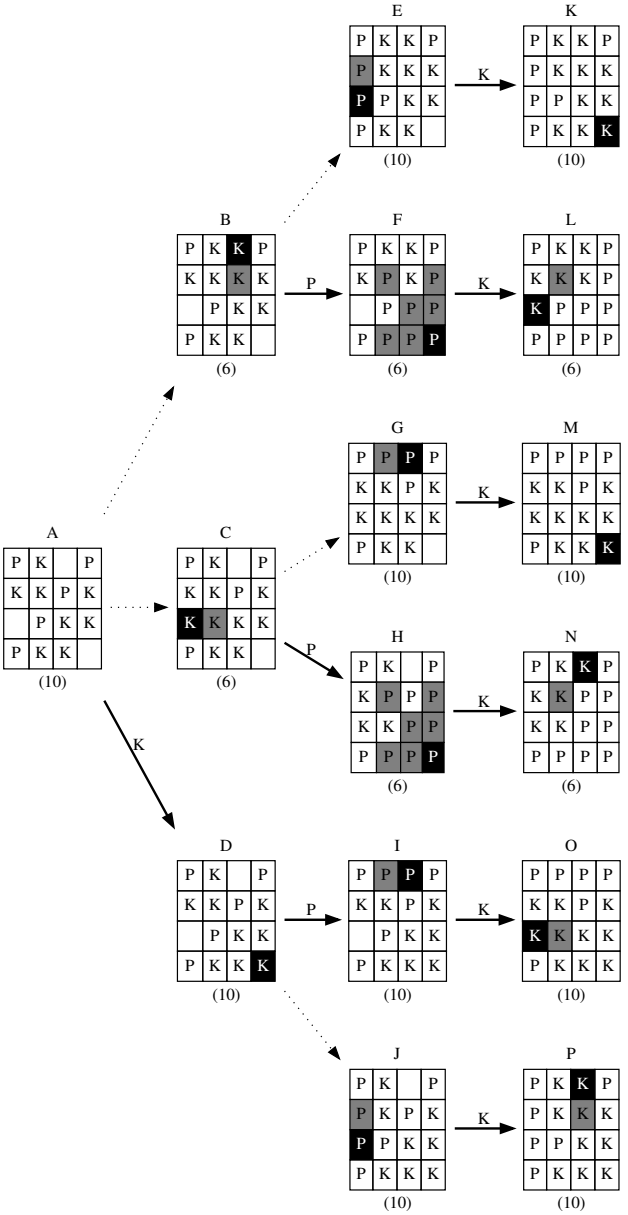
Program (C): `http://ksp.mff.cuni.cz/viz/26-Z4-3.c`

Poznámka: V automatickém vyhodnocování odevzdaných řešení byla chyba: za správnou odpověď jsme nebrali počet kamenů, když oba hrají optimálně, nýbrž nejvyšší vůbec dosažitelný počet kamenů. To odpovídá tomu, že Kevin hraje optimálně a Petr „anti-optimálně“, neboli v druhém tahu vybírá maximum namísto minima. Poté, co nás na tuto skutečnost upozornil jeden řešitel na fóru, upravili jsme vyhodnocování, aby přijímalo obě varianty, a upozornili všechny, kdo se pokoušeli úlohu řešit a řešení jim nebyla uznána.

Pročež si pamatujte: jste-li přesvědčeni, že máte spravně řešení, ale odevzdávátko ho odmítá, upozorněte nás. I organizátor se občas splete…

Filip Štědronský

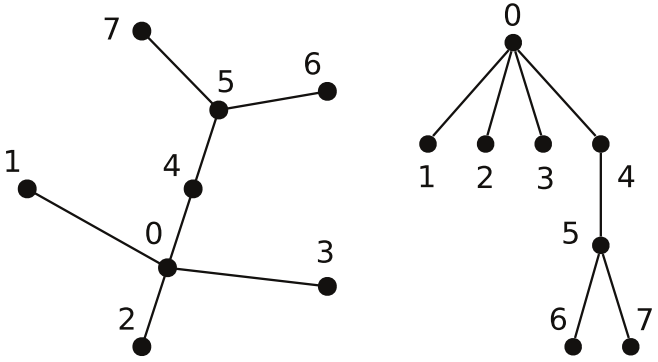
[17] `http://ksp.mff.cuni.cz/viz/26-Z1-2/reseni`


A
(10)
B
(6)
C
(6)
D
(10)
E
(10)
F
(6)
G
(10)
H
(6)
I
(10)
J
(10)
K
(10)
L
(6)
M
(10)
N
(6)
O
(10)
P
(10)
K
P
K
P
K
K
K
K
K
K
K
K

26-Z4-4 Hlídači v labyrintu

Připomeňme si, že v informatické hantýrce se náš Labyrint nazývá *stromem,* křižovatky označujeme jako *vrcholy* a chodby jsou *hrany.*

Univerzálním receptem na tři čtvrtiny stromových úloh je strom si *zakořenit* a úlohu řešit rekurzivně od kořene. Nejsnáze si to ukážeme na obrázku – vlevo je původní strom a vpravo jeho zakořeněná verze.



Jeden libovolný vrchol (v našem případě 0) prohlásíme za *kořen.* Tím nám ve stromu přirozeně vzniknou směry „nahoru“ (ke kořeni) a „dolů“ (od kořene). Z tohoto způsobu kreslení je také vidět, proč se těmto grafům říká stromy (až na to, že informatici mají zvláštní zvyk kreslit je kořenem vzhůru).

Z každého vrcholu u vede právě jedna cesta do kořene (kdyby dvě, tvořily by dohromady cyklus, a ty ve stromech nejsou). Nejbližší vrchol na této cestě (tedy vrchol „těsně nad“ u) nazýváme *otcem* u. Např. vrchol 4 je otcem vrcholu 5. Naopak vrcholy 6 a 7 označujeme jako *syny* vrcholu 5.

Ještě se nám bude hodit pojem *podstromu.* Podstrom pod u (značíme $T(u)$) je tvořen vrcholem u a všemi jeho (i nepřímými) potomky (syny, syny synů, ...). Např. $T(4)$ je tvořen vrcholy 4, 5, 6, 7 (a hranami mezi nimi). Jako *podstromy vrcholu* budeme označovat podstromy pod každým z jeho synů. Např. podstromy vrcholu 0 jsou $T(1), \ldots, T(4)$.

Tím bychom měli z krku část „zakořenit“. Nyní samotné rekurzivní řešení, které obvykle spočívá v tom, že když řešíme úlohu pro nějaký strom T s kořenem u, vyřešíme ji nejprve rekurzivně pro všechny podstromy kořene a pak z těchto dílčích řešení nějak poskládáme řešení pro celé T. Pokud jste o rekurzi nikdy neslyšeli, doporučujeme si přečíst příslušnou kapitolu v naší kuchařce.[18]

Označme si $v_1, \ldots, v_k$ všechny syny vrcholu u a zaveďme zkratku $T_1 := T(v_1), \ldots, T_k := T(v_k)$. Nyní máme dvě možnosti:

[18] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy`

KSP-Z

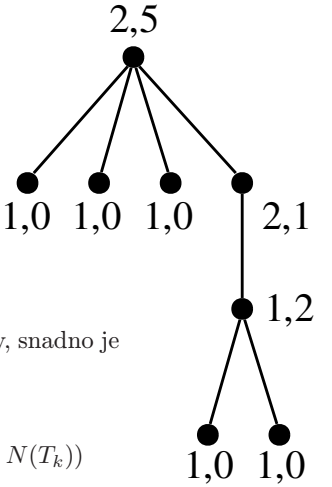
řešení

a) Do u postavíme hlídače. Tak máme postaráno o hrany uv_1 až uv_k a hlídače v každém podstromu T_i rozestavíme dle rekurzivně získaného optimálního řešení pro tento podstrom. To si určitě můžeme dovolit, neb v libovolném rozmístění hlídačů můžeme tu část, která je v T_i, nahradit za optimální rozmístění, a počet hlídačů tím nezvýšíme.

b) Do u nepostavíme hlídače. Pak musíme umístit hlídače do každého v_i. Tedy v rámci libovolného podstromu rozmístíme co nejméně hlídačů tak, aby alespoň jeden stál v jeho kořeni.

Z toho už je vidět, že potřebujeme, aby nám rekurze vrátila nejen optimální počet hlídačů. Výstupem našeho rekurzivního algoritmu spuštěného na strom T budou dvě čísla:

- Nejmenší počet hlídačů, kteří dokáží uhlídat chodby T, za předpokladu, že jeden stojí v kořeni ($K(T)$).
- Nejmenší počet hlídačů za předpokladu, že v kořeni žádný nestojí ($N(T)$).



Pokud tato čísla známe pro všechny podstromy, snadno je spočítáme i pro T:

$$N(T) = K(T_1) + \ldots + K(T_k)$$
$$K(T) = \min(K(T_1), N(T_1)) + \ldots + \min(K(T_k), N(T_k))$$

Ještě musíme vyřešit okrajový případ stromu, který žádné podstromy nemá (je tvořen jediným vrcholem, takovému říkáme *list*). Tam je to ale jednoduché, neb takový strom neobsahuje žádné chodby, a tudíž žádné hlídače nepotřebuje ($N(T) = 0, K(T) = 1$).

Pro ilustraci ukážeme hodnoty K a N pro strom výše:

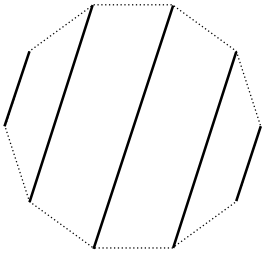
K uhlídání stromu tedy stačí dva hlídači (umístění ve vrcholech 0 a 5).

Program (Python 3): `http://ksp.mff.cuni.cz/viz/26-Z4-4.py`

Filip Štědronský

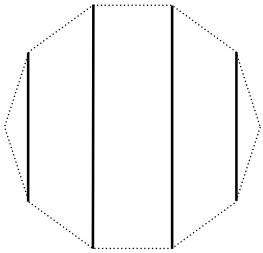
26-Z4-5 Podávání rukou

Zamyslíme se nad úlohou zvlášť pro případy, kdy je N liché a kdy sudé. Nejprve třeba pro N sudé. Pokud si představíme lidi jako vrcholy pravidelného N-úhelníku a podání ruky jako úsečku mezi nimi, je vidět, že aby nikdo v nějakém taktu nezahálel, musí všechny úsečky vést rovnoběžně. Je tedy $N/2$ možností, jak takovým způsobem podávání rukou udělat v různých natočeních.



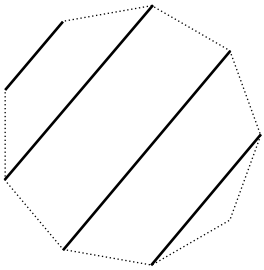
Tím si ale nepodaly ruku všechny dvojice! Třeba se sousedem ob jedno místo si nikdo ruku nepodal. Obecně si nepodala ruku žádná dvojice dvou lidí taková, že je mezi nimi lichý počet lidí (jinými slovy mezi každými dvěma lidmi byl sudý počet lidí, jak je vidět z obrázku). Zato ale platí, že si podala ruku každá dvojice lidí, kteří mají mezi sebou sudý počet lidí (to totiž odpovídá nějakému natočení rovnoběžek).

Abychom doplnili dosud nevyřešené dvojice, navrhneme ještě další schéma podávání – v něm budou vždy dva lidi proti sobě, kteří si s nikým ruku nepodají, a ostatní si budou podávat ruce zase rovnoběžně. Opět máme $N/2$ možností, jak toto schéma natočit, a opět žádné nepropojí jednoho člověka dvakrát se stejným. Teď je akorát potřeba ukázat, že bylo nezbytné, aby v těchto taktech dva lidi zaháleli. Ti, co zaháleli, byli vždy sevřeni svými oběma sousedy, kteří si navzájem podávali ruku. Tím je vždy odříznuti od zbytku světa. Přitom si ale ruce museli podat, proto toto odříznutí bylo nezbytné.



Nikdy si žádná dvojice nepodala ruku dvakrát a každý si podal s někým ruku $(N-1)$-krát (to je přesně tolik, kolikrát měl). Navíc jsme ukázali, že proběhlo jenom tolik zahálení, kolik opravdu muselo, proto pro sudé N může proběhnout jenom N taktů a lépe to nejde.

Teď podobně vyřešíme úlohu pro liché N: Tady bude muset být v úplně každém taktu jeden zahálející. Jeho sousedi si mohou spolu podat ruku, jeho sousedi ob jednoho si mohou podat ruku a tak dále, takže takto si každý podá s někým ruku.



To, kdo zrovna bude zahálet, vždy určí natočení rovnoběžek. Pokaždé bude zahálet někdo jiný, takže rovnoběžky budou natočeny pokaždé jiným směrem. Z toho plyne, že si žádný člověk nepodá ruku s nikým dvakrát, protože každý je od něj jiným směrem. Natočení se vystřídalo celkem N a každý jenom v jednom zahálel, takže si každý musel podat ruku se všemi.

Zahálel každý jenom jednou a podle stejného argumentu, jako jsme použili výše, to lépe nejde.

Martin Španěl

KSP-Z

26-Z4-6 Překreslení obrázku

řešení

Začněme nejprve lehčí variantou. Kevin může obarvit všechny souvislé úseky černé barvy, které mají délku alespoň K. Kratší úseky neobarví nikdy, protože by tím přetáhl na bílá políčka.

Stačí tedy postupně číst vstup a pamatovat si délku aktuálního černého úseku. Pokud jsme na bílé, délka bude nulová. Jakmile nějaký černý úsek skončí, nebo nastane konec vstupu, porovnáme jeho délku s číslem K. Pokud je délka úseku větší, připočteme ji k počtu obarvitelných políček.

Celková časová složitost tohoto řešení je $\mathcal{O}(S)$, lineární s velikostí vstupního obrázku. Paměti spotřebujeme jenom konstantně, $\mathcal{O}(1)$.

Těžší varianta

Nejprve si předpočítáme, na které pozice můžeme štětec umístit, a kde bychom již přetahovali. Uděláme to tak, že pro každé políčko obrázku spočítáme velikost největšího čtverce s pravým dolním rohem v daném políčku.

Obrázek projdeme postupně po řádcích zleva doprava. Pokud jsme na bílém políčku, zapamatujeme si nulu. Pokud jsme na černém, podíváme se o jedno políčko doleva, nahoru a šikmo doleva nahoru. Na nich jsme již hodnotu spočítali dříve. Ze zapamatovaných čísel vezmeme minimum, přičteme k němu jedničku a dostaneme správný výsledek na aktuální pozici.

Například pro obrázek:

```
Č B B Č Č Č
Č Č Č Č Č B
B Č Č Č Č B
B B Č Č Č B
```

Předpočítáme:

```
1 0 0 1 1 1
1 1 1 1 2 0
0 1 2 2 2 0
0 0 1 2 3 0
```

Jak z toho ale zjistíme, kolik políček můžeme obarvit štětcem velikosti $K \times K$? Nejsnazší bude si celý obrázek překreslit a na závěr políčka jen přepočítat. Kreslení však musíme udělat chytře, abychom některá políčka nepřemalovávali mockrát a nepokazili si tím časovou složitost.

Vynulujeme všechna čísla menší než K. Tím nám zůstanou nenulová ta políčka, která můžeme obarvit pravým dolním rohem štětce.

Pro $K = 2$ jsou to pouze tato políčka:

```
0 0 0 0 0 0
0 0 0 0 2 0
0 0 2 2 2 0
0 0 0 2 3 0
```

Nyní obrázek projdeme v přesně opačném pořadí, čili z pravého dolního rohu. Při tomto průchodu všechna čísla postupně „rozšíříme“ doleva nahoru. Pokud je na aktuálním políčku nenulové číslo, tak na políčko vlevo, nahoru a šikmo vlevo nahoru napíšeme číslo o jedna nižší. Při tomto přepisování akorát nikdy nesmíme číslo snížit, vždy je zapíšeme jen pokud tím hodnotu zvýšíme.

Tím jsme celý obrázek překreslili. Stačí už jenom spočítat obarvená – nenulová políčka. Těžší variantu jsme tedy vyřešili v čase a prostoru $\mathcal{O}(RS)$, tedy lineárním s celkovým počtem políček obrázku.

```
0 0 0 1 1 0
0 1 1 1 2 0
0 1 2 2 2 0
0 0 1 2 3 0
```

Program (C): `http://ksp.mff.cuni.cz/viz/26-Z4-6.c`

Jenda Hadrava

Pořadí řešitelů KSP-Z

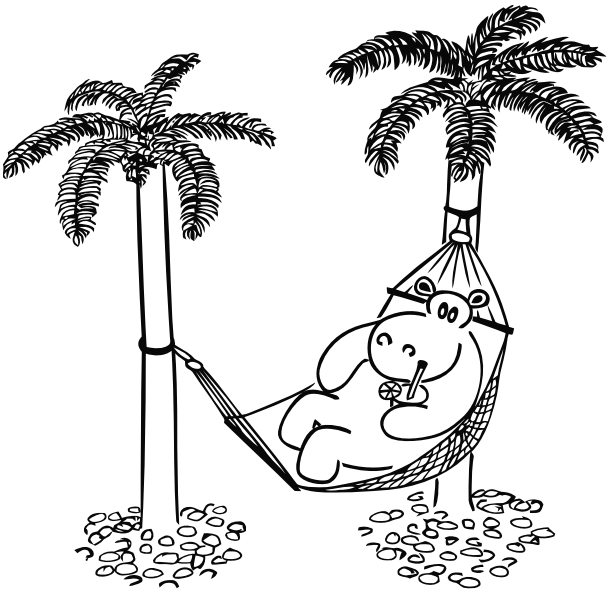
Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
0.				24	264.0
1.	Miroslav Šerý	GValašKlob	1	24	237.5
2.	Jan Tománek	GPelhřimov	3	22	224.0
3.	Jakub Pelc	G_UherBrod	0	22	223.5
4.	Václav Fabík	ZŠKřídloBO	−1	18	190.0
5.	Jonáš Malena	SŠJeštědLI	4	18	164.5
6.	Václav Končický	GSOŠ_FrMís	3	15	140.0
7.	Přemysl Šťastný	GŽamberk	0	17	139.0
8.	Michal Töpfer	G_DrJPekMB	1	17	138.0
9.	Pavel Mikuš	GMělník	3	13	112.0
10.	František Zajíc	G_Nymburk	1	14	102.0
11.	Lucie Studená	GKepleraPH	4	11	95.0
12.	Jakub Lukeš	GNAlejíPH	1	13	88.0
13.	Antonín Teichmann	GJeronýmLI	4	9	72.0
14.	Petr Šíma	GKlatovy	1	9	70.0
15.	Jiří Vozár	G_UherBrod	2	6	59.5
16.	Michal Převrátil	GKlatovy	1	6	58.0
17.–18.	Antonín Bruštík	G_UherBrod	3	8	54.0
	Jan Burda	G_Holice	−1	9	54.0
19.	Marek Vitula	GJarošeBO	3	8	51.0
20.	Jakub Heyduk	SŠP_ČB	4	5	48.0
21.	Jakub Tětek	CírkG Plzeň	0	7	47.0
22.	Václav Trpišovský	GOpenGaBab	−3	6	44.0
23.	Lukáš Fruněk	GLesníZlín	1	5	40.0
24.–26.	Josef Gajdůšek	SŠKKamPard	1	4	39.0
	Stanislav Lukeš	GPísnickáPH	1	5	39.0
	Radovan Švarc	G_ČTřebová	3	5	39.0
27.	Daniel Šerý	G_RožnovPR	2	8	37.0
28.–29.	Jan Horák	GŠumperk	3	5	34.0
	Viktor Kovařík	G_UherBrod	3	5	34.0
30.	Milan Malina	GMikulášPL	1	7	30.0
31.	Jan Václavek	GUnOrl	2	3	28.0
32.	Jan Vargovský	GSPŠFrenšt	4	3	25.0
33.–34.	Ivana Krumlová	GJarošeBO	1	3	23.0
	Vojtěch Václavík	GSOŠ_FrMís	4	3	23.0
35.–36.	Tomáš Michna	SPŠEOstrava	4	4	20.0
	Benedikt Žour	G_UherBrod	−1	5	20.0
37.	Janek Hlavatý	ZŠ_DukelČB	−5	5	19.0
38.–39.	Štěpán Košan	GKlatovy	2	2	18.0

Pořadí řešitelů KSP-Z

	Aneta Šťastná	GOmskPha	4	2	18.0
40.	David Karlík	G_UherBrod	3	2	16.0
41.	Zdeněk Pavlátka	GMikulášPL	2	5	15.5
42.	David Dvořáček	G_UherBrod	3	2	14.0
43.	Martin Jílek	GKlatovy	2	3	13.0
44.–46.	Ivona Hrivová	GŽilina	4	4	11.0
	Dominik Krasula	GKrnov	1	2	11.0
	Jan Vozár	G_UherBrod	0	3	11.0
47.	Michaela Bačová	G_UherBrod	3	2	9.0
48.	Petr Pacner	GBroumov	2	1	7.0
49.	Tereza Bohumská	GPísnickáPH	−1	1	2.0
50.	Martin Majtán	G_SNPPiešť	1	2	1.0

KSP

Hlavní kategorie KSP



Zadání úloh KSP

První série

Prázdnotu vesmíru prořízl oslepující záblesk. Kde před chvílí byl jenom volný prostor, nalézaly se teď megatuny hmoty hvězdné lodě. Z trupu se vysunuly anténní systémy a senzory začaly prozkoumávat okolní prostor.

Mohlo by se to jevit jako standardní skok z hyperprostoru do normálního vesmíru, nebýt dlouhého roztřepeného šrámu táhnoucího se skoro přes celý levobok. Těsně nad šrámem se nalézaly ještě stěží rozeznatelné insignie hlásající, že se jedná o UFC Freya*, těžkou nákladní loď Spojené federace.*

V tom se objevil další záblesk. Některé z hlavních motorů lodě se spustily a začaly do prostoru za lodí chrlit gejzíry světla z nukleární fúze tak jasné, že by se do nich nechráněné lidské oko nemohlo ani podívat. Poškození lodě bylo příliš rozsáhlé, Freya *umírala...*

Bylo skoro zázrakem, že se lodi podařilo přežít cestu hyperprostorem v takovémto stavu. Zdaleka však neměla vyhráno. Hlavní počítač stále zápasil se selhávajícími systémy a soupeřil o kontrolu nad lodí s poškozenými obvody vysílajícími do lodních rozvodů nesmyslné příkazy.

26-1-1 Blokující signály — 8 bodů

Hlavní počítač poškozené hvězdné lodě se pokouší obnovit kontrolu nad většinou důležitých systémů. Bohužel ale poškozené obvody, dříve než je hlavní počítač zvládl izolovat, vyslaly do lodní počítačové sítě několik vadných signálů, které je potřeba zastavit, než se dostanou do svého cíle.

Lodní počítačová síť je představována N propojenými počítačovými uzly, mezi kterými je M přímých spojení (kabel spojující nějaké dva uzly). Síť tedy vlastně představuje neorientovaný graf.

Každý vyslaný vadný signál je zadaný posloupností uzlů sítě (cestou) a svým cílovým uzlem. Každou časovou jednotku se posune o jeden uzel na své cestě dál.

Zastavení signálu probíhá tak, že ve vhodný okamžik vyšleme z hlavního počítače (jeden určený uzel sítě) vhodný blokující signál. Ten cestuje stejnou rychlostí jako vadný signál, ale může jinou cestou. Když se signály potkají (dorazí ve stejný čas do stejného uzlu), tak blokující signál zabrání dalšímu šíření vadného signálu.

Ptáme se, kolik signálů dokážeme zastavit ještě před tím, než dosáhnou svých cílových vrcholů.

Motory postupně ustálily svůj chod a Freya *se stabilizovala. Stav však zůstával kritický, poškozený hlavní reaktor, který byl nyní pod šrámem zčásti odhalený, stále hrozil výbuchem.*

Freya *byla transportní loď postavená ještě za války, měla tedy sice zastaralou, ale snad stále platnou mapu hvězdných soustav. Soustava, do které s vypětím všech sil dolétla, byla sice daleko od běžných letových tras, ale obsahovala planetu schopnou udržet lidský život.*

Pátrací senzory malou planetu konečně našly, loď mírně změnila svůj kurz a začala se připravovat na nouzové přistání. Původně měla celý svůj život zůstat v mezihvězdné prázdnotě a k planetě se přiblížit nejblíže na dosah raketoplánu, ale poctiví programátoři a konstruktéři ji před lety připravili i na tuto eventualitu. Devatenáct lidí v kryogenickém spánku stále nic netušilo...

26-1-2 Přeskládání nákladu — 9 bodů

Hvězdná loď potřebuje připravit svůj náklad na nouzové přistání. Přípravy spočívají mimo jiné v tom, že se musí rozdělit, který náklad bude ve skladišti na pravoboku a který na levoboku.

Náklad je tvořen celkem N kontejnery. Z důvodu bezpečnosti a rovnoměrného rozložení zásob (aby jich bylo dost i v případě ztráty jednoho skladiště) existuje také M doporučení. Každé doporučení říká, že nějaká dvojice kontejnerů by neměla být v tom stejném skladišti.

Všechny předpisy najednou pravděpodobně splnit nepůjde, ale hlavní počítač chce nalézt rozdělení kontejnerů do skladišť (množství kontejnerů v jednotlivých skladištích nemusí být stejné) takové, aby byla splněna alespoň polovina doporučení.

Dopravníky uvnitř lodi dokončily přesuny kontejnerů, loď přečerpala zásoby paliva tak, aby kompenzovala vyvážení, a byly uzavřeny všechny vzduchotěsné dveře.

Hlavní motory už nějakou chvíli nepracovaly, jejich další chvíle měla přijít v konečné fázi sestupu. Freya *pomalu klesala do atmosféry planety. Jak se začala třít o horní vrstvy atmosféry, tak její příď postupně změnila barvu přes temně rudou až do jasné oranžové. Okolo trupu začaly šlehat plameny a trupové nástavby začaly odletovat jedna za druhou. Během chvíle zmizely pátrací radary, jeřábové manipulátory i zbytky poškozených komunikačních antén.*

V přesně vypočtenou chvíli naběhly přední manévrovací motory. Nebyly stavěné jako brzdící, ale jejich předimenzovaná velikost a spuštění na kritický výkon by mohly stačit, pokud ten nápor vydrží.

Loď zpomalovala. V tom ale její pravobok pohltil oslňující výbuch. Zdejší vrchní vrstva atmosféry obsahovala nějaké kapsy výbušných plynů. Tentokrát to odneslo jen skladiště na pravoboku, ale další takový výbuch by mohl loď rozpůlit.

Hlavní počítač vysunul jeden z posledních fungujících radarů. Ve zlomku sekundy, než ho proud vzduchu vyrval z trupu, stihl zaznamenat rozložení kapes plynů v atmosféře.

26-1-3 Plynové kapsy 8 bodů

Snímkovací radar dodal průřez atmosférou obsahující dva druhy plynů. Průřez je znázorněn v podobě posloupnosti znaků a a b (dva druhy plynů) délky N.

Pro loď širokou dva znaky je bezpečné proletět skrz souvislou oblast buď jednoho, nebo druhého plynu. Na rozhraní plynových kapes je to moc nebezpečné (tedy může proletět oblastí *aa*, *bb*, ale nemůže proletět místem, kde se setkávají – *ab* nebo *ba*).

KSP

zadání

Hlavní počítač potřebuje vytvořit datovou strukturu, které by se mohl rychle ptát, kolik míst k průletu je v zadaném intervalu. Tedy kolik v něm je stejných sousedních polí.

Počítejte s tím, že dotazů bude řádově N a že počítáme i ta místa, která se vzájemně překrývají.

Příklad: Výstup z radaru a odpovědi na dotazy (indexujeme od nuly):

```
Radar:  aababbaaa
[0,7]   -> 3 místa
[0,8]   -> 4 místa
[2,4]   -> 0 míst
```

Po dalším výbuchu vzplál jeden z brzdících motorů, a proto ho nouzový systém i s okolní sekcí odhodil. Explodoval v obrovskou kouli ohně těsně za lodí. To se však už Freya *dostala do spodních vrstev atmosféry.*

Kdyby se hlavní počítač mohl divit, asi by se teď divil. Ale nic takového neměl ve svém programu, a tak jen provedl rychlý výpočet, aby se nějak vypořádal s údaji o mnohem vyšší rychlosti, než byla původně plánovaná.

Hlavní motory podruhé naběhly, tentokrát s tryskami obrácenými na reverzní tah. Počítač vyřadil všechny pojistky a spustil je na výkon, na který ještě nikdy neběžely. Za lodí zůstával pruh doslova spálené atmosféry.

Loď brzdila s takovým přetížením, že začaly povolovat vnitřní přepážky. Jedno z kryogenních oddělení, společně s celým levobočním hangárem, vylétlo z lodi a zaniklo v záři atomového ohně motorů. Odletovaly kusy trupu a konstrukce se bortila.

Byla to dobře postavená loď, ještě podle válečných specifikací. Dnešní lodě by se už dávno rozpadly, Freya *však držela dál. Pak ale přišla i její chvíle. Nejdříve se v půlce trupu zkroutila a zanedlouho se celá zadní polovina třísetmetrové lodě utrhla společně se všemi hlavními motory.*

Svou práci však hlavní motory vykonaly, zbrzdily sestup. Zatím stále funkční přední část pokračovala v řízeném pádu korigovaném manévrovacími motory. Pak dopadla. Odrazila se a dopadla podruhé. Vyryla v místním ekvivalentu deštného pralesa brázdu dlouhou skoro tři kilometry a pak se konečně zastavila...

⋆⋆⋆

Jacob otevřel oči. Nad ním se nacházel otevřený poklop jeho kryogenní kóje. Otřepal z rukou poslední zbytky kryogenního gelu a protřel si oči.

„Tak počkat, tady něco nehraje!“ pronesl pomalu při pohledu na rozbitou místnost, do níž odněkud prosvítalo podivné žluté světlo. Vyhoupl se z kóje a přistál bosýma nohama na nakloněné podlaze.

Druhá strana místnosti, na které se nacházely zbylé kóje, byla celá zavalená. Pomocí kusu nosníku se mu povedlo roztáhnout dveře a skrz trosky se prodral na chodbu a k nejbližšímu počítačovému uzlu, aby zjistil, co se stalo.

26-1-4 Oprava databáze — 10 bodů

Databáze hlavního počítače je silně poškozena a samoopravný mechanismus je vyřazen. Musíme ji proto opravit ručně. Jak ale poznat, že jsme ji sestavili správně? Jistá naděje tu je: víme, že databáze měla podobu posloupnosti celých čísel, a počítač si pamatuje, kolik v ní bylo *trojných prvků.*

Trojný říkáme takovému prvku, který se dá poskládat jako součet nějakých tří předchozích prvků v posloupnosti. (Dodejme ještě, že jeden prvek nemůžeme v součtu použít vícekrát, ale dva prvky téže hodnoty už ano.)

Příklad: Pro posloupnost $1, 4, 7, 2, 7, 16, 10$ jsou trojnými prvky druhá sedmička ($7 = 1 + 4 + 2$), šestnáctka ($16 = 7 + 2 + 7$) a desítka ($10 = 1 + 2 + 7$), jiné trojné nejsou. První sedmička není na rozdíl od druhé trojná, protože u ní ještě nemůžeme použít číslo 2.

⇧ **Lehčí varianta (za 6 bodů):** Vyřešte úlohu pro případ, kdy hledáme dvojné prvky namísto trojných (tedy skládáme jen ze dvou předchozích prvků v posloupnosti).

Když Jacob zjistil, co se stalo, chvíli stál naprosto bez pohnutí. Potom udeřil pěstí do přepážky.

„Zatraceně, tak jsem jediný přeživší na téhle planetě, která dokonce ani nemá jméno!“

Když se trochu vzpamatoval, začal postupovat směrem k bývalému můstku. Cestou se zastavil ve výstrojním skladu a z hromad popadaných beden vylovil nějaké základní věci jako baterku, nůž, lékárničku a nějaký batoh. Taky se převlékl do odolnější kombinézy a vzal si boty.

Po dalších několika minutách zjistil, že můstek lodě již neexistuje. Místo toho se mu však naskytl pohled na okolní prales. Stromy nevypadaly zase tolik jinak než ty pozemské. Měly trochu jinou barvu a byly hlavně mnohem vyšší. Převyšovaly o pár metrů dokonce i zabořené torzo lodě.

Věnoval ještě asi hodinu průzkumu, během něhož obešel většinu lodě. Nakonec vylezl po již dávno vychladlém trupu na nejvyšší místo a posadil se pozoruje dlouhou brázdu od pádu lodě.

„Prales... To znamená, že tu asi bude nějaký mimozemský život. A nemusel by být úplně přátelský," zamyslel se Jacob nahlas. Už se také stmívalo a z lesa se začínaly ozývat podivné zvuky. Chtělo by to asi okolí nějak zajistit. Shodou okolností Freya *zrovna převážela senzorové soupravy, které by mohl použít.*

Vypravil se tedy do trosek skladu a rychle vytáhl několik funkčních senzorových věží a pár desítek metrů kabelů.

26-1-5 Senzory — 10 bodů

KSP

zadání

Chceme rozestavit K senzorových věží na čtvercovou síť o rozměrech $M \times N$. Pro správnou funkci senzorů je potřeba, aby žádné dvě senzorové věže nestály ve stejném sloupci nebo na stejném řádku.

Každá senzorová věž má navíc určený obdélník, ve kterém musí být postavena. Najděte pro dané zadání nějaké fungující rozestavení věží nebo určete, že takové rozestavení neexistuje.

⇧ **Lehčí varianta (za 6 bodů):** Vyřešte úlohu pro případ, že sít má tvar jednořádkové „nudle" (tedy $M = 1$) a dvě věže nesmí stát v témže sloupci.

Po zapojení poslední věže do systému se Jacob uložil ke spánku v bývalé jídelně.

Druhý den na planetě již probíhal trošku poklidněji. Po dalším průzkumu lodě zjistil, že má celkem dostatečné zásoby pitné vody a že nouzové palivové články mu budou schopny dodávat energii ještě přinejmenším rok, pokud se mu do té doby nepovede znovu nahodit záložní reaktor.

Mnohem větší problém byl ale s jídlem. Hlavní sklad jídla se totiž nacházel v zadní části lodě a vepředu byla jen hydroponická zahrada, která navíc při přistání dost utrpěla. Jelikož nechtěl zatím zkoušet ochutnávat místní jídlo, rozhodl se Jacob hydroponiku opravit.

26-1-6 Hydroponie — 11 bodů

Hydroponická zahrada je tvořena soustavou N očíslovaných přihrádek na rostliny, v každé může být maximálně jedna rostlina (ale také tam nemusí být žádná). Mimo to je zde M napájecích okruhů, každý napojený na určitý interval přihrádek.

V každém napájecím okruhu chceme mít umístěnou dohromady minimálně jednu rostlinu, jinak nám na rozmístění a počtu rostlin nezáleží.

Ptáme se na počet možných způsobů, jak můžeme obsadit přihrádky rostlinami tak, aby byly splněny výše zmíněné podmínky. Jelikož počet může být obrovský, spočítejte ho modulo $1\,000\,000\,007$.

Jacob zrovna přemístil poslední rostlinku, když v tom se ozval z chodby poplach. To počítač napojený na senzory nahlásil, že něco velkého porušilo perimetr. Jacob popadl velký nůž a vyběhl ven.

Doběhl na místo, odkud senzory hlásily narušitele. Na zemi tam byly v bahně obtisknuté nějaké stopy a... V Jacobovi hrklo, vedle stop ležel umně vyrobený a nazdobený malý oštěp. To znamená, že je tady inteligentní život! To musí prozkoumat!

Natěšený doběhl nazpět do torza lodi, během pěti minut si pobral zásoby jídla a vody na několik dní, sbalil si lékárničku, kameru a další potřebné věci a vydal se opět na místo, kde nalezl ten oštěp.

Vydal se opatrně po stopách směrem do lesa. Po několika metrech přišel na malý palouk, kde ho zaujal místní podivný hmyz. Podobal se trochu pozemským mravencům, jen byl mnohem větší. Zvedl kus klacku s několika těmito tvory a chvíli je pozoroval.

26-1-7 Mravenci — 8 bodů

Tvorové podobní mravencům lezou po rovném kusu klacku dlouhém D centimetrů. Na začátku se jich zde nachází N a každý tvor se pohybuje buď doprava, nebo doleva, a to rychlostí jednoho centimetru za sekundu.

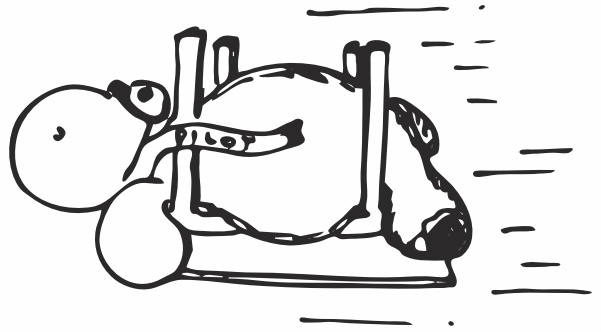
Když tvor přijde na konec klacku, tak spadne. Když se dva tvorové potkají, tak se oba otočí čelem vzad. Nás zajímají dvě věci:

a) (za 3 body) Za kolik sekund z klacku spadne poslední tvor?

b) (za 5 bodů) Na jaké pozici tento tvor na začátku stál?

Tvory v této úloze považujte za zanedbatelně malé vzhledem k velikosti klacku.

Od pozorování tvorů na klacku náhle Jacoba vyrušilo zapraskání za jeho zády. Rychle se otočil a...



Druhá série

Mezi listy zlatě zasvítily dva velké kruhy. Byly to oči nějakého zvířete. K očím patřil i čumák a překvapivě malá hlava. A tělo a čtyři dlouhé nohy. Jacobovi vyskočilo srdce někam do krku.

Nedělal si iluze o tom, že zvíře přišlo, aby se stalo jeho průvodcem po neznámé planetě. Když jako jediný přežijete nouzové přistání vesmírné lodi, je to samo o sobě dost šílené a na další zázrak si obvykle musíte zase chvíli počkat.

Zvíře zatím Jacoba pozorovalo. Možná pozorovalo i kus jeho okolí. Rozhodně se zdálo, že pozorně naslouchá. Aspoň pokud to, co Jacob považoval za uši, skutečně byly uši. Něco tu nesedělo, ale strach nedovolil jeho mozku zabývat se nějakými nepodstatnými nesrovnalostmi. Místo toho se dal na útěk.

Běžel, co mu síly stačily, a litoval, že se nedá vylézt na okolní stromy. Netušil, jestli zvíře zůstalo stát na místě, nebo jestli běží za ním a on ho jen pro svůj vlastní dusot neslyší. Netroufal si ohlédnout se, a navíc, dokud zvíře neviděl, mohl doufat, že tam není.

Najednou vzduch prořízlo nějaké zasvištění. Ani tehdy se Jacob neohlédl, jen si uvědomoval, že ho nic neskolilo, že může běžet dál.

Zastavil se až za hodnou dobu, kdy začal opadávat jeho strach, a s tím mu začaly docházet síly. Našel odvahu ohlédnout se. Po zvířeti nebylo ani vidu. Jacob si nebyl úplně jistý, že trefí zpět, ale pro tuhle chvíli se rozhodl doufat, že cestu najde.

Začal se zase soustředit na svůj původní plán, totiž hledat místní inteligentní život. Pozorně se rozhlédl po okolí. A zděsil se.

Teprve teď si všiml, že kus od něj jsou volně natažené dva provazy, které se do sebe navíc poněkud zamotaly. Byl div, že se o ně při svém úprku nepřerazil. Jacob se k nim sehnul, aby je mohl prozkoumat blíž.

26-2-1 Zamotané provazy — 7 bodů

V pralese jsou volně natažené dva provazy, které se do sebe zamotaly. Provazy jsou pevně uvázané na dva stromy, první provaz níž a druhý výš. Žádný z nich se nikde nevrací. Může to tedy vypadat třeba takto:

O každém překřížení víme, který z provazů je vpředu. Chceme zjistit, jestli je možné provazy rozmotat bez toho, aby bylo nutné některý z nich odvazovat.

Program dostane na vstupu číslo N udávající počet křížení a N čísel 1 nebo 2 udávajících, který provaz je vpředu (na stromech je provaz 2 vždy uvázaný nad

provazem 1). Odpovědět má `ano`, nebo `ne`, podle toho, zda je provazy možné rozmotat.

Příklad (odpovídá obrázku):

```
4
1 2 2 1
```

Odpovědí je `ano`, jelikož provazy rozmotat jdou.

Při zkoumání provazů Jacob najednou zahlédl mezi stromy jakýsi barevný záblesk. Pouhým pohledem se mu nedařilo zjistit víc, a tak se s patřičnou obezřetností vydal tím směrem. Jak se přibližoval, měnily se občasné záblesky v barevnou plochu prosvítající mezi stromy.

Konečně se Jacob dostal až k ní. Ocitl se na něčem, co snad mohlo připomínat mýtinku, ovšem po stromech ani trávě tu nebyly vůbec žádné stopy. Teď už Jacob viděl, že trojúhelníkovou barevnou plochu tvoří zvláštní hexagonální kameny, některé červené, některé do žluta.

Zdálo se, že je kdosi naskládal do mnoha vrstev, kterými pečlivě vyplnil hlubokou jámu; několik dalších kamenů se válelo v nejbližším okolí. V jednom z nich byl vyrytý podivný nápis:

Xeŕa trŕineźr ĝaetȯ všpogiń
Xeŕa giĝesȯr ĝaetȯ mànǰine
Íhgišpoƿ kiĝes trǔćeƿ náƿ
matrǰineƿ vàs heriŕeƿ sdi náƿ

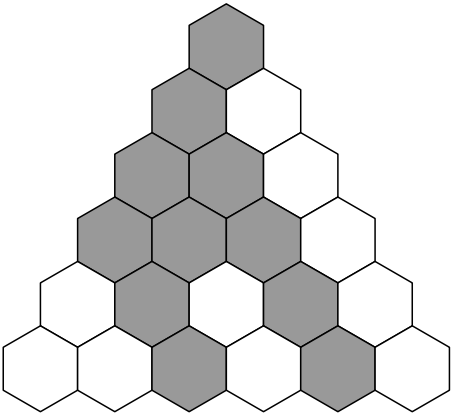
Jacob si všiml, že jedna z barev spojuje všechny tři strany trojúhelníku, a napadlo ho, jestlipak je to tak i ve vrstvách, které nevidí.

26-2-2 Barevný trojúhelník — 8 bodů

Máme rovnostranný trojúhelník o hraně N tvořený hexagonálními kameny dvou barev.

Dokažte, že v trojúhelníku existuje jednobarevná souvislá plocha, ve které se nachází alespoň jeden kámen z každé strany trojúhelníku. Kameny ve vrcholech patří do dvou stran zároveň.

Příklad takového trojúhelníku si můžete prohlédnout na obrázku. Barvou, která spojuje všechny tři strany, je zde šedá.



Protože se začalo připozdívat, rozhodl se Jacob brzy pokračovat dál. Kamenný trojúhelník mu nyní sloužil jako dobrý orientační bod, a on se vypravil směrem dál od jednoho z jeho vrcholů.

Po nějaké době došel k místu, kde se prales najednou začal rozestupovat. Vzniklý výhled odhalil krom jiného to, že tato část pralesa je mírně vyvýšená nad okolím – zdola musel být nahoru opravdu impozantní pohled.

O kus dál uviděl Jacob řeku. To ho potěšilo, neboť pokud se tu dá spoléhat na podobné věci jako na Zemi, mohl by ji sledovat a dojít po jejím proudu k nějaké civilizaci.

Zdálo se ovšem, že terén je všelijaký a některá místa by mohla cestu hodně zpomalit. Musel si ji proto dobře rozmyslet.

26-2-3 Plánování cesty — 10 bodů

Jacob se chce co nejrychleji dostat k řece. Terén, přes který musí projít, si lze představit jako čtvercovou síť o rozměrech $R \times S$. V některých oblastech je tento terén dost nepříjemný, a tak přes některá políčka trvá průchod delší dobu, některá jsou dokonce zcela neprůchozí.

Průchod přes oblast ležící na souřadnicích i, j trvá $t_{i,j}$. Přecházet mezi políčky je možné pouze vodorovně nebo svisle (šikmo ne).

Vaším úkolem je najít nejrychlejší cestu.

Jacob cestu naplánoval, jak nejlépe uměl. Dál byl ovšem rozhodnutý vyrazit až ráno. Přeci jen není nejrozumnější pohybovat se unavený někde, odkud můžete spadnout. Teď si raději opatřil provizorní přístřešek. Chvíli se ještě podivoval nad tím, jak jasná je noc, a pak se uložil ke spánku.

⋆⋆⋆

Ráno se Jacob vzbudil brzy. Pobalil si svých několik věcí a odhodlaně vyrazil na cestu. Terén tu byl skutečně všelijaký, místy příjemně ušlapaná hlína, místy skoro bažiny, mezitím ledacos jiného, ovšem připravená trasa se vcelku osvědčila. Mohlo být krátce po poledni, když Jacob dorazil k řece.

V jednom kameni u břehu byl opět vyrytý podivný nápis:

Íhgišpop̌ ki̇ĝes trǔćep̌ náp̌
lugi íhgišpop̌ ki̇ĝes ńeríp̌ athĝep̌ sdi̇ náp̌
Reho ́ćer̀ ná sjo ĝesŕit
ASN sjo tŕjine

Ještě víc ovšem Jacoba zaujaly podivné kostky na protějším břehu. Vypadaly trochu jako dětské dřevěné kostičky, jen o mnoho větší. A zdálo se, že z nich někdo chce podobně jako z kostiček pro děti postavit velkou věž.

26-2-4 Stavba věže — 10 bodů

Někdo se z K kostek snaží postavit věž. Všechny kostky jsou stejně velké, ale každá kostka má svoji váhu w_i a nosnost ℓ_i. Nosnost ℓ_i znamená, že i-tá kostka unese na ní stojící kostky o maximální celkové váze ℓ_i, jinak se zbortí a věž spadne.

Zajímá nás, jestli lze postavit věž ze všech K kostek. Věž stavíme tak, že přímo na sebe pokládáme jednotlivé kostky (můžete si představit i to, že stavíme komín).

⇧ **Lehčí varianta (za 3 body):** Vyřešte úlohu za předpokladu, že všechny kostky mají váhu $w = 1$.

Stavba, ať už měla být v budoucnosti čímkoli, byla zrovna opuštěná. Jacob se tedy vydal dál po proudu řeky. Cestou občas potkal nějaký ten nápis na kameni, stále se mu v nich ale nedařilo odhalit žádný smysl.

Po několika nápisech a mnoha krocích se před Jacobem najednou objevila malá věžička. Nebo alespoň malá na místní poměry. Jak už Jacob očekával, stál před ní kámen s nápisem:

Ĝesŕit íhgišpop̌ ki̇ĝes ńeríp̌ athĝep̌ sdi̇ náp̌
ASN krees

Věžička neměla okna, měla ovšem dveře. Po troše váhání si Jacob dodal odvahy a zkusil dveře otevřít. Šlo to překvapivě lehce. Jako první zaznamenal směs různého hučení a bzučení. Pak teprve s jistou úlevou zjistil, že vevnitř nikdo není.

Původcem podivných zvuků se ukázaly být jakési krabičky. Byly na sobě různě zavěšené, celá konstrukce byla navíc uchycená ve věžičce. Jacob se ale nemohl ubránit dojmu, že konstrukce už nemůže dlouho vydržet. Ačkoli si nebyl jistý, jestli je to dobrý nápad, rozhodl se jí pomoci tím, že ji vyváží.

26-2-5 Vyvažování 12 bodů

Očíslované krabičky jsou zavěšené ve věžičce, hrozí ale, že celá konstrukce spadne. Chceme ji proto vyvážit. Podle piktogramů na zdech víme, že je potřeba zachovat určité pořadí krabiček na konstrukci. Celou úlohu si tak můžeme představit jako vyvažování binárního vyhledávacího stromu.

Na vstupu máme obecný binární vyhledávací strom (jeho definici naleznete v kuchařce) a chceme z něj udělat dokonale vyvážený binární vyhledávací strom. Pro každý vrchol výsledného stromu tedy musí platit, že rozdíl velikostí jeho levého a pravého podstromu je nejvýše 1.

Konstrukce už drží jen silou vůle, takže požadujeme, aby váš algoritmus pracoval s lineární časovou složitostí. Počet bodů, které dostanete, bude záviset na prostorové (paměťové) složitosti vašeho řešení. Nějaké body dostanete i za lineární, na plný počet ale potřebujete konstantní složitost.

Po práci si Jacob udělal pauzu na svačinu. Bylo pomalu vhodné začít myslet na návrat, ale on chtěl ještě pokračovat ve svém pátrání. Usoudil, že alespoň do večera může jít dál, a pak se uvidí.

S plnějším žaludkem se pak Jacob vypravil opět po proudu řeky. Tentokrát ovšem potkal něco zajímavého mnohem dřív. Došel totiž k něčemu, co se snad dalo považovat za brod.

Tradičně tu potkal vyrytý nápis, tentokrát dost složitý. Po chvilce jeho zkoumání Jacob pojal podezření, že znaky, které už nějakou dobu potkával na kamenech u břehu, jsou zkratkou nějakého názvu. Studoval další slova na kameni a přemýšlel, která z nich by také mohla být názvy a mít nějakou svoji zkratku.

26-2-6 Zkratky míst 12 bodů

Máme podezření, že pro názvy míst se používají třípísmenné zkratky, a máme také seznam slov, o kterých si myslíme, že by mohly být názvem nějakého místa.

Zkratka má první písmeno vždy stejné, jako je první písmeno názvu daného místa, a zbylá dvě písmena jsou libovolná dvě písmena z názvu místa ve správném pořadí. Název *Praha* tak lze zkrátit např. na *PRH* nebo *PHA*, ale už ne na *PHR* nebo *RHA*.

Vaším úkolem je zjistit, pro která všechna slova bychom skutečně uměli takovou zkratku najít. Zkratky míst musí být navzájem jedinečné, chceme jich ale najít co nejvíce. Je-li možných řešení se stejným počtem nalezených zkratek víc, vypište libovolné z nich.

Příklad:

Vstup:	Výstup:	
Pra	Pra	PRA
Praga	Praga	PAA
Prak	Prak	PAK
Prk	Prk	PRK
Prkg	Prkg	PRG
Pak	Prague	PGU
Prague		

Jacobovi se povedlo překonat řeku a dostat se na druhý břeh. Před ním začínalo něco, co silně připomínalo cestičku. Zhluboka se nadechl a pak po ní vyrazil.

Cestička chvíli vedla skoro podél řeky, pak se od ní ovšem prudce odklonila a zavedla Jacoba za hradbu stromů. Nebyly to stejné stromy jako v pralese, ale podobně jako ony byly vysokánské a měly o trochu jinou barvu, než by člověk čekal na Zemi.

Jacob se rozhlédl kolem sebe a bezděky zatajil dech. Našel, co hledal! Byl tu inteligentní život, byl tady, přímo před ním.

Tvorové byli vlastně podobní lidem. Byli vyšší, Jacob by vedle nich vypadal jako malé dítě. Podobně jako zvíře minulý den měli překvapivě malou hlavu a naopak nezvykle velké oči. Barvou kůže připomínali spíše pozemšťany, kterým zrovna není dobře od žaludku. Ale bez nejmenších pochyb to byli představitelé místního inteligentního života.

Skupinka tvorů se zrovna motala kolem obrovského kmene stromu, který zřejmě pocházel z pralesa.

26-2-7 Čištění kmene — 13 bodů

Humanoidní tvorové čistí kmen stromu. Celkem se práce účastní T tvorů. Na kmeni je M míst, která je potřeba očistit. Každé takové místo je na nějaké pozici m_i (to může být třeba vzdálenost od konkrétního konce kmene; pozice bude vždy celočíselná).

Na začátku stojí j-tý tvor na pozici t_j. Práce tvorů probíhá v jednotlivých krocích: v každém kroku se každý tvor (nezávisle na ostatních) může přesunout o jednu pozici vlevo, vpravo, nebo může zůstat stát na místě.

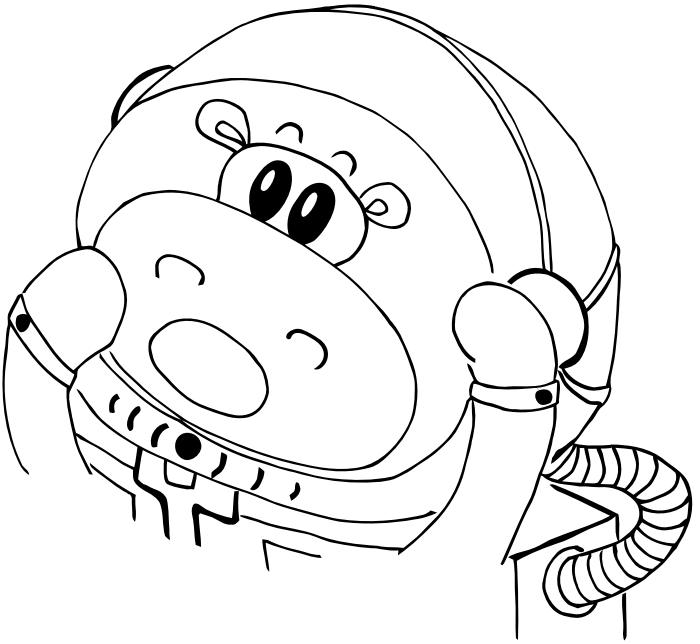
Očištění části kmene, nad kterou tvor stojí, je bleskové (proběhne ve stejném kroku, kdy tvor k této části kmene dojde). Speciální výjimkou je počáteční pozice každého tvora. Ta je očištěná ještě před tím, než tvor udělá první krok.

Zajímá nám nejmenší počet kroků potřebný k očištění celého kmene. Při řešení můžete předpokládat $1 \leq M, T \leq 100\,000$ a $1 \leq m_i, t_j \leq 1\,000\,000\,000$.

Příklad: Mají-li se zkontrolovat místa $2, 5, 6$ a tvorové stojí na pozicích $3, 5$, zvládnou kmen očistit na 1 krok. Kdyby stáli na $3, 4$, budou kroky potřebovat 2.

Když se před ním tvorové vynořili, nebyl si Jacob najednou vůbec jistý, co by vlastně měl udělat. Pozoroval skupinku, všiml si také kdesi za nimi ďolíků v zemi, všiml si, že v jednom z nich zmizel stejný tvor.

Najednou se ozval výkřik. Jacob se polekaně rozhlédl. Někdo si ho zřejmě všiml a upozornil na něj ostatní. V té chvíli se na něj upřely pohledy všech tvorů.



Třetí série

Dva mimozemšťané vydali zvláštní výskot, při kterém Jacobovi ztuhla krev v žilách. Rozhodl se nic neriskovat, prudce se otočil a přešel v trysk. Terén se zde prudce svažoval. Ozvaly se další výskoty. Zněly tak hrozivě, že Jacob na chvíli přestal dávat pozor na své kroky a nohy se mu zamotaly do lián.

Udržet balanc se mu nepodařilo a pak už šlo vše velmi rychle. Jacob ještě stihl natáhnout ruce před sebe, z pádu se tak stal kotrmelec. První kotrmelec byl vystřídán druhým, mnohem rychlejším kotrmelcem. Nežli získal příležitost jakkoliv ovlivnit dráhu svého pohybu, už se řítil dolů ze zhruba desetimetrového srázu.

Po probuzení byl Jacob oslněn červeno-žlutými světýlky, odvrátil proto svůj pohled pryč. Viděl, že se nachází v místnosti se zelenými stěnami. V místnosti se mimo té, na níž ležel, nacházely další čtyři postele. Jednalo se vlastně spíše o lenošky nežli klasické postele, na jaké byl Jacob zvyklý. Jakmile Jacob pořádně zaostřil svůj zrak, zpozoroval, že neleží v klasické místnosti, nýbrž ve velmi honosném stanu.

Vrátil se pohledem ke stropu, teď už byl schopen si prohlédnout ona světýlka. Nebyla to světýlka, ale ozdobné drahokamy odrážející záři svící rozmístěných všude po stanu. Kromě svícnů se zde nacházely zdobené zlaté oštěpy, zlaté masky a totemy. Stan působil jako sídlo šamana.

26-3-1 Výklad z drahokamů — 11 bodů

Žluté a červené drahokamy se těšily velké oblibě a mimo ozdobných účelů se využívaly i k vykládání osudu.

Při takovém výkladu se drahokamy rozsypou na zem a uspořádají do obdélníkového tvaru. Protože všechny drahokamy mají zhruba stejné rozměry, vytvoří tak čtvercovou síť.

Následně se zkoumají všemožné pravidelnosti ve vzniklém obrazci. Zvláštní význam mají uspořádání, ve kterých se drahokamy střídají jako políčka na šachovnici. Vaším úkolem je proto najít největší takovéto uspořádání.

Nyní o něco formálněji: Je zadána tabulka znaků `.` a `#` o R řádcích a S sloupcích. Nalezněte největší čtvercovou oblast, ve které jsou znaky uspořádány jako na šachovnici. Tedy v této čtvercové oblasti nikdy nesousedí dva stejné znaky celou stranou, ale pouze rohem.

Příklad: Podívejme se na následující tabulku o šesti řádcích a sedmi sloupcích:

```
.#....#
#.##.#.
.#.#.##
#.#.#.#
...#.#.
#.#.#.#
```

Největší hledaná čtvercová oblast s šachovnicovým vzorem má stranu dlouhou čtyři a svůj levý horní roh má na třetím řádku a v třetím sloupci.

Teprve po prohlédnutí stanu si Jacob uvědomil, co se dělo před jeho probuzením. Radost z neporaněné hlavy a rukou vyhasla v momentu, kdy zjistil, že nohama nemůže nejen pohybovat, ale dokonce je ani necítí. Nezbývalo než zůstat ležet a zaposlouchat se do venkovních zvuků.

Jacob věděl pouze to, že se nachází v mimozemském táboře a stará se zde o něj trojice léčitelů. Dobře jej živili a Jacob sílil. Pomocí posunků se s nimi i poměrně dobře dorozuměl.

Po zhruba měsíci donesli léčitelé Jacobovi berle. Vyrobeny byly z precizně leštěného dřeva. Dřevo připomínalo pozemský mahagon, jen bylo mechově zelené. Rukojeti byly omotány kožešinou a z boku berlí se nacházela spousta rytin vyobrazujících souboje mimozemšťanů s divou zvěří.

Pohyb s berlemi byl obtížný, protože Jacob nohy za sebou pouze vláčel. I tak měl ohromnou radost, když se mohl začít procházet mimozemským táborem. Při jedné z prvních procházek natrefil na stan zdejších švadlen.

26-3-2 Střih látky — 12 bodů

Jacob vypomáhá švadlenám v jejich práci. Jeho prvním úkolem je stříhání látky. Pozemský pomocník je však ukrutně nešikovný a švadlenám je pro smích.

Jacobovi se daří vést střih rovně, má však velmi špatný odhad. Látku prostě nikdy neustřihne tak, jak by si představoval. Pomozte mu!

Vašemu algoritmu bude předložen konvexní mnohoúhelník zadaný posloupností vrcholů (vrcholy budou zadány v pořadí, v jakém se vyskytují na obvodu mnohoúhelníka). Jeho úkolem pak je vybudovat si datovou strukturu, se kterou bude schopen efektivně odpovídat na střihové dotazy.

V rámci jednoho střihového dotazu bude zadána polopřímka, podél které bude veden střih. Úlohou datové struktury je určit průsečíky polopřímky s mnohoúhelníkem, pokud existují.

Následující čas v táboře ubíhal našemu hrdinovi jen pozvolna. Přes usilovnou práci léčitelů (nebo právě kvůli ní) trvalo dva roky, než byl Jacob schopen chodit bez berlí. I poté většina procházek po chvíli skončila ostrou bolestí. Až po zhruba třech a půl letech byl Jacob schopen chodit asi půl dne v kuse, než přišla ona bolest.

Jak Jacob celý ten čas trávil? Inu, zažil ještě vesmírné výpravy před kryogenickým spánkem, přečkat pár let s omezenými možnostmi pohybu a lidského kontaktu pro něj nebylo nic nového. Našel si celou paletu činností. Velké úsilí věnoval například luštění mimozemského jazyka a po zhruba třech letech byl schopen se s mimozemšťany běžně dorozumět. Zdálo se však, že čím lépe Jacob hovořil mimozemsky, tím menší chuť s ním hovořit mimozemšťané měli.

Vynechme drobné Jacobovy zážitky z těchto časů a přesuňme se raději do doby zhruba čtyř let od ztroskotání. Tehdy se Jacob poprvé odvážil vypravit pěšky až k vraku lodi UFC Freya*, kdysi věhlasné nákladní lodi Spojené federace. Loď zarůstala zelení, působila však poměrně celistvě. K výbuchu poškozeného hlavního reaktoru evidentně nedošlo.*

U trosek lodi potkal Jacob mimozemšťanku. Po krátké konverzaci se ukázalo, že toto není její první návštěva vraku lodi. Ada, jak se mimozemšťanka jmenovala, dle svých slov společně se svými mimozemskými kumpány získala velkou část lodní knihovny.

V luštění beletrie prý mimozemšťané příliš úspěšní nebyli. Na lodi se však nacházelo mnoho knih s informatickou a matematickou tématikou. Na základě obrázků pak odvodili význam většiny pozemské matematické notace.

Adě se prý obzvlášť líbila kniha o teorii grafů. Toho se Jacob rozhodl využít.

Při svém pobytu se bavil řešením spousty úloh, které si pamatoval ze Země, ale nikdy dřív na ně neměl dostatek času. Jednu úlohu řešil marně celé čtyři roky a třeba by Adu mohla napadnout právě ta myšlenka, která jemu unikala.

26-3-3 Grafová — 9 bodů

Nechť k je libovolné celé číslo větší než jedna. Uvažme graf, jehož všechny vrcholy mají stupeň alespoň k. Tím myslíme, že z každého vrcholu vede alespoň k hran.

Dokažte, že v takovém grafu existuje kružnice délky alespoň $k+1$, a sestrojte algoritmus, který nějakou takovou kružnici najde.

Trvalo sice dlouho, než si vzájemně vyjasnili terminologii, nakonec se však dorozuměli a úlohu překvapivě svižně vyřešili.

Jacob byl setkáním s Adou doslova nadšen. Během pobytu v mimozemském táboře dávno přestal věřit, že by snad mohli existovat mimozemšťané se smyslem pro humor. Nebo nedej bože mimozemšťané, se kterými by se dalo jen tak volně povídat.

V průběhu roku se Jacob s Adou pravidelně scházeli. Zásadně vždy u onoho vraku. Tato setkání dávala Jacobovu životu na planetě smysl.

Jak se chýlil pátý rok soužití s mimozemšťany v táboře ke svému konci, dokázal Jacob pobíhat celý den po pralese, aniž by cokoliv cítil. Přišel čas zvažovat co dál. Rozhodně neměl chuť zůstávat u těchto prazvláštních mimozemšťanů. Sice se o něj pět let starali, nebyli však za celou dobu schopni si s ním rozumně promluvit a říci, co jsou zač. Na druhou stranu o Adě toho věděl ještě méně. Neměl ani to nejmenší tušení, kde a jak vlastně žije.

Během jedné noci tohoto období, kdy Jacob nevěděl kudy kam, jej ze spánku vyrušilo zašelestění. Nebral na něj žádný ohled a pouze se začal v posteli přetáčet na druhý bok. Uprostřed pohybu jej polekaly obrovské oranžové oči.

V úleku začal šmátrat po dýce, kterou měl vždy položenou u své postele. Než však pevně uchopil rukojeť dýky, rozpoznal v šepotu neznámé osoby Adin hlas.

Jacoba napadla celá řada dotazů. Než je stihl všechny vyslovit, Ada jej přerušila. „Není čas na dotazy," šeptala Ada, „Bratrstvo tě potřebuje."

Jacob se omezil na jediný dotaz. Jak se bez povšimnutí a bezpečně dostanou pryč z tábora? Kolem tábora byla totiž rozmístěna spousta pastí pro zneškodnění vetřelců neznalých terénu. V noci byly tyto pasti obzvláště zákeřné.

Na to měla Ada prostou odpověď. Ukázala Jacobovi pečlivě vypracovanou mapku rozmístění těchto pastí po okolí tábora.

26-3-4 Kladení pastí — 10 bodů

V této úloze budete navrhovat algoritmus pro hledání optimálního rozmístění pastí v pralese.

Prales je neprostupný a pohybovat se lze prakticky jen po vyšlapaných pěšinkách. K dispozici máte mapu terénu. Na mapě se nachází N křižovatek a M pěšinek. Každá pěšinka spojuje dvě křižovatky.

Pro každou dvojici křižovatek existuje právě jedna cesta (posloupnost pěšinek), po které se lze přemístit z první křižovatky na druhou. V informatické řeči bychom řekli, že mapa je *stromem*.

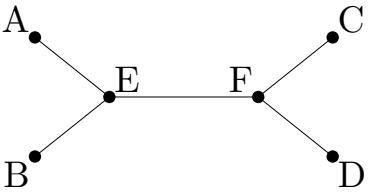
Pasti umisťujeme do křižovatek. Vyžadujeme, aby pro každou pěšinku platilo, že na alespoň jedné z křižovatek, které spojuje, leží past.

Umístit past na křižovatku nemusí vždy stát stejné úsilí. Pro každou křižovatku máte zadánu hodnotu U, která modeluje míru vynaloženého úsilí.

Navrhněte algoritmus, který nalezne rozmístění pastí do křižovatek tak, aby každá pěšinka měla na alespoň jednom ze svých konců past. Součet hodnot U všech křižovatek, na které algoritmus umístí past, musí být minimální možný.

Příklad: Představte si mapu cestiček jako na následujícím obrázku. V případě, že by umístění pastí stálo na všech křižovatkách stejné úsilí, bylo by nejlepší umístit pasti na křižovatky E a F, čímž bychom pokryli všechny cestičky.

Pokud by však ohodnocení úsilí vypadalo jako $U(A) = 1$, $U(B) = 1$, $U(C) = 2$, $U(D) = 3$, $U(E) = 3$, $U(F) = 4$, bylo by nejvýhodnější umístit pasti na křižovatky A, B a F za celkově 6 jednotek úsilí. Žádné jiné rozmístění pokrývající všechny cestičky nemá menší cenu.



Lehčí varianta (za 3 body): Vyřešte úlohu pro případ, kdy křižovatky budou tvořit jenom jednu nerozvětvenou cestu.

Ada s Jacobem pod hávem noci úspěšně opustili tábor a vydali se na cestu. Po cestě se Ada podělila o několik základních informací.

Ada je příslušnicí tajného společenstva s názvem Podzemní bratrstvo. Není mu však oprávněna v tuto chvíli prozrazovat cokoliv o poslaní Bratrstva. Bratrstvo má po okolí rozesetu síť základen. Všechny základny jsou řízeny z Hlavního štábu, kam právě směřují.

Hlavní štáb zabírá většinu rozsáhlého jeskynního komplexu, v řeči mimozemšťanů zvaného Ĝesrít íhgišpoƿ kiĝes sda ƿkuteraƿ. V posledních dnech se objevil problém, se kterým si Bratrstvo není schopno poradit. Chce proto požádat o pomoc Jacoba a využít jeho pozemských znalostí. Podrobnosti není Ada oprávněna sdělovat, žádost o pomoc chce vyslovit osobně Rada stařešinů.

Po poledni dorazili na pralesní mýtinku. Krom zpěvu ptáků neupoutalo nic Jacobovu pozornost. Ada na správném místě odhrnula listí a objevil se mřížový poklop. Poklop chránil úzký otvor, kterým se oba protáhli do jeskyně. Po průchodu takřka třísetmetrovou úzkou chodbou se objevili v ohromném dómu.

Ada pokynula hloučku mimozemšťanů, ať Jacobovi ukážou dóm. Sama zmizela v další chodbě vycházející z dómu hledajíc stařešiny. V dómu se nacházely desítky stanů, obydlí členů Bratrstva. Za stany se nacházel plácek sloužící jako shromaždiště. Zde byli každé ráno členové Bratrstva informováni o všem potřebném.

Dobrou čtvrtinu dómu zabírala mimozemská kovárna. Sestávala z asi tří výhní a dobrých dvou tuctů kovadlin. Mezi právě kovanými předměty Jacob zahlédl meč, lopatu, ba dokonce i svícen.

Jacob byl uchvácen sehraností mimozemšťanů, kladiva svými pravidelnými údery spíše než jako pracovní nástroje působila jako malý orchestr.

26-3-5 Rozvrh kovárny — 11 bodů

V této úloze se budete zabývat plánováním práce mimozemské kovárny vyrábějící všechny potřebné kovové nástroje.

Základny posílají kovárně své požadavky. Parametry požadavku jsou druh nástroje, priorita P a hodina H, do které je základna ochotna čekat. Než by dostala nástroj po hodině H, raději se zařídí jinak. Hodnoty P a H jsou kladná celá čísla.

Pro zjednodušení předpokládejme, že výroba jakéhokoli nástroje trvá jednu hodinu a během této doby nelze vyrábět nic jiného. Výroba začíná v hodinu 0, tedy na konci této hodiny může být vyroben první výrobek.

Na vstupu dostanete N požadavků, každý požadavek je určen dvojicí hodnot P a H. Protože výroba jakéhokoli nástroje trvá hodinu, typ nástroje se na vstupu vůbec neobjeví.

Váš algoritmus má za úkol sestavit optimální rozvrh výroby. To znamená pro každý požadavek určit hodinu, během které se požadovaný nástroj bude vyrábět,

případně -1, pokud požadavek nebude splněn vůbec. Výrobek může být vyroben nejpozději v hodinu H.

Rozvrh je optimální, pokud součet priorit splněných požadavků je nejvyšší možný.

Příklad: Pro požadavky $(4, 4)$, $(1, 1)$, $(2, 2)$, $(2, 3)$, $(4, 4)$ (zadané v pořadí priorita, hodina nutného dokončení) je jednou ze správných odpovědí $3, -1, 1, 0, 2$. Tedy vyrobíme předměty s celkovou prioritou 12 a druhý předmět nevyrobíme vůbec.

KSP

zadání

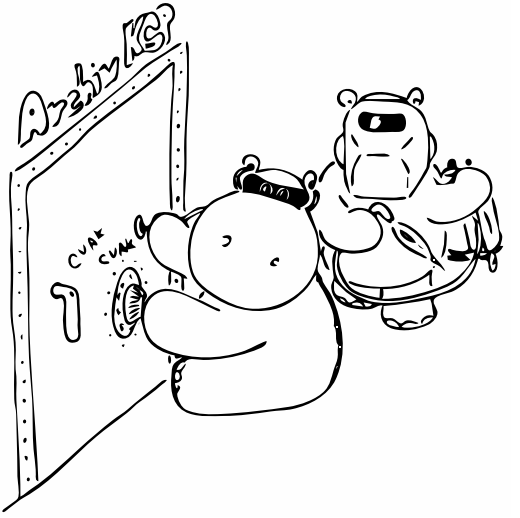
Z jedné z mnoha chodeb ústících v dómu se vynořila Ada následovaná trojicí mimozemšťanů. Nebylo pochyb o tom, že se jedná o stařešiny. Podsaditý mimozemšťan se představil jako Ubu, Tajemník Bratrstva a Vrchní stařešina. Přivítal Jacoba a pozval jej do svého stanu.

Jacob byl pohoštěn dobrým jídlem a dozvěděl se, že za hodinu bude oficiálně přivítán. Po hodině zdvořilé konverzace se rozezněly fanfáry. Když vyšli z Ubuova stanu, na shromaždišti už postávaly hloučky mimozemšťanů.

Ubu před davem představil Jacoba a oznámil, že Jacobovi bude jako výraz úcty Bratrstva předán dar. Asi dvacítka statných mimozemšťanů po těchto slovech donesla ohromný trezor.

Ke třem stařešinům, které už Jacob poznal, se přidalo dalších devět. Všichni stařešinové se shromáždili u trezoru.

Ke zdárnému otevření trezoru bylo nutno podniknout sérii přesně daných kroků. Nejprve bylo nutno ovládací páku přepnout do spodní polohy. Čtveřice mimozemšťanů poté asi minutu točila obrovitou klikou.



Stařešinové přistoupili k trezoru a každý vložil svůj klíč do otvoru v trezoru. Páka byla přepnuta do své horní polohy a čtveřice mimozemšťanů točila další dvě minuty klikou. Ozvalo se cvaknutí.

Další čtveřice mimozemšťanů otevřela těžké víko trezoru.

26-3-6 Trezor — 8 bodů

V této úloze budeme zkoumat mimozemský trezor. K otevření trezoru je potřeba do některých otvorů z vnějšku trezoru vložit sadu klíčů. Pokud je sada klíčů správná (správných sad může existovat více), trezor se po dostatečně dlouhém točení klikou otevře.

Princip trezoru tkví v tom, že všechny klíče odpovídají mocninám dvojky. Trezor má navíc ve svých útrobách skrytou druhou sadu klíčů. (Tvůrce trezorů totiž vyrábí všechny trezory stejně a až podle potřeb kupce navolí tyto vnitřní klíče.)

Během otáčení kliky vnitřní mechanická sčítačka sečte dohromady všechny klíče, jak vnitřní, tak ty zasunuté zvnějšku.

Je-li výsledný součet při zápisu v dvojkové soustavě tvořen samými jedničkami (na úvodní nuly se ohled nebere), trezor se otevře.

Vaším úkolem je pro zadanou N-tici vnitřních klíčů určit nejmenší počet vnějších klíčů potřebných k otevření trezoru. Klíče jsou zadány exponenty a nemusí být různé.

Poznámka: Možná jste se ještě nezabývali výpočetními modely a otázkou toho, o kterých operacích můžete prohlásit, že proběhnou v konstantním čase. Pak by vás mohlo napadnout umocnit dvojku na zadané exponenty a dále s nimi počítat. Věžte však, že výpočetní model, ve kterém bychom mohli v konstantním čase provádět aritmetiku nad libovolně velkými čísly, by byl absurdně silný.

Takový model by už neměl mnoho společného s tím, jak lze využívat skutečné počítače. Nejčastěji se proto uvažují modely, ve kterých lze v konstantním čase počítat pouze s čísly, která jsou polynomiálně velká vzhledem ke vstupním číslům. Takový model uvažujte i při řešení této úlohy.

Příklad: Pro seznam exponentů `0 1 0 1 0 2 4 4` lze trezor otevřít třeba pomocí pětice klíčů 2^1, 2^1, 2^3, 2^3 a 2^6.

To však není správná odpověď. Když použijeme pouze klíče o hodnotách 2^2 a 2^4, bude součet hodnot všech klíčů 63. To je ve dvojkovém zápise číslo `111111` a trezor se také otevře. Použít pouze jeden klíč pro zadaný příklad již nepostačuje.

⇧ **Lehčí varianta (za 2 body):** Vyřešte úlohu pro případ, kdy všechny exponenty vnitřních klíčů budou různé.

Ubu z trezoru vytáhl meč, došel k Jacobovi, poklekl a v natažených rukou mu nabídl tento dar. Za potlesku davu se stal Jacob majitelem mimozemského meče.

Po ceremoniálu se Jacob, tentokrát společně se všemi stařešiny, přesunul zpátky do Ubuova stanu. Ubu Jacobovi stručně povyprávěl o historii meče. Pak najednou zvážněl. Všichni stařešinové si sesedli blíž k Jacobovi. Zjevně nadešel čas poodhalit Jacobovi účel zdejší návštěvy.

„Drahý příteli," začal svou řeč Ubu, „skutečnost je prostá. Bratrstvu hrozí zkáza. Během posledního týdne se začal probouzet vulkán nedaleko Hlavního štábu. Pokud se potvrdí nejhrozivější obavy našich šamanů, láva z vulkánu zaplaví tento jeskynní systém. Nemáme dostatek jiných jeskyní, kam bychom přesunuli všechen náš lid, a na povrchu by jej stihla zkáza. Žádáme tě jménem celého našeho lidu o pomoc."

„Drazí přátelé," volil Jacob opatrně svá slova. „Učiním vše, co bude v mých silách. Obávám se však, že toho nebude mnoho. Má loď ztroskotala a nezdá se, že by jakákoli technologie z ní byla použitelná. Rád bych se však na onen vulkán alespoň podíval."

„Dobrá," odvětil trochu zklamaně Ubu a na chvíli se zamyslel. „Vyšlu s tebou své nejlepší lidi, budou ti ve všem nápomocni."

O pár hodin později stál Jacob spolu s Adou a dalšími čtyřmi mimozemšťany u úpatí sopky. Z jejího vrcholku pomalu stoupaly malé obláčky dýmu.

Jacob se zamyslel. Co od něj Bratrstvo vůbec očekává? Jistě, na Zemi dávno existovala technologie, která by se s touto hrozbou vypořádala. Co si však lze počít zde? Zdálo se, že sopka lehce smete Bratrstvo. Neměl pocit, že by si to nechala jen tak vymluvit...

26-3-7 Sopečné pokrytí — 12 bodů

Pro zabezpečení sopky je na Zemi nutno podniknout dvě opatření – zahájit odčerpávání lávy ze sopouchu speciálním potrubím a pokrýt ústí sopky panely tak, aby neunikal sopečný popel ani gejzíry lávy. Vaším úkolem bude nalézt optimální pokrytí.

Okolí sopky si můžeme představit jako čtvercovou síť velikosti $R \times S$. Na každém políčku se nachází buď znak `Z`, nebo `K`. Pomocí `Z` je označena zem, kterou není potřeba pokrývat, naopak znaky `K` označují oblast sopečného kráteru, kterou je nutno pokrýt. Oblast kráteru je souvislá.

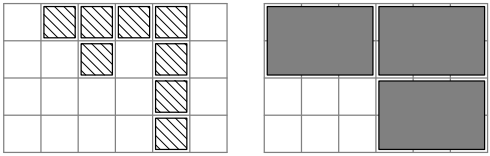
V celé úloze budeme za sousední políčka považovat ta, která se dotýkají celými stranami. Souvislou oblastí pak rozumíme množinu políček takovou, že kdykoli máme políčka A, B z této oblasti, existuje posloupnost políček z této oblasti s následujícími vlastnostmi: prvním políčkem je A, posledním B a pro každé políčko (mimo posledního) platí, že následující políčko je jeho sousedem.

Pokrývá se panely obdélníkového tvaru rozměrů $\varrho \times \sigma$. Pro zjednodušení úlohy budeme předpokládat, že panel lze položit pouze tak, aby pokryl obdélníkovou podoblast čtvercové sítě o ϱ řádcích a σ sloupcích.

Aby celé pokrytí fungovalo, musí panely do sebe zapadnout. To znamená, že pokud spolu dva panely sousedí nějakými políčky, pak spolu musí sousedit celými svými stranami. Panely mohou sahat i za hranice původní mapy.

Nalezněte panelové pokrytí, které pokryje všechna políčka se znakem `K` a využije k tomu nejmenší možný počet panelů.

Příklad: Na obrázku níže vidíte jedno z možných pokrytí kráterů vlevo pomocí panelů o rozměrech 2×3.



Nezdálo se však, že by podobná technologie byla k dispozici zde. Jak si vlastně lidstvo umělo poradit se sopkami v dobách, kdy ještě neexistovaly materiály dost odolné k jejich usměrnění? Přeci by jim neustupovalo...

Z úvah Jacoba vytrhlo zapraskání. Znělo to skoro, jako by se sopka po probuzení potřebovala nejprve protáhnout. Po dalších dvou zapraskáních nastalo opět ticho. Nemělo však dlouhého trvání, bylo po chvilce prořízнuto ránou tak ohromnou, že si všichni instinktivně zacpali uši. Ze sopky se vyvalil oblak popela.

Šedivé kousky popela zvolna jako sníh dopadaly na Jacoba a mimozemšťany. Všichni stáli na místě jako přikovaní. Nikdo nebyl schopen slova. Začaly se objevovat první gejzíry lávy. První se vzpamatoval Jacob a zavelel všem k útěku.

Ozval se další výbuch. Tentokrát se už nejednalo o pouhý gejzír. Zformoval se celý proud lávy, který si jako horská bystřina razil svou cestou krajinou. Zřejmě se vydal stíhat prchající skupinku. Jacob se ohlédl a odhodil svůj batoh. Pochopil, že bude muset být opravdu rychlý.

Čtvrtá série

Jacob pomalu popadal dech. Právě se mu podařilo uniknout před proudem lávy valící se z rozběsněné sopky. Dokonce našel malý převis, který ho uchránil před deštěm kamení a žhavého sopečného popela.

Jeho myšlenkami letěly vzpomínky na události, které zažil od okamžiku, kdy ho ztroskotání vesmírné lodi UFC Freya *uvěznilo na této planetě. První opatrné zkoumání pralesa. Setkání s mimozemšťany. Polámané nohy. Dlouhé léčení. Tajuplná Ada. Podzemní bratrstvo. Darovaný meč. Probuzený vulkán ...*

Podařilo se aspoň části členů Bratrstva utéci do bezpečí, nebo všechny v jejich podzemním komplexu zaplavila láva? To se Jacob ještě nějaký čas nedozví – teď mu totiž nezbývá než vyčkat v úkrytu, dokud se sopka neuklidní. Aby zahnal nervozitu, zkouší hrát nejrůznější hry pro jednoho hráče neboli solitéry. Třeba s kamínky, těch je teď všude plno.

26-4-1 Kamínkový solitér — 10 bodů

Jacob hraje následující hru. Nejprve vytvoří n hromádek kamínků. Pak odebírá kamínky podle následujícího pravidla: Vždy si vybere dvojici různě velkých hromádek a z té větší odebere tolik kamínků, kolik jich je na té menší. Cílem hry je zbavit se co nejvíce kamínků.

Vymyslete algoritmus, který pro tuto hru najde optimální strategii. Na vstupu dostane počet hromádek n a počáteční počty kamínků $h_1, \ldots, h_n$. Výstupem má být posloupnost tahů typu „od h_i odečti h_j“ taková, že po provedení těchto tahů bude součet $h_1 + \ldots + h_n$ nejmenší možný.

⇧ **Lehčí varianta (za 4 body):** Rozmyslete si, jak úloha dopadne pro dvě hromádky. Své tvrzení dokažte.

Jacob se s trhnutím probudil. Co to bylo? Venku byla tma jako v pytli, hvězdy pohltil všudypřítomný sopečný prach. Opodál cosi šramotilo. „Psst, Jacobe, to jsi ty?“ špitl hlas nápadně podobný Adinu. Po chvilce oboustranného ujišťování se ze tmy vynořila Ada spolu s dalšími čtyřmi mimozemšťany, kteří s Jacobem zkoumali kráter sopky. Bylo to včera, ale zdálo se to jako věčnost ...

Přeci jen se jim podařilo uniknout potokům prskající lávy a schovat se ve stejných skalách, které poskytly azyl Jacobovi. Jacob si pomyslel, že jsou jistě celí žhaví zjistit, co se událo v okolí. Nejspíš i doslova. Každopádně jim nezbývá než zůstat v úkrytu, dokud se popel nerozptýlí. Zatím není vidět na krok a kdo ví, jak se erupcí změnila krajina.

Sedli si tedy společně na teplou zemi. Ze začátku tiše, ale po chvíli jeden z tvorů vytáhl jakési zvíře podobné bažantovi, které našel pod vrstvou rozpáleného popela. Zjevně dobře upečené. Po dobrém jídle nervozita opadla a Jacob využil příležitosti a zeptal se na pár věcí, které mu už delší dobu vrtaly hlavou.

Například proč mají všichni členové Bratrstva na krku podivný amulet – náhrdelník s řadou barevných korálků. Každý s jinou kombinací barev, ale přeci jen bylo možné vysledovat určité podobnosti. Ada usoudila, že teď už před Jacobem není potřeba nic tajit. Prozradila mu, že amulet slouží jako poznávací znamení členů Bratrstva a obsahuje heslo, které se čas od času mění.

26-4-2 Výroba amuletu — 10 bodů

Amulet definujeme jako posloupnost červených, zelených a modrých korálků. Heslo je také nějaká posloupnost korálků. Amulet obsahuje heslo, pokud lze z amuletu vypustit některé korálky tak, aby zbylo právě heslo. Matematik by tedy řekl, že heslo tvoří vybranou podposloupnost amuletu.

Ada zná nové heslo a chce svůj amulet upravit tak, aby toto heslo obsahoval. Upravovat ho může vkládáním korálků na libovolné místo. Ovšem výroba jednoho korálku trvá nějaký čas závislý na jeho barvě, tak by chtěla vymyslet, kam vložit který korálek, aby tím celkově strávila co nejméně času.

Vymyslete algoritmus, který jí v tom pomůže. Na vstupu dostane dva řetězce písmen `R`, `G` a `B`: *amulet* a *heslo*. Mimo to ještě dostane celá kladná čísla c_R, c_G a c_B udávající, kolik času trvá vyrobit korálek které barvy.

Výstupem algoritmu má být posloupnost operací „za i-tý korálek vlož korálek barvy b“, která zabere nejkratší možný čas.

Noc pokračuje. Skupinka se snaží usnout, ale ve stísněném prostoru pod převisem to jde jen obtížně. Polštáře tu nejsou, tak musel Jacob vzít zavděk jakýmsi kamenem. Nepříjemně tlačil do ucha a navíc se z něj ozývalo podivné zvonění, jako by v hlubinách planety nějaký skřítek mlátil kladivem do skály.

Zvonění je ale podivně pravidelné. Skoro jako by si ti skřítci posílali nějaké zprávy. Jacob místo počítání oveček přemýšlí, jak by takový přenos zpráv mohl fungovat. Snaží se vymýšlet různé způsoby a zkoušet pomocí nich zvonění dešifrovat. Evidentně to k ničemu nebude, ale aspoň svou mysl unaví a konečně usne.

„J. . . S. . . M. . . E. . . Z. . . A. . . S. . . Y. . . P. . . “ Cože??!!! Jacob ihned vzbudil ostatní a společně naslouchali skalám. Text byl poněkud zmatený, postupně však pochopili, že se jedná o víc překrývajících se zpráv. Posílají je skupinky členů Bratrstva uvězněné na různých místech v podzemí.

Do jeskynního systému nejspíš nenatekla žádná láva, ale výbuch na mnoha místech zavalil chodby. Ada hned začala do vrstvy popela zapisovat, které části podzemí zůstaly propojené, ale situaci znepřehledňovalo, že se zprávy o spojení jeskyní často opakovaly.

26-4-3 Obnovené spojení — 10 bodů

Mějme n jeskyní a m neuspořádaných dvojic $\{x_i, y_i\}$, které popisují, že jeskyně x_i je propojena s jeskyní y_i. Navrhněte co nejefektivnější algoritmus, který z tohoto seznamu odstraní opakující se dvojice.

Výstupem je tedy seznam navzájem různých dvojic, které se alespoň jednou vyskytly ve vstupním seznamu. Na pořadí dvojic nezáleží.

Ada dokreslila plán podzemí a ve tváři jí zazářila radost. Právě zjistila, že navzdory všem závalům stále existuje cesta, jak se do všech obydlených jeskyní dostat. Značně složitá, ale je tu.

Vzduch, který se mezitím trochu pročistil, ovšem odhalil, že okolní skaliska jsou zasypaná hromadami sopečného popela, tufu a kamení. Dříve důvěrně známé kopce se najednou proměnily v nepřehlednou krajinu plnou skrytých nebezpečí.

Skupina se rozdělila a každý dostal za úkol důkladně, ale velmi opatrně prozkoumat část okolí a pokusit se nakreslit mapu. Když se vrátili, zjistili, že mapy se poněkud překrývají. Některá místa nejsou zmapována vůbec, zatímco jiná velmi důkladně. Jak se v tom vyznat?

26-4-4 Skládání mapy — 12 bodů

V rovině je položeno několik kusů mapy. Kusy mají tvar obdélníků se stranami rovnoběžnými s osami souřadnic.

Naším úkolem je vytvořit datovou strukturu, která bude umět rychle odpovídat na dotazy typu „v kolika obdélnících leží zadaný bod?“

Důležitá je přitom jak časová složitost dotazů, tak čas potřebný na vybudování struktury.

Konečně se podařilo poskládat jednotlivé mapy do jednoho jakž takž použitelného celku a naplánovat záchrannou akci. Než se ale podaří zpustošené podzemí vrátit do obyvatelného stavu, bude potřeba vybudovat pro všechny provizorní tábor v horách.

Tiše přemýšleli, kde ve skalách vzít kousek rovné plochy. Naštěstí sopečné tufy a popel jsou lehké, takže menší nerovnosti půjde snadno srovnat. I tak by ale bylo milé si co nejvíc práce ušetřit. Sesedli se okolo mapy a uvažovali nad vhodným místem.

26-4-5 Místo pro tábor — 14 bodů

Je dána výšková mapa krajiny. Terén je rozdělen na $R \times S$ políček a pro každé z nich známe jeho nadmořskou výšku v mimozemských pídích.

Na nějakém místě chceme postavit tábor. To obnáší vybrat obdélníkovou část krajiny o rozměrech $r \times s$ políček a srovnat ji do roviny. Tedy přesunout

mezi těmito políčky zeminu tak, aby všechna políčka byla stejně vysoko. Jednotky si zvolme tak, že zvýšení políčka o jednu mimozemskou píď vyžaduje přivezení jedné mimozemské kárky zeminy.

Vaším úkolem je pro zadanou výškovou mapu a velikost tábora najít takové místo pro tábor, abychom museli přesunout co nejméně zeminy.

Zemina je neomezeně dělitelná, ale lze ji pouze přesouvat. Není možné ani vytvořit zeminu z ničeho, ani ji zničit.

⇧ **Lehčí varianta (za 10 bodů):** Nalezněte řešení pro jednorozměrnou krajinu ($S = s = 1$).

Tábor utěšeně rostl. Proudili do něj stále noví členové Bratrstva, vysvobození z čím dál vzdálenějších částí jeskynního systému. Na krajinu se snášela další noc, mnohem optimističtější než ty předchozí.

Středu tábora vévodilo veliké ohniště, u kterého právě odpočíval Ubu spolu s několika staršími mimozemšťany. Jacob si k nim přisedl. Chtěl totiž využít klidné chvilky a dozvědět se něco o tom, co je Bratrstvo zač a proč se tolik snaží svou existenci utajit.

A důvody k tajnostem skutečně existovaly: Bratrstvo organizovalo odboj proti místnímu králi. Členové královské dynastie byli sice považováni za potomky bohů (a dokonce prý vypadali trochu jinak než jejich poddaní), ale vládli velmi nevybíravě a nebývale krutě.

Spiklenci už dlouho připravovali plán na svržení panovníka. Podezírali ale krále, že o jejich úmyslech ví a že se celého Bratrstva pokusil zbavit výbuchem sopky vyvolaným magií. Jacob se tvářil značně nedůvěřivě – na kouzla nevěřil a ještě méně pravděpodobné bylo, že by kdokoliv na této planetě disponoval potřebnou technikou.

Ale dost už pochybností, dnes je den mnoha šťastných shledání a takový si zaslouží oslavu. Jacoba napadlo, že by ostatní mohl naučit nějakou pozemskou hru. Všiml si, že sopečný tuf je natolik měkký a lehký, že z něj jde nožem vyřezávat něco jako sněhové koule. Sice tolik nestudí, vlastně vůbec, ale házet jdou úplně stejně. Hej! Kryj se! Pal!

26-4-6 Sněhová bitva — 11 bodů

V rovině stojí n bojovníků se sněhovými koulemi v rukou. Za chvíli začne velká řež. Pokud bojovník A hodí kouli po bojovníkovi B, může trefit kohokoliv, kdo se nachází na polopřímce AB.

Na vstupu dostanete polohy bojovníků v rovině. Vaším úkolem je vytvořit datovou strukturu, pomocí které budete umět efektivně odpovídat na dotazy „Pokud A míří na B, může zasáhnout někoho dalšího?“. Jako odpověď stačí ANO nebo NE, není potřeba hledat, koho zasáhnete.

Do všeobecného veselí se vkrádaly stíny nedůvěry. Jak se mohl král o existenci Bratrstva dozvědět? Není mezi nimi nějaký špión, nebo dokonce víc takových? Královská tajná policie je přeci svými schopnostmi po celé říši proslulá.

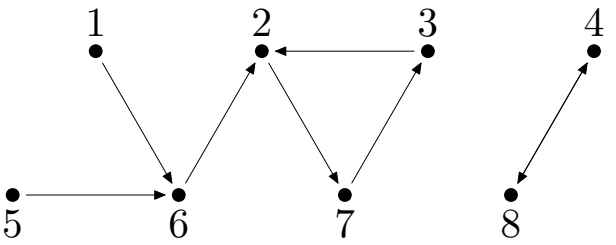
Jacoba napadlo, že to mohl být důvod, proč se k němu jeho někdejší léčitelé chovali tak uzavřeně a odmítali mu odpovídat na jeho otázky. Konec konců, královská rodina přeci má vypadat jinak než ostatní obyvatelé planety, tak není divu, že byl Jacob krajně podezřelý. O to víc si vážil důvěry Bratrstva.

Teď s Ubuem probírali různé hypotézy, jak by si mohli královi zvědové předávat informace.

26-4-7 Královští špioni — 9 bodů

Král má n špionů. Každý špion má pevně určeno, kterému jednomu špionovi předává všechny informace, jež zjistí.

Špionská síť s osmi špiony může například vypadat následovně:



Pokud špion dostane zprávu, kterou už zná, neposílá ji dál. Šíření každé zprávy se tedy po konečném počtu kroků zastaví.

Váš algoritmus dostane zadanou síť špionů. Jeho úkolem je pro každého špiona spočítat, jak dlouho bude v síti putovat zpráva, kterou tento špion vyšle.

V síti na obrázku jednotlivé zprávy urazí postupně v pořadí dle čísla vysílajícího špiona $5, 3, 3, 2, 5, 4, 3$ a 2 kroků.

Jakmile v táboře přestala být potřeba každá pomocná ruka, Jacob dostal chuť projít se po okolí. Chtěl si zblízka prohlédnout ztuhlé potoky lávy, na kterých se utvořily zajímavé obrazce.

Zvolna kráčel mezi skalami a sledoval pustou krajinu. Připomněla mu povrch Měsíce. Posmutněl, když si uvědomil, že tam už se nejspíš nikdy nepodívá.

Najednou mu nohy uvázly v sopečném popelu. Když se je pokusil vyprostit, jenom se propadl o něco hlouběji. Zjevně narazil na jámu plnou popela. Marně se pokoušel rukama zachytit okraje. Sjížděl čím dál rychleji. „Už zase!“ blesklo mu hlavou.

Zadání úloh KSP – 4. série

Proletěl jakousi šikmou chodbou a přistál na podlaze nevelké jeskyně. Všude se válely podivné kovové krabice. Značně omšelé a propojené zašlými zkroucenými kabely. Na nejbližší z nich zahlédl kovový štítek s nápisem.

Stálo na něm: „Made in China.“ Uhhh...

Pátá série

Tohle je přeci lidská technika! A určitě to nejsou jen náhodně popadané trosky z Freyi*, protože tady je to celé propojené. To znamená… „Jsi v pořádku?“ přerušil náhle jeho myšlenky hlas shora. Jacob vzhlédl a spatřil jednoho mimozemšťana, jak klečí na okraji díry, kterou sem propadl.*

Ubu ho asi poslal, aby na něj dával pozor. No teď se to hodí, pomyslel si Jacob. Poslal mimozemšťana pro posily a za půl hodiny už táhli největší z kovových beden do tábora.

Jacob chtěl přijít na to, k čemu zařízení slouží. Jelikož bylo, zdá se, vyrobeno jako co nejlevnější a nejuniverzálněji použitelné zařízení, skládalo se jen z univerzální řídící jednotky, která byla naprogramována pomocí spousty relátek uspořádaných v pravidelné mřížce. Bohužel některá z nich byla spálená a bylo nutné je vyměnit.

26-5-1 Made in China — 9 bodů

Jacob má před sebou podivné zařízení a chtěl by ho spustit. Jenže předtím musí vyměnit několik spálených relé. Každé relé má své typové označení (přirozené číslo), které jde poznat po vyndání relé ze zařízení. Avšak spálená relé mají označení bohužel spečená k nepoznání.

Podle schématu na zařízení víme, že původně byla relé uspořádaná podle tabulky, kde v horní řádce i na levém okraji této tabulky byla relé označená postupně rostoucími přirozenými čísly začínajícími nulou $(0, 1, 2, 3, \ldots)$.

Zbytek tabulky byl vyplněn tak, aby se na pozici $[A, B]$ nacházelo relé s nejmenším takovým číslem, které se doposud doleva ani nahoru od něj nevyskytlo (všechny tyto pozice již samozřejmě musí být osazeny).

Prvních šest řádků a sloupců této tabulky tak bude vypadat takto:

```
0 1 2 3 4 5
1 0 3 2 5 4
2 3 0 1 6 7
3 2 1 0 7 6
4 5 6 7 0 1
5 4 7 6 1 0
```

Jacob potřebuje vyměnit několik spálených relé, ale nechce se mu kvůli tomu zkoumat všechna relé doleva a nahoru od těchto pozic. Proto by od vás potřeboval postup, který by mu rychle řekl, jaké relé má umístit na pozici $[A, B]$.

Po osazení všech relé se zařízení spustilo a zobrazilo na malé obrazovce několik informací. Po zběžném pročtení bylo Jacobovi vše jasné. Za pár minut už Ubu a několik dalších starších seděli v jednom ze stanů.

„Ctihodný Ubu, viděl někdy někdo z vašich lidí přímo někoho z vládnoucí kasty?“ zeptal se rovnou.

„Ne, podle našich zvědů nikdy nesundávají své blyštivé helmy, Jacobe.“

Jacob se nadechl, tohle by mohlo být ošemetné: „Ctihodný Ubu, rado starších, věc, kterou jsem vykopal, pravděpodobně spustila výbuch sopky. Je to asi nástroj, který sem umístil někdo z vládcových lidí. Ten nástroj ale poznávám,“ nadechl se, „je to technika mých lidí, nebešťanů.“

Viděl, jaké reakce to mezi radou starších vyvolalo, a tak rychle pokračoval: „O těch lidech nic nevím, možná to bude nějaký odštěpený klan. Musím se ale vypravit do královského města a promluvit si s nimi, přesvědčit je o tom, že dělají špatnou věc.“

Ještě chvíli diskutoval s radou, než se mu ji povedlo přesvědčit, že ho mají nechat jít samotného. Přízvuk už měl docela dobrý, tak dostal alespoň místní ekvivalent kutny, kterou tu občas nosili starší mimozemšťané a která mu zakrývala obličej i celé tělo. S ní by snad mohl splynout s místními. Mimo to si ještě vzal meč, který získal při rituálu, a dostal od Ady docela přesnou mapu pralesa. Rozloučil se s místními, minul místní ekvivalent kočky ležící na kraji tábora a vyrazil.

Jako první se vydal k vraku Freyi. *Tam si cestu pamatoval, a tak si během ní plánoval trasu od vraku směrem ke královskému městu. Bude to dlouhá cesta, vedoucí v první části skrz hustý prales, kde bude muset dávat pozor, aby v něm nepřehlédl žádnou odbočku.*

26-5-2 Cesta pralesem — 9 bodů

Prales je místo, kde se velmi lehce přehlédne pokračování cesty. Máme mapu pralesa představovanou čtvercovou mřížkou o rozměrech $R \times S$ políček, dále máme zadané startovní políčko a cílové políčko.

Pohybovat se v mapě můžeme jen horizontálně nebo vertikálně, šikmo ne. Každé políčko má navíc dva *koeficienty přehlédnutelnosti*. Jeden pro cestu vedoucí přes něj rovně (shora dolů, zleva doprava nebo naopak) a druhý pro odbočení (tedy všechny zbylé průchody – zleva dolů, shora doleva, ...).

Protože chceme mít co největší naději, že se v pralese neztratíme, je vaším úkolem nalézt cestu ze startovního do cílového políčka, která bude mít v součtu

přes všechna políčka cesty co nejmenší přehlédnutelnost (přehlédnutelnost ve startovním a cílovém políčku neuvažujte).

Než si Jacob naplánoval celou další cestu, už stál u boku UFC Freya. *Torzo lodě i po těch letech stále vypadalo majestátně, ale brázda v pralese i poničené okolí po nouzovém přistání už pomalu zarůstaly novou vegetací.*

Během svého několikaletého pobytu v mimozemském táboře se do vraku párkrát podíval, ale vždy se vydal jen k výstrojnímu skladu pro nové boty. Od ztroskotání nikdy nezašel dál, to místo na něj z nějakého důvodu působilo tísnivě a stejně tam nebylo nic zajímavého – nouzový signál nešel vyslat kvůli nějakému rušení v atmosféře a vody i jídla měl od mimozemšťanů dost.

Teď se skrz zprohýbaný koridor prodral hlouběji do lodi. Freya *jako původně válečná transportní loď měla v přídi několik zabezpečených skladů a Jacob je chtěl zkontrolovat. Zastavil se u zavřené přepážky, která nesla stopy po použití malého plazmového hořáku – někdo se chtěl propálit skrz, ale malým hořákem se mu to zdá se nepovedlo. To znamená, že se tady objevil nějaký jiný člověk, pravděpodobně někdo z těch, kteří teď vládnou domorodcům.*

Jacob odklopil ovládací panel a několika stisky probudil palubní počítač z jeho spánku. Chvíli trvalo, než naběhl, ale záložní baterie stále poskytovaly dostatek energie. Potom, co Jacob zadal svůj přístupový kód, se ale nic nestalo. Zkusil to znovu, a teprve pak si všiml, jakou paseku na kabelech neznámý návštěvník s plazmovým hořákem provedl. Ještě že ve skladu údržby byla celá krabice náhradních – jen rychle najít ten správný.

26-5-3 Náhradní kabel — 10 bodů

Máme před sebou bednu plnou náhradních kabelů a potřebujeme rychle poznat, jestli jsou nějaké dva kabely stejné. Nejsou to ale jen kabely se dvěma konci, tyto jsou rozvětvené a mohou obecně propojovat i více věcí dohromady.

Každý kabel si tedy můžeme představit jako spoustu uzlů pospojovaných jednotlivými dráty. Celý kabel je navíc pospojovaný tak, že v něm neexistují cykly – z každého uzlu do každého jiného se lze dostat jen jedinou cestou. Informatik by tedy takovýto rozvětvený kabel mohl nazvat *stromem*.

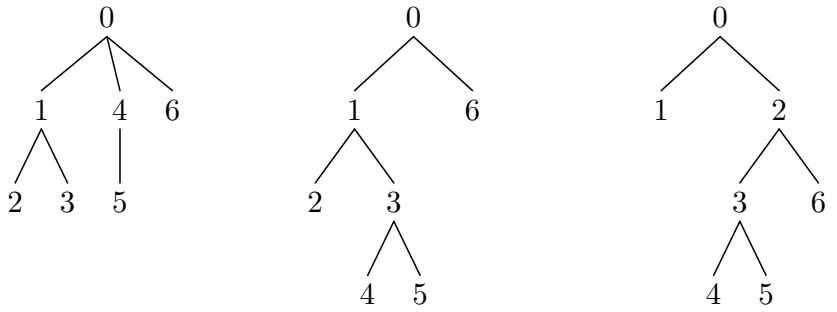
Jacob vždy náhodně uchopí dva kabely, a to tak, že je chytí za nějaký uzel a zbytek nechá viset dolů. Tomuto uzlu budeme říkat *kořen*. Díky tomu, že kabely jsou v uzlech spojeny v pevném pořadí, můžeme lehce popsat zbytek kabelu například tak, že kořen označíme indexem 0 a zbylé uzly vyjádříme jako N čísel označujících, pod kterým jiným uzlem je připojený i-tý uzel.

V případě více uzlů připojených pod ten samý je uvedeme třeba v pořadí proti směru hodinových ručiček (rozmyslete si, že je jedno, od jaké pozice začneme připojené uzly popisovat, když dodržíme jejich pořadí).

Jacoba by nyní zajímalo, jestli náhodou nejsou dva kabely, které drží v ruce, stejné. To znamená, že kdyby si je chytl oba za správný uzel, vypadaly by pak oba stejně (mimo přechycení za jiný uzel nesmíme pořadím kabelů v jednotlivých uzlech nijak rotovat).

Příklad: Pro kabely níže stačí, abychom prostřední kabel chytili za uzel s indexem 1 a správně otočili, pak budou levý a prostřední kabel stejné (všimněte si, že uzel 2 se sice zrotoval na druhou stranu, ale pořadí uzlů připojených k uzlu 1 se tím nijak nezměnilo). Nicméně pravý kabel se od nich liší (má jiné pořadí podstromů v uzlech).


```
Levý          Prostřední    Pravý
kabel           kabel       kabel
1: 0            1: 0        1: 0
2: 1            2: 1        2: 0
3: 1            3: 1        3: 2
4: 0            4: 3        4: 3
5: 4            5: 3        5: 3
6: 0            6: 0        6: 2
```



Po zapojení správného kabelu zadal Jacob znovu svůj přístupový kód. Panel zezelenal a s hrozivým skřípotem se pancéřová přepážka otevřela asi do poloviny. Jacob prolezl skrz a ocitl se v sekci lodě, ve které od startu nebyl. Matně si pamatoval, že sem nakládali nějaké záhadné bedny s tajným obsahem – stále zde stály a vypadalo to, že jim nouzové přistání ani příliš neublížilo.

Jacoba ale zajímalo něco jiného. V malém skladu na konci oddělení se nalézala bedna, o níž vědělo jen několik členů bývalé posádky. Pomalu ji otevřel a vytáhl pár věcí ven. Pak zvedl plazmovou karabinu, zapřel si ji o rameno a pohledem skrz zaměřovač vyzkoušel, že vojenská zbraň stále funguje. Myslí mu proletěly vzpomínky na válku... jeho služba u elitní roty mariňáků, kamarádi v jednotce a jejich heslo Semper Fidelis – vždy věrní.

Zatřepal hlavou a zahnal vzpomínky. Je to už skoro dvacet let, co po konci války opustil armádu. Schoval masivní zbraň do batohu, sundal z helmy noční

vidění a termovizi, k boku si připnul malou pistoli na uspávací šipky a pobral ještě několik drobností včetně sady nářadí. Pak vyrazil.

Podle mapy odhadl, že královské město bude ležet něco přes pět set kilometrů od místa, kde havaroval. Během svého putování minul několik mimozemských osad, ale vždy se držel mimo ně. Patnáctý den cesty, odhadem půlden cesty od královského města, mu však už došlo všechno jídlo. Chtěl si před setkáním raději pořádně odpočinout, a tak se rozhodl, že navštíví nejbližší vesnici, na kterou narazí.

Jednu takovou objevil k večeru. Asi hodinu ji pozoroval, než se odvážil ukázat se. Zdálo se, že místní náčelník zrovna řešil nějaký problém s rozdělováním surovin na výrobu jídla.

26-5-4 Rozdělování jídla — 11 bodů

Vesnický náčelník má k dispozici K typů různých surovin a chce je co nejlépe využít, aby mu jich dohromady zůstalo co nejméně nepoužitých. Od každé suroviny má na začátku nějaké množství $m_1, \ldots, m_K$ $(m_i \geq 0)$.

Obyvatelé vesnice mu předložili N receptů. Každý recept spotřebuje nějaké množství od každé suroviny, tedy je zadaný K čísly $a_1, \ldots, a_K$ $(a_i \geq 0)$ a lze ho použít maximálně jednou (a to jen tehdy, pokud ještě od každé suroviny zbývá alespoň tolik, kolik recept spotřebuje).

Vaším úkolem je ze všech N receptů vybrat takové, které dohromady zanechají co nejmenší zbytek surovin (tedy pokud si zbytky označíme jako $z_1, \ldots, z_K$, tak $\sum_{i=1}^{K} z_i$ bude co nejmenší možné).

Potom, co se s nimi Jacob najedl, poznal podle náhrdelníku s tajným heslem v jednom z domorodců člena Bratrstva. Počkal, než odejde stranou, a pak se za ním nenápadně vydal.

Ve chvíli, kdy si před ním Jacob sundal z hlavy kápi, se mimozemšťan zarazil, jako by ho někdo praštil palicí po hlavě. Pak ale jeho pohled sklouzl na náhrdelník Bratrstva, který Jacob dostal od Ady, a došlo mu to: „Ty jsi ten neznámý cizinec, který nám slíbil pomoc, že? Co děláš takhle daleko od hlavního tábora? A mimochodem, já jsem Lakus.“

Jacob mu všechno vysvětlil a také mu řekl, že by se potřeboval nenápadně dostat do královského města, aby si promluvil s králem. To, že jsou to taky pozemšťané jako on, raději nezmiňoval.

Mimozemšťan se zamyslel a pak pravil: „Pracuji se skupinkou řemeslníků, kteří do města dopravují kámen. Využíváme k tomu staré podzemní tunely, tudy by ses tam mohl dostat.“

⋆⋆⋆

Druhý den ráno už Jacob pomáhal tlačit velký povoz kamení naložený tak, že byl trojnásobně vyšší než on sám. Měl na sobě stále kápi, ale batoh s karabinou

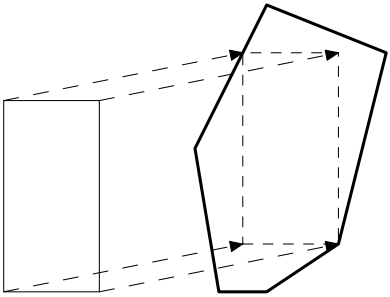
a největšími věcmi dal raději Lakusovi do úschovy, nechtěl aby se případně dostaly do rukou královým lidem.

Právě se blížili k vstupu do tunelu, město se tyčilo na ostrohu nad nimi. Jacob si mimo jiné všiml pozemsky vyhlížející antény na největším z domů. Náhle je zastavilo zavolání jednoho z hlídačů u ústí do tunelu. Začal se hádat s předákem skupinky, jestli se takhle velký povoz do tunelu vejde, nebo jestli ho bude nutné složit tady a náklad odnosit ručně.

26-5-5 Průjezd tunelem — 12 bodů

Potřebujeme zjistit, jestli náš povoz projede tunelem. Povoz má průřez obdélníku rovnoběžného se souřadnými osami, průřez tunelu je představován konvexním mnohoúhelníkem (zadaným na vstupu třeba jako posloupnost vrcholů po směru hodinových ručiček).

Ptáme se, jestli se povoz vejde do tunelu, neboli jestli existuje nějaké posunutí obdélníku takové, že se celý obdélník vejde dovnitř konvexního mnohoúhelníku. Obdélník můžeme posouvat v libovolném směru (tedy klidně i nahoru a dolů), ale nesmíme ho otáčet. Pokud takové posunutí existuje, najděte ho.



Po proniknutí do města se Jacob oddělil od skupinky mimozemšťanů. Opatrně proklouzl okolo stráží a vyšplhal na vyvýšenou věž na okraji hradeb. Zde zalehl a vytáhl z kapsy dalekohled.

Pozorně si prohlédl antény, které už dříve spatřil. Podle toho, kam byly zaměřené, se pravděpodobně používaly jen k místní komunikaci. Jeho pohled upoutala vyčnívající kovová věc za královským palácem, ale ze své současné pozice tam neviděl. Rozhodl se počkat do noci a pak se přes střechy jednotlivých staveb opatrně přesunout blíže paláci.

Čekání na tmu využil k tomu, že si pro každou oblast města našel nejvyšší stavbu, kterou by při nočním přesunu mohl použít jako pozorovatelnu.

26-5-6 Nejvyšší stavby — 12 bodů

Chceme v mimozemském městě najít vždy lokálně nejvyšší stavbu. Plán mimozemského města je tvořen čtvercovou sítí o rozměrech $R \times S$, kde pro každé políčko známe výšku stavby, která se na něm nalézá. Dále máme zadaná čísla r a s (platí $r \leq R$ a $s \leq S$) – velikost oblasti, která nás zajímá.

Chtěli bychom pro každou oblast velikosti $r \times s$ zjistit, jakou nejvyšší stavbu obsahuje.

Výstupem programu by tedy měla být tabulka velikosti $(R-r+1)\times(S-s+1)$, kde políčko $[i, j]$ bude obsahovat maximální výšku stavby nalézající se v oblasti velikosti $r \times s$ s levým vrchním rohem v tomto políčku.

Ukázkový vstup:

```
r = 3, s = 3
1 4 2 2 0 2
0 2 2 2 1 1
2 1 3 3 0 1
8 1 0 5 0 6
```

Ukázkový výstup:

```
4 4 3 3
8 5 5 6
```

⇧ **Lehčí varianta (za 6 bodů):** Vyřešte úlohu jen pro jednorozměrný případ (tedy $R = r = 1$).

Padla noc. Jacob se plížil po střechách jen v černé kombinéze, pomáhaje si nočním viděním. Díky tomu, že si přesně zmapoval nejvyšší budovy ve městě, se dokázal snadno vyhýbat hlídkám. Metr po metru se blížil ke královskému paláci.

Seskočil z poslední střechy, obešel věžičku a naskytl se mu pohled na něco podivného. Samozřejmě tu věc poznával, za svoji kariéru ji viděl mnohokrát na spoustě hvězdných lodí. Nebo alespoň její modernější sourozence, tohle byl vážně starý model, ten už se skoro půlstoletí nepoužíval.

Co ho ale zaskočilo mnohem víc než shledání se starým záchranným raketoplánem Mark II, bylo to, že aktuálně vypadal jako vyvržený vorvaň. Jeho motory to asi odnesly při tvrdém přistání, aspoň podle škod na zádi. Plátování z celého okolí fúzního reaktoru bylo odstraněno a od odhalených součástek vedlo směrem do paláce několik tlustých svazků kabelů.

Další vedly z místa bývalého kokpitu a končily u soustavy antén na střeše vlevo od něj. Působilo to celé, jako kdyby se někdo rozhodl využít z raketoplánu každou použitelnou věc, začal ho přestavovat na obytný přívěs křížený s továrnou na boty a v půlce navíc ztratil výrobní plány.

Náhle ho vyrušil šoupavý zvuk po jeho pravici. Rychle vyskočil, otočil se a ztuhnul s rukou na šipkové pistoli. Z půl metru hleděl do zrcadlového hledí skafandru. Kriticky si uvědomil, že ho ten člověk má špatně nasazený, zámky helmy nebyly zacvaklé a navíc na sobě neměl rukavice. To mu však nijak nezabránilo v tom praštit Jacoba po hlavě kovovou tyčí.

Probudil se v temné místnosti a příšerně ho bolela hlava. Zahlédl vedle sebe nějakého člověka. Poprvé po pěti letech zase spatřil příslušníka lidské rasy!

„Omlouvám se za tu ránu na hlavě, můj syn je trochu zbrklý," začal člověk starým hlasem. „To víte, nespatřil nikdy nikoho jiného mimo nás, narodil se pět let po ztroskotání."

„Jak jste tu dlouho?" zeptal se Jacob a mnul si bouli na hlavě.

„Naše loď, Hermes*, tu ztroskotala před asi 30 pozemskými roky, pokud počítám správně. Já jsem Cedric, nejvyšší přeživší důstojník," řekl stařec, „kapitán a většina ostatních důstojníků zemřeli, když naváděli loď někam do nejhlubšího moře... selhával nám reaktor a oni nechtěli zamořit místní ekosystém výbuchem."*

Hermes*, to mu něco říkalo. Nebyla to ta civilní loď, která se ztratila během války? Původně se myslelo, že byla v nějaké potyčce omylem zničena, ale průzkum záznamů všech zúčastněných stran neukázal nic.*

„A od té doby tu využíváte domorodce?" přešel Jacob hned k jádru věci.

„Oni si myslí, že jsme bohové. A já nemám v úmyslu jim to vyvracet. Sem tam se sice objeví nějaký pochybovač, ale jak se říká, dostatečně pokročilou technologii nelze odlišit od magie – ty blázni se daj hrozně snadno přesvědčit. A když to nejde..." s těmi slovy si nadhodil v ruce původně Jacobovu šipkovou pistoli.

„Ty jsi tu spadl s tou velkou lodí před pár lety, ne? Pokoušeli jsme se do ní dostat, ale s našimi nástroji ze záchranného člunu jsme se nedokázali proříznout ani přes jednu přepážku. Ty ale určitě máš přístupové kódy, že?"

„Předpokládáte, že vám pomůžu?!?" zeptal se Jacob.

„Ty se nechceš nechat považovat těmihle tupci za boha?" podíval se Cedric na Jacobův znechucený výraz. „No dobrá, pár dnů na přemýšlení možná změní tvůj názor."

S těmito slovy odešel a nechal Jacoba samotného v cele. S vyhlídkou na dlouhý pobyt začal Jacob zkoumat své vězení. Na jedné zdi objevil dlouhý zápis jedné z oblíbených mimozemských her, asi si tu nějaký vězeň taky krátil dlouhé čekání. Hra se docela podobala pozemským piškvorkám a Jacoba by zajímalo, jak vlastně dopadla.

26-5-7 Partie piškvorek — 13 bodů

Jacob našel zápis jedné partie mimozemských piškvorek, která se hraje se na čtvercové síti o rozměrech $N \times N$. Zápis hry je tvořen třemi typy tahů:

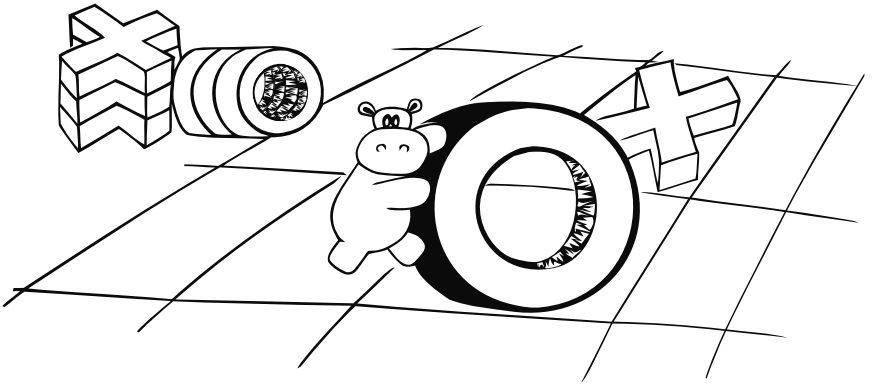
- Na políčko $[A, B]$ umísti kolečko
- Na políčko $[A, B]$ umísti křížek
- Vymaž znak z políčka $[A, B]$

Po každém tahu by nás zajímalo, jak je dlouhá aktuálně nejdelší souvislá řada symbolů (v řádce, sloupci nebo na úhlopříčce). Vybudujte datovou struktu-

ru držící aktuální stav hrací desky, která by navíc měla zvládat rychle zapisovat jednotlivé tahy a po každém tahu rychle vypsat aktuálně nejdelší řadu.

Předpokládejte, že tahů bude řádově stejně, jako je velikost hrací desky (tedy řádově N^2). Tahy budou vždy korektní, tedy nebude docházet k mazání prázdného políčka ani k umisťování znaku na neprázdné políčko.

⇧ **Lehčí varianta (za 5 bodů):** Vyřešte úlohu pro partii, v níž se znaky nebudou mazat (nastanou jen tahy prvních dvou typů).



Po zamotání se v partii piškvorek Jacob ulehl na tvrdou zem – ráno moudřejší večera, pomyslel si. Uprostřed noci ho ale probudil slabý hlas. Chvíli přemýšlel, jestli se mu to nezdálo, ale když Adu zaslechl znova, okamžitě vyskočil. Dívala se na něj skrz malé okénko a podávala mu jeho plný batoh.

Bez rozmýšlení ho popadl, vyndal z něj plazmovou karabinu, připjal si meč k boku a řekl Adě: „Radši ustup.“ Pozvedl zbraň, zacílil a několika přesnými výstřely zničil mříž. Pak se vzniklým otvorem protáhl k Adě. „Jak. . . “ začal, ale Ada ho umlčela. Ukázala mu rukou a rozeběhla se do temnoty.

Nedostali se ale daleko. Na nádvoří, kus od raketoplánu, je obklíčili strážci s ostrými oštěpy. „Já je nechci zranit, Ado,“ sykl, „není tu nějaká jiná cesta?“

„Máš meč? Vytáhni ho, dělej!“ řekla rychle Ada.

Jacob si spustil karabinu na popruhu dolů a tasil meč. Co se stalo dál, nechápal. Strážní upřeli pohled na blyštivou čepel zdobenou kameny, začali si něco mumlat a ustupovali. Zaslechl něco o proroctví a prorokovi. Co ho ale udivilo víc, bylo to, že jeden z kamenů na meči začal v blízkosti fúzního reaktoru raketoplánu nezvykle pulzovat. Vrhl rychlý pohled do změti kabelů a pak mu to došlo.

„Ado, chyť ten meč a drž je od nás dál!“

Ada opatrně uchopila meč, jako by se bála, že se o něj spálí. Jacob začal kuchat obnažené zařízení, než se mu povedlo uvolnit vlnový rezonátor montovaný na tyto staré reaktory. Okamžitě se spustil nouzový protokol a reaktor se odstavil, palác náhle potemněl.

Ada s Jacobem využili nastalý zmatek a rychle unikli z města. Tam na ně čekalo pár dalších členů Bratrstva s tvory vzdáleně podobnými koňům. Cesta nazpět k Freye *jim nezabrala ani tři dny.*

Během zastávek vyrazil Jacob z meče, ke zděšení ostatních, několik kamenů a něco s nimi a s ukradeným rezonátorem dělal. Co, to nechtěl říct, ale naléhal, že jako první musí k vraku lodě. Také se cestou věnoval sepisování nějakých věcí do elektronického zápisníku.

Když tam dorazili, vylezl po trupu nahoru až k troskám anténního pole. Vůbec neočekával, že by se na téhle planetě mohlo vyskytovat thallium v krystalické podobě. Pokud tohle vyjde, mohl by přes něj pomocí toho prastarého rezonátoru vyslat krátký subprostorový impulz, který by mohl proniknout vším tím rušením okolo.

Potom, co všechno správně pospojoval, usedl k jednomu z řídících panelů. Přehrál do paměti lodního počítače připravenou zprávu z elektronického zápisníku. Zhluboka se nadechl a potvrdil příkaz. Lodní počítač ve zlomku sekundy přetížil anténní systém a v efektním záblesku spálil krystal na prach...

V souvislosti s obdrženým nouzovým voláním byla vyslána záchranná loď UFRS Backspace, které se povedlo vyzvednout jediného přeživšího z havárie nákladní lodi UFC Freya a zachránit cenný náklad.

Hlášení záchranného týmu je dostupné samostatně na stránce záchranné akce,[19] *Jacob však o své záchranně a o věcech ji předcházející také sepsal poslední část svého deníku.*

Boj

Jacob vzdychl, když zjistil, že v troskách výstrojního skladu ve vraku Freyi *už zbývá jen poslední pár bot jeho velikosti. Bylo to už skoro dvacet let, co tady ztroskotal, patnáct let, co už byl součástí Bratrstva a zvykal si na místní zvyky, ale stejně mu zatím nic nenahradilo kvalitní lidské pracovní boty.*

V přílet záchranné mise ale už přestal doufat před více jak pěti lety. Signál přece nemohl putovat na Zemi tak dlouho, pravděpodobnější bylo, že byl příliš slabý. Mezitím co si nazouval nové boty, přemýšlel, jak nejnovější plán může zvrátit poměr sil mezi Bratrstvem a královými lidmi.

Přímo s královými lidmi se Jacob nesetkal už patnáct let, od té chvíle, co utekl s pomocí Ady z králova paláce, ale bojů se mezitím pár odehrálo. Bratrstvo bylo příliš rozptýlené a příliš dobře skryté, než aby se ho král dokázal zbavit, králova armáda zase příliš velká. Byla to patová situace.

Najednou se v průchodu objevilo světlo pochodně a kus za ním Ada. Prý co mu trvá tak dlouho. Jacob se usmál, rozsvítil svoji svítilnu s novým energetickým článkem a vyšel za Adou ven. Před lodí je už čekala jejich malá skupinka elitních domorodců. Byl čas na další přepad.

[19] `http://ksp.mff.cuni.cz/sksp/2014/`

Rozsahem sice jejich část neměla být nijak velká, ale byla součástí velkého plánu připravovaného více jak rok. Pokud se jim v nejbližších pár dnech všechno povede, odříznou krále od všech tří ložisek železa a tím mu v dlouhodobé perspektivě zabrání kovat další zbraně pro svou armádu.

⋆⋆⋆

Po dvoudenním přesunu se dostali na své místo. Nyní Jacob s Adou hleděli z kopce dolů na tábor, ze kterého královi vojáci vyráželi na hlídky na stezky v místních kopcích. Ačkoliv nevelký, byl klíčem k celému vstupu do pohoří a k dolům v něm.

Jacob se stále držel svého přesvědčení, že jako pozemšťan nezabije žádného domorodce, a tak popadl svoji foukačku s uspávacími šipkami, vydal smluvené znamení a skupina vyrazila. Zdálky to mohlo pozornému divákovi připadat, jako kdyby se začaly přesouvat keře na svahu, tak dobře byli maskovaní. Ve chvíli, kdy dorazili až k táboru, přišla zrada, byla to past!

V táboře mělo být jen několik desítek králových lidí, namísto toho se ozvalo zatroubení na roh a z protějších kopců se spustil příval více než dvou stovek ozbrojenců. Nebyl čas přemýšlet, Ada zakřičela povel pro ústup. Skupina se pokoušela koordinovaně stáhnout zpátky na kopec, ale teď byl svah naopak jejich nepřítelem. Náhle odněkud přiletěl kámen a...

V zajetí

Jacob se s trhnutím probudil. Cítil ostrou bolest na hlavě, asi od zásahu kamenem. Rozhlédl se po temné místnosti a přemýšlel, co ho tak náhle probudilo. Jasné, byl ve vězení, asi padl do zajetí při útoku na tábor, ale stejně měl neodbytný pocit, že ho probudilo ještě něco jiného. A pak to uslyšel.

Zpočátku ten zvuk nepoznával, však ho také dvacet let neslyšel. Ale pak mu to došlo, takhle zní jen hypersonický vstup do atmosféry ve velké výšce. Dobelhal se k zamřížovanému oknu a pohlédl na oblohu. Chvíli nic neviděl, ale pak se na okraji zorného pole vyloupl jasný bod a rychle se zvětšoval.

Nějaká kosmická loď vstupovala do atmosféry a ten jasný bod, jak záhy zjistil, byl atomový oheň z ústí čtyř motorů obrácených na přistání. Teda, ten, kdo s tím klesal, se vůbec nepáral s nějakým pohodlím, tohle vypadalo spíše jako bojový sestup výsadkového člunu, který si pamatoval z armády.

Zvuk se zesiloval, než loď prolétla nad místem, kde byl, stále ve výšce přes deset kilometrů. Pak přelétla mimo místo, kam z okna viděl, a ztratila se mu. Jacob byl úplně u vytržení, že by to konečně byla záchranná výprava? Teď jen doufal, že se Ubuovi s Adou povede pozemšťanům nějak sdělit, kde je on.

Osvobození

Uplynulo několik dní, Jacob je ve svém stavu nebyl schopný ani spočítat, když tu se venku za zdmi jeho cely ozvaly zvuky boje. Netrvalo dlouho a dostaly se až za dveře, které náhle padly na zem. Za nimi byl nějaký pozemský voják,

s bojovým kladivem stál nad jeho stráží a křičel kamsi dozadu: „To byl poslední, už je tady čisto!“

Než se Jacob stihl vzpamatovat, už se do cely hnal nějaký člověk v pilotní uniformě a za ním další, jehož oslovovali kapitáne, a pak ještě někdo v černém plášti. Chvíli mu trvalo, než uvěřil, že se mu to nezdá, ale pak se Jacobovi úlevou roztřásla kolena.

V následující konverzaci, kdy si po dlouhé době vzpomínal zase na pozemský jazyk, si se zachránci vyměnil několik informací. Zachránce ale víc zajímala jeho přístupová karta na UFC Freyu. Bohužel, jak zjstil, asi mu ji sebrali králové lidé a odvezli do hlavního města.

Jak zjistil, zachránci přiletěli na lodi UFRS Backspace, kterou svým tvrdým přistáním také trochu poškodili, ale jinak byla loď funkční. Souhlasil tedy s tím, že výměnou za horkou sprchu jim pomůže se další den infiltrovat do králova města.

⋆⋆⋆

Najedený Jacob musel souhlasit, že pozemské jídlo mu již dlouho scházelo. Teď ale nebyl čas na nějaké lenošení, zrovna před hodinou vyrazila většina posádky směrem ke královu městu, aby se pokusila o jeho infiltraci a na lodi zůstalo jen několik důstojníků. I ti se měli brzy vypravit do města, avšak rychlejší cestou.

Bratrstvu se povedlo vybudovat v královském městě síť informátorů a s jejich pomocí se snad posádka dopátrá toho, jak se vloudit přímo do králova paláce, aniž by to místním přišlo divné. Na něm a na důstojnících teď bylo připravit noční akci na zajištění jejich ústupu.

Záchrana

Po probdělé noci a po odpoledním vytahování cenného nákladu z radioaktivních částí vraku Freyi *se Jacob vydal ještě naposledy do hlavního stanu Bratrstva. Pomalu přišel čas rozloučit se s místními, kteří přes patnáct let byli jeho rodinou. Další den již měla* UFRS Backspace *opět vzlétnout a Jacob nevěděl, kdy se sem bude moci opět vrátit, jestli vůbec.*

Záchranné týmy po vraku Freyi *rozmístily dezintegrační antihmotové nálože, stejné, jaké se v noci povedlo důstojníkům rozmístit v králově paláci po zbytku pozemské techniky. Federační předpisy zněly jasně, nesmí se zde nechat žádná funkční část pozemské techniky.*

Absolvoval dojemné rituální loučení v Bratrstvu a cestou k záchranné lodi se ještě naposledy zastavil u torza lodi, která mu před dlouhými lety svou nezdolností umožnila přežití. Chvíli nechal ruku položenou na ožehnutém a pokrouceném pancéřovém trupu a pak se se slzami v očích obrátil a již se neotočil. . .

Seriál – výpočetní modely

Martin „Medvěd“ Mareš, Jirka Setnička, Karolína „Karryana“ Burešová

26-1-8 Turingova strojovna — 12 bodů

Při řešení KSP po vás obvykle chceme, abyste na daný problém sestrojili *algoritmus.* Přemýšleli jste někdy nad tím, co to takový algoritmus přesně je? Intuitivně je to jasné: nějaký zápis postupu, jak něco spočítat, poskládaný z dostatečně jednoduchých kroků. To se ale sotva dá pokládat za pořádnou definici.

Tak jinak: algoritmus je totéž co program v našem oblíbeném programovacím jazyce, jen zbavený přízemních detailů (třeba omezené velikosti datových typů). Je to lepší? O moc ne – sice už je jasné, co to znamená, ale zase se nám definice rozrostla o specifikaci celého programovacího jazyka (umíte popsat Céčko nebo Python jedním odstavcem? stránkou? knížkou?). A navíc ani není jasné, jestli pro nás algoritmus znamená totéž jako pro dědečka Pascalistu nebo pro pradědečka, který ještě programy děroval ve strojovém kódu.

Dobrá, jaká je tedy správná definice? Teoretičtí informatikové obvykle postupují tak, že si zavedou nějaký *výpočetní model.* Tím se myslí jednoduchý matematický stroj s přesně určenými operacemi, řízený programem. Za algoritmus pak prohlásíme program pro tento stroj. Na první pohled to vypadá, že jsme se z deště přesunuli pod okap, ale přeci jen tu prší méně: Výpočetní modely jsou daleko jednodušší zvířátka než běžné programovací jazyky (však za chvíli uvidíme). Navíc není těžké o různých výpočetních modelech dokázat, že dovedou spočítat totéž, takže nezáleží na tom, který z nich jsme pro zavedení algoritmu použili.

V letošním seriálu se spolu vydáme na procházku po zoo výpočetních modelů. Výběhů tu je habaděj, my se zastavíme u pěti z nich a pokaždé si v daném modelu zkusíme něco naprogramovat a třeba i dokázat pár obecných větiček o jeho vlastnostech. Mezitím můžete sami rozmýšlet, jak do každého modelu překládat programy z vašeho oblíbeného jazyka. Pěknou cestu!

Turingovy stroje

Nejklasičtějším a nejspíš i nejstarším modelem počítače je bezpochyby Turingův stroj. Alan Turing ho popsal v roce 1936 ve své práci, kterou položil základy matematického zkoumání počítačů.

TS je vybaven *oboustranně nekonečnou páskou* rozdělenou na *políčka.* Na každém políčku je zapsán jeden *znak* ze zvolené *abecedy*. Tou se myslí nějaká množina znaků, o níž víme jen to, že je konečná a že obsahuje mezeru (tu značíme ␣).

Nad páskou se pohybuje *hlava.* Vždy se nachází nad jedním políčkem a umí z něj přečíst znak a případně ho přepsat na jiný.

Činnost stroje ovládá *řídicí jednotka* podle *programu.* Řídicí jednotka se v každém okamžiku nachází v jednom z konečně mnoha *stavů.* V každém kroku výpočtu se podívá, v jakém stavu S se nachází a jaký znak z vidí hlava, a podle toho vybere jednu *instrukci* programu. Ta stroji říká, že má znak z přepsat na z', posunout hlavu o políčko v daném směru a nakonec se přepnout do stavu S'.

Program tedy můžeme popsat tabulkou, jejíž řádky odpovídají možným stavům S, sloupce znakům z a v každé buňce tabulky je uložena jedna instrukce v podobě trojice (z', p, S'). Instrukce říká, co má stroj ve stavu S, který přečetl znak z, udělat. Tedy zapsat znak z', posunout se ve směru $p \in \{\leftarrow, \rightarrow, \bullet\}$ (o políčko doleva či doprava, případně zůstat na místě) a přejít do stavu S'.

Nyní definujeme *výpočet* stroje. Na počátku výpočtu je na pásce zapsán *vstup,* hlava stroje stojí na začátku vstupu a zbytek pásky je vyplněn mezerami. Řídicí jednotka se nachází ve zvoleném *počátečním stavu* S_0. Výpočet probíhá v *krocích* (taktech) – v jednom kroku stroj provede jednu instrukci programu (přečte znak, rozhodne se podle tabulky, zapíše znak, posune hlavu, změní stav). Tak pokračuje až do doby, kdy se dostane do některého z *koncových stavů.*

Koncové stavy budeme mít dva a budeme jim říkat ANO a NE, čímž umožníme stroji, aby výpočet ukončil úspěšně nebo neúspěšně. Pokud chceme, aby výsledkem výpočtu bylo něco víc než jediný bit, dohodneme se, že výstup bude opět napsán na pásce, obklopen mezerami.

Dodejme ještě, že vždy hledáme jeden TS, který danou úlohu vyřeší pro všechny vstupy – počet stavů, velikost abecedy nebo obsah programu nesmí záviset na vstupu. Pak můžeme snadno definovat časovou a prostorovou složitost.

Čas budeme měřit počtem provedených instrukcí, *prostor* počtem políček pásky, jež během výpočtu navštívila hlava stroje. Nezapomeňme, že mohou existovat i výpočty, které se nikdy nezastaví; ty pak mají nekonečnou časovou složitost a možná i nekonečnou prostorovou.

Příklad

Suchá formální definice bude stravitelnější, když ji doplníme příkladem. Sestrojíme TS, který dostane řetězec složený ze znaků `+` a `-`, vždy se zastaví a odpoví ANO právě tehdy, když je *vyvážený*. Tím myslíme, že obsahuje stejně plusek jako minusek.

Stroj bude na pásce opakovaně hledat dvojice znaků `+` a `-` (ne nutně vedle sebe) a oba znaky přepisovat na `*`. Vyvážený vstup tedy nutně předělá na samé hvězdičky. Jestliže vstup vyvážený není, zbude nakonec jedno `+`, ke kterému už neexistuje `-`, či opačně.

Za abecedu stroje zvolíme množinu $\{\sqcup, +, -, *\}$, řídicí jednotka bude vybavena stavy $\{S_0, P, M, R, \text{ANO}, \text{NE}\}$ a následujícím programem:

stav/znak	$\sqcup$	+	−	*
S_0	$(\sqcup, \bullet, \text{ANO})$	$(*, \rightarrow, P)$	$(*, \rightarrow, M)$	$(*, \rightarrow, S_0)$
P	$(\sqcup, \bullet, \text{NE})$	$(+, \rightarrow, P)$	$(*, \leftarrow, R)$	$(*, \rightarrow, P)$
M	$(\sqcup, \bullet, \text{NE})$	$(*, \leftarrow, R)$	$(-, \rightarrow, M)$	$(*, \rightarrow, M)$
R	$(\sqcup, \rightarrow, S_0)$	$(+, \leftarrow, R)$	$(-, \leftarrow, R)$	$(*, \leftarrow, R)$

Ve stavu S_0 hledáme první znak různý od *. Pokud je to +, přejdeme do stavu P, v němž hledáme − do páru. Podobně stav M odpovídá tomu, že jsme našli − a hledáme párové +. Po nalezení páru pokračujeme stavem R, který hlavu stroje vrátí zpět na začátek vstupu a pak přejde na hledání dalšího páru, tedy do stavu S_0.

Časová složitost tohoto stroje pro vstup délky n činí $\mathcal{O}(n^2)$, neboť na vstupu se vyskytuje až n párů znamének a každý z nich můžeme hledat až $\mathcal{O}(n)$ kroků. Paměti spotřebuje $\mathcal{O}(n)$ políček.

Úkol 1 [3b]: Navrhněte Turingův stroj, který na posloupnost závorek (a) odpoví ANO nebo NE podle toho, zda je vstup správně uzávorkovaný. Tím myslíme, že závorky jsou správně spárované a páry se nekříží. Například na vstupy ()() a (()()) odpoví ANO a na)()(odpoví NE.

Součástí řešení úkolu by měl být kompletní popis stroje: abeceda, množina stavů, program. Sluší se též spočítat, jakou má stroj časovou a paměťovou složitost.

Vícepáskové stroje

Možná vás překvapilo, že stroj z předchozího příkladu potřeboval na tak obyčejnou věc, jako rozpoznání vyváženosti, kvadratický čas. Zčásti to bylo způsobené naší nešikovností (zkuste sestrojit jiný stroj, který tutéž úlohu zvládne rychleji), zčásti tím, že musíme neustále přejíždět hlavou mezi různými místy, která nás zajímají současně.

Často se proto uvažuje vícepásková varianta Turingova stroje. Ta má libovolný počet pásek (budeme ho značit p) a na každé pásce samostatnou hlavu. Řídicí jednotka se pak rozhoduje podle kombinace symbolů viděných jednotlivými hlavami. Program proto není 2-rozměrná tabulka, nýbrž $(p+1)$-rozměrná (jeden rozměr odpovídá aktuálnímu stavu stroje, ostatní přečteným znakům). Instrukce programu pak říkají každé hlavě, jaký symbol má na svou pásku zapsat a kterým směrem se má posunout; různé hlavy se mohou pohybovat různě, nebo zůstat stát.

U vícepáskových TS bývá zvykem, že jedna z pásek je vyhrazena pro vstup a jiná pro výstup. Na vstupní pásce je na počátku vstup, stejně jako u jednopáskového TS. Ostatní pásky ze začátku obsahují samé mezery. V průběhu výpočtu smí program ze vstupní pásky pouze číst a na výstupní pouze zapisovat; ostatní

pásky (těm se říká *pracovní*) může využívat libovolně. Do prostorové složitosti se pak počítají pouze políčka na pracovních páskách.

Příklad podruhé

Předveďme si, jak úlohu s plusky a minusky vyřešit rychleji za pomoci stroje se dvěma páskami: jednou vstupní a jednou pracovní (jelikož odpovídáme pouze ANO/NE, výstupní pásku nepotřebujeme).

Půjde to snadno: nejprve projdeme vstup a všechna − zkopírujeme na pracovní pásku. Poté vstup projdeme podruhé a za každé + smažeme jedno − z pracovní pásky; pokud už tam žádné není, odpovíme NE. Na konci ověříme, zda je pracovní páska prázdná, a podle toho odpovíme ANO nebo NE.

Stroj pracuje s abecedou $\{+, -, \sqcup\}$ a stavy $\{S_0, R, \text{ANO}, \text{NE}\}$. Jeho program vypadá následovně:

$$
\begin{aligned}
(S_0, +, \alpha) &\to ((+, \rightarrow), (\alpha, \bullet), S_0)\\
(S_0, -, \alpha) &\to ((-, \rightarrow), (-, \rightarrow), S_0)\\
(S_0, \sqcup, \alpha) &\to ((\sqcup, \leftarrow), (\alpha, \bullet), R)
\end{aligned}
$$

$$
\begin{aligned}
(R, +, -) &\to ((+, \leftarrow), (\sqcup, \leftarrow), R)\\
(R, +, \sqcup) &\to \text{NE}\\
(R, -, \alpha) &\to ((-, \leftarrow), (\alpha, \bullet), R)\\
(R, \sqcup, \sqcup) &\to \text{ANO}\\
(R, \sqcup, -) &\to \text{NE}
\end{aligned}
$$

(zbývající kombinace znaků na páskách nenastanou)

Jelikož trojrozměrnou tabulku neumíme nakreslit, popsali jsme ji pomocí pravidel $(S, x, y) \to ((x', p), (y', q), S')$. Tím myslíme: jsme-li ve stavu S a vidíme-li na vstupní pásce znak x a na pracovní znak y, pak na vstupní pásku zapíšeme x' a provedeme posun p, na pracovní pásku zapíšeme y' a provedeme posun q; nakonec přejdeme do stavu S'. Symbol α značí libovolný znak abecedy, na levé straně pravidla stejný jako na pravé. Jelikož na vstupní pásku není povoleno zapisovat, tváříme se, jako bychom tam vždy zapisovali to, co tam už je. Pokud zastavujeme výpočet, píšeme prostě ANO či NE a nestaráme se, co stroj udělá s páskami.

Náš nový stroj pracuje v lineárním čase (vstup projde všeho všudy dvakrát) a spotřebuje lineární množství prostoru.

Úkol 2 [5b]: Vyřešte první úkol na vícepáskovém Turingově stroji. Snažte se dosáhnout co nejlepší časové i paměťové složitosti.

Úkol 3 [2b]: Dokažte, že každý Turingův stroj jde upravit, aby si vystačil pouze se dvěma stavy (kromě ANO a NE). Počítat musí samozřejmě stále totéž, byť třeba pomaleji.

Úkol 4 [2b]: Ukažte, jak vyřešit předchozí úkol pro jednopáskové stroje tak, aby zůstaly jednopáskovými.

Poznámky

S Turingovými stroji se řešitelé KSP potkali už jednou: zabýval se jimi seriál 14. ročníku. Zkuste se na jeho zadání i řešení podívat, skrývá se tam nejedna další zajímavost. Letošní úlohy jsou ale samozřejmě řešitelné i bez toho.

Zkoušeli jsme zesílit TS přidáním dalších pásek. Co kdybychom ho naopak chtěli trochu oslabit? Nabízí se nahradit přepisovatelnou pásku „děrnou páskou" – v abecedě existuje speciální znak „díra" a stroj má povoleno pouze přepsat mezeru na libovolný znak a nemezerový znak na díru. V úloze 14-4-5 se dokazuje, že takový TS je stejně silný jako ten klasický.

KSP

V omezování TS bychom mohli ještě pokračovat: mohli bychom úplně zakázat zápis, takže stroj by směl jenom pohybovat se po pásce a číst (jinými slovy měl by jenom vstupní pásku a žádné pracovní). Takový stroj už je mnohem slabší, konkrétně ekvivalentní s konečnými automaty, které jste mohli potkat v seriálu 23. ročníku.

seriál

26-2-8 Továrna na přepisování — 14 bodů

V prvním díle letošního seriálu jste se mohli seznámit s Turingovými stroji coby zajímavým teoretickým modelem počítače. Dnes v naší zoo výpočetních modelů popojdeme k sousednímu výběhu, kde se pasou *přepisovací pravidla.* Ne nadarmo jsou hned vedle – časem uvidíme, že si spolu tahle dvě zvířátka náramně rozumějí.

Abeceda a vstup s výstupem

Jak už název napovídá, přepisovací pravidla pracují nad nějakým zadaným řetězcem znaků (říkejme mu *vstup*) a postupně ho nějak přepisují, až dostanou nějaký další řetězec znaků (*výstup*). Než začneme mluvit o samotných pravidlech, pojďme tyto řetězce pořádně definovat.

Za *abecedu* budeme označovat konečnou množinu všech *znaků,* které se mohou vyskytnout v řetězci. *Vstup* a *výstup* jsou pak vždy nějaké konečné řetězce znaků z této abecedy.

Mimo to zavedeme dva metaznaky, kterými budeme označovat okraje řetězce: `^` na začátku a `$` na konci řetězce. Tyto znaky nebudou součástí abecedy, a tedy se nebudou vyskytovat nikde jinde než právě na okrajích řetězce.[20]

Abecedou mohou být například čísla `0-9`, písmena `a-z`, nebo třeba symboly {♡, ♠, ♣, ♢}. Jednotlivé úlohy pak mohou navíc určit, že ve vstupu nebo výstupu se smí vyskytovat jen některé znaky abecedy.

[20] Podobnost s regulárními výrazy používanými v praxi vůbec není náhodná. Chcete vědět víc? Nahlédněte do seriálu 23. ročníku.

Přepisovací pravidla

Každé *přepisovací pravidlo* má svou levou a pravou část: levé části budeme říkat *vzor* a pravé *přepis*. Zapisovat ho budeme takto:

$$vzor \rightarrow přepis$$

Ve vzoru se mohou kromě znaků abecedy vyskytovat i metaznaky začátku a konce řetězce (pak se vzor váže k nějakému z okrajů řetězce), v přepisu se už mohou vyskytovat jen znaky abecedy.

Aplikace konkrétního pravidla na nějaký řetězec pak vypadá následovně:

- Najde se první (nejlevější) výskyt vzoru v řetězci.
- Tento výskyt se vymaže a na jeho místo se vloží přepis.

Pravá strana (přepis) může být i prázdná. Takové pravidlo lze například využít k vymazání části vstupu. Vzor naopak prázdný být nemůže, vždy musí obsahovat alespoň jeden znak nebo metaznak.

Uvažme například následující pravidla:

$$\texttt{a} \rightarrow \texttt{b}$$
$$\texttt{a\$} \rightarrow \texttt{b}$$
$$\texttt{\^{}aa} \rightarrow \texttt{ccc}$$

Každé samostatně aplikujeme na vstup `baa`. První pravidlo ho přepíše na `bba`, druhé na `bab` a třetí pravidlo nic nepřepíše (takovýto vzor ve vstupu neexistuje, a tedy se nic neprovede).

Samostatnou aplikací pravidel ovšem dost silný výpočetní model nedostaneme. Na to bude potřeba poskládat více pravidel dohromady.

Přepisovací programy

Přepisovacím programem nazveme nějaký uspořádaný soubor přepisovacích pravidel, čili několik pravidel seřazených za sebe.

Výpočet přepisovacího programu probíhá v *krocích*: první krok přepisuje vstup programu, výstup prvního kroku se stane vstupem druhého, a tak stále dokola. Výpočet končí ve chvíli, kdy již mezi pravidly neexistuje žádné, jehož vzor by se vyskytoval ve vstupu. To se samozřejmě nemusí stát – jistě snadno vymyslíte přepisovací program, který se nikdy nezastaví.[21]

Výpočet každého kroku by se dal formálně popsat takto:

- Najde se první pravidlo, jehož vzor se vyskytuje někde ve vstupu.
- Toto pravidlo se provede (najde se první výskyt vzoru ve vstupu a na jeho místo se vloží přepis).
- Výstup tohoto pravidla se použije jako vstup dalšího kroku. Výpočet dalšího kroku začíná opět od prvního pravidla a od začátku řetězce.

[21] Jednou z možností je třeba program skládající se z jediného pravidla $ → a. Ten nikdy neskončí, protože vzor pravidla je obsažen v každém řetězci.

Zbývá zavést nějakou míru efektivity – analogii časové složitosti normálních programů. Nám pro tyto účely bude sloužit počet provedených kroků přepisovacího programu. (Na zamyšlení: co by odpovídalo prostorové složitosti?)

Příklady

Pojďme vyzkoušet, co přepisovací programy dovedou.

První příklad bude jednoduchý. Představte si, že máme zadanou posloupnost nul a jedniček (třeba `01101`) a chceme ji setřídit. Jinými slovy chceme přesunout všechny nuly nalevo od jedniček.

To půjde překvapivě snadno. Sestrojíme program obsahující jediné pravidlo:

$$\texttt{10} \to \texttt{01}$$

Na našem ukázkovém vstupu bude výpočet probíhat takto: `01101`, `01011`, `00111`.

Povšimněme si, že dokud posloupnost není setříděná, existuje v ní aspoň jedna po sobě jdoucí dvojice `10`, takže program běží dál.

Doběhne někdy? Sledujme *inverze* v posloupnosti: tak budeme říkat dvojicím (ne nutně sousedních) čísel, která jsou ve špatném pořadí. Naše ukázková posloupnost obsahuje 2 takové dvojice. Každé použití pravidla sníží počet inverzí právě o jedna, takže se program po konečně mnoha krocích musí zastavit.

Jak dlouho náš program poběží? Ve vstupu délky n může být nejvýše $(n/2)^2$ inverzí (to odpovídá situaci, kdy na vstupu máme $n/2$ jedniček a za nimi $n/2$ nul) a program je odstraňuje po jedné, takže provede řádově n^2 kroků. (Mimochodem, pokud budeme provádět chytřejší operace než pouhé prohazování, je možné dosáhnout i lepší efektivity. Zkuste vymyslet, jak.)

Druhý příklad už bude záludnější. Možná si vzpomínáte na kontrolu správnosti uzávorkování z minulé série. Pojďme si ukázat, jak snadno se dá vyřešit pomocí přepisovacích pravidel.

Na vstupu budeme mít posloupnost levých a pravých závorek a naším úkolem bude rozhodnout, jestli tvoří správné uzávorkování, nebo netvoří. Podle toho vrátíme na výstupu buď `ANO`, nebo `NE`. Pro připomenutí, správné uzávorkování je takové, ve kterém jsou závorky správně spárované a páry se nekříží.

Idea našeho přepisovacího programu bude takováto: Všimneme si, že se v každé neprázdné správně uzávorkované posloupnosti vyskytuje po sobě jdoucí dvojice `()`. Tu můžeme odstranit, čímž získáme další správně uzávorkovanou posloupnost, a tak dále, až nakonec dospějeme k prázdné posloupnosti. Funguje to i naopak: přidáním `()` do jakéhokoliv správného uzávorkování nelze získat nesprávné, takže se nemůže stát, že bychom na prázdný řetězec zredukovali nějaké nesprávné uzávorkování.

Program vypadá následovně:

```
X$ → NE
X( → X
X) → X
() →
^$ → ANO
^( → X
^) → X
```


Dokud vše probíhá tak, jak má, čtvrté pravidlo umazává závorky. Pokud umaže všechny závorky, páté pravidlo vypíše `ANO`. Pokud ale již nebude možné umazat žádnou dvojici závorek, nastoupí jedno z posledních dvou pravidel a na scénu přijde `X`. To pak za pomoci prvních tří pravidel vymaže zbytek vstupu, vypíše `NE` a program se zastaví.

Rozmyslete si, proč namísto posledních dvou pravidel nelze použít `^` → `X`.[22]

Úlohy

Vymyslete přepisovací programy na následující problémy. U všech programů odhadněte efektivitu (v počtu provedených kroků) a pokuste se o co nejlepší. Součástí řešení by měl být slovní popis a alespoň náznak zápisu použitých přepisovacích pravidel.

Úkol 1 [2b]: Vstup je tvořený posloupností čísel `0` až `9`. Vaším úkolem je zanechat na výstupu `ANO`, pokud je posloupnost čísel neklesající, a `NE` v opačném případě.

Úkol 2 [3b]: Na vstupu je posloupnost znaků `*`. Spočítejte, kolik jich je, a výsledek zanechte na výstupu jako číslo ve dvojkové soustavě.

Úkol 3 [5b]: Na vstupu jsou dvě čísla ve dvojkové soustavě oddělená křížkem `#`. Na výstupu nechť se objeví to větší z nich. Pozor, zápisy čísel nemusí být stejně dlouhé. Případ, kdy jsou si čísla rovna, nemusíte uvažovat.

Převod na Turingův stroj a naopak

Jak už jsme zmínili v úvodu, přepisovací pravidla a Turingovy stroje spolu úzce souvisí. Ukážeme, jak převést jakýkoli přepisovací program na Turingův stroj.

Nejdříve se zamyslíme, jak převést jedno přepisovací pravidlo do řeči Turingova stroje. Začneme s hlavou na levém konci pásky a budeme se postupně pokoušet najít výskyt vzoru (vyzkoušíme všechny začátky a vždy porovnáme se vzorem; stav stroje bude říkat, kolikátý znak vzoru porovnáváme).

Ve chvíli, kdy nalezneme první výskyt, přepneme se do dalšího stavu a pásku přepíšeme odpovídajícím přepisem (a případně odsuneme či přisuneme zbytek

[22] Správná odpověď je, že pak by se nám program nikdy nezastavil, protože vzor takového pravidla by byl nalezen vždy.

znaků na pásce, pokud přepis bude mít jinou délku než vzor). A nakonec, po dokončení přepisování, vrátíme hlavu opět na začátek pásky (a stejně tak v případě, že se nám nepovede najít žádný výskyt vzoru).

Tím jsme úspěšně naučili Turingův stroj zpracovat jedno pravidlo. K překladu celého přepisovacího programu už zbývá jen krůček. Každé pravidlo v programu bude mít svou vlastní sadu stavů (sady nechť jsou třeba očíslované). Po úspěšném provedení pravidla a návratu hlavy zpět na začátek pásky se přesuneme do první sady; po neúspěšném hledání vzoru se přesuneme do následující sady (zkusíme aplikovat další pravidlo v pořadí). Pokud budeme neúspěšní i u posledního pravidla, ukončíme výpočet.

Právě jsme dokázali, že každý přepisovací program lze simulovat Turingovým strojem. Pokud navíc ukážeme, že každý Turingův stroj lze převést na přepisovací program, bude jasné, že oba výpočetní modely jsou stejně silné (co jde spočítat v jednom, jde i ve druhém a opačně). To už ale bude na vás:

Úkol 4 [4b]: Dokažte, že pro každý jednopáskový Turingův stroj existuje přepisovací program, který počítá totéž. Přesněji řečeno, pokud se pro daný vstup Turingův stroj zastaví, přepisování se také zastaví a vydá stejný výsledek. Pokud se stroj nezastaví, přepisování se také nezastaví. Rozmyslete si, v jakém vztahu je časová složitost stroje a pravidel. Svůj přístup demonstrujte na stroji z příkladu v 1. sérii (vyvážená posloupnost).

26-3-8 Zdivočelá počítadla — 15 bodů

Seriál o výpočetních modelech pokračuje, tentokrát ve znamení minimalismu. V tomto dílu si ukážeme jeden z vůbec nejjednodušších teoretických strojů. Takové obyčejné kuličkové počítadlo, jen trochu programovatelné. Proto mu budeme říkat *počítadlový stroj.*

Definice stroje

Počítadlový stroj pracuje výhradně s přirozenými čísly. Ta mohou být libovolně velká a zahrnujeme mezi ně i nulu. Obvykle jim budeme říkat prostě *čísla.*

Stroj je vybaven libovolným konečným počtem *registrů.* Registry jsou očíslovány a každý z nich obsahuje jedno číslo.

Program stroje je tvořen konečnou posloupností *instrukcí.* Těch jsou k dispozici 3 druhy:

- `INC` x (increment) – zvýší registr x o jedna
- `DEC` x (decrement) – sníží registr x o jedna; pokud už byl nulový, nic se nestane.
- `JNZ` x, p (jump if non-zero) – pokud je hodnota v registru x nenulová, skočí na p-tou instrukci programu; pokud je hodnota nulová, nic se nestane.

Na počátku výpočtu je ve smluvených registrech vstup a ostatní registry obsahují nuly. Instrukce se vykonávají jedna po druhé, počínaje první instrukcí programu. Pouze instrukce skoku může způsobit, že se místo následující instrukce začne provádět nějaká jiná, od níž program opět pokračuje sekvenčně.

Pokud se ocitneme mimo program (ať už tím, že jsme tam skočili, nebo po provedení poslední instrukce programu), stroj se zastaví a ve smluvených registrech najdeme výstup.

Časovou složitost bychom mohli zavést jako počet provedených instrukcí, paměťovou třeba jako velikost čísel, která se během výpočtu vyskytnou v registrech. U úloh v této sérii se nicméně nebudeme počítání složitosti věnovat, bude nás zajímat pouze počet použitých registrů. Ten se jako míra složitosti nechová, protože nezávisí na vstupu – je to spíš míra komplikovanosti programu.

Rozšíření instrukční sady

Instrukční sada našeho stroje je značně spartánská. Proto ji rovnou rozšíříme o několik zkratek pro běžné programátorské obraty.

- Zkratka `CLR` x (clear) bude sloužit k vynulování registru x:

```
A:  DEC x
    JNZ x, A
```

  Druhý parametr instrukce `JNZ` by měl udávat pořadové číslo neboli adresu instrukce `DEC` v programu. Abychom si nemuseli adresy pamatovat, instrukci `DEC` si pojmenujeme *návěštím* `A` a kdekoliv se na její adresu potřebujeme odkázat, uvedeme místo ní návěští. Tuto konvenci budeme používat u všech instrukcí skoku.

- `JMP` p (jump) bude značit nepodmíněný skok na instrukci s adresou p. Jak si ho pořídit? Můžeme si například obstarat nějaký registr n a na začátku programu instrukcí `INC` n zařídit, aby zaručeně nebyl nulový. Kdykoliv pak použijeme `JNZ` n, p, stroj vždy skočí na adresu p.

  Pokud vám přijde, že plýtváme registry, máte pravdu. Proto si raději místo nového registru n „vypůjčíme“ nějaký registr x, který už se v programu používá. Vždy ho na chvíli inkrementujeme, aby byl nenulový, a po použití zase vrátíme do původního stavu.

  Pokud by program vypadal takto:

```
    (nějaké instrukce)
A:  (další instrukce)
    JMP A
```

  mohli bychom ho přeložit na:

```
    (nějaké instrukce)
    INC x
A:  DEC x
    (další instrukce)
    INC x
    JNZ x,A
```

- `JZ` x, p (jump if zero) – opak instrukce `JNZ`, tedy skok, pokud je registr x nulový:

```
    JNZ x,Q
    JMP p
Q:
```

- `MOV` a, b (move) – zkopírování hodnoty z registru a do registru b. Pořídíme si pracovní registr t a provedeme:

```
    CLR b
    JZ  a,Z
    CLR t
X:  DEC a
    INC b
    INC t
    JNZ a,X
Y:  DEC t
    INC a
    JNZ t,Y
Z:
```

V prvním cyklu (návěští `X`) snižujeme a po jedné a zvyšujeme b. Tak přesuneme hodnotu z a do b, ale a si zničíme. Proto kromě b zvyšujeme i t a v druhém cyklu (`Y`) přelijeme t zpět do a. Tato konstrukce ovšem nefunguje, pokud $a = 0$, což vyřešíme výjimkou (`JZ` na počátku programu).

Úkol 1 [3b]: Navrhněte následující zkratky:

- `ADD` x, y, z (add) – sečte obsah registrů x a y a výsledek uloží do registru z.
- `SUB` x, y, z (subtract) – odečte od obsahu registru x obsah registru y a výsledek uloží do registru z. Pokud by mělo vyjít záporné číslo, uloží nulu.
- `MUL` x, y, z (multiply) – vynásobí obsah registrů x a y a výsledek uloží do registru z.

Zase ty závorky

Nemůžeme opomenout naši tradiční úlohu o závorkování. Potřebujeme však vymyslet, jak řetězec závorek popsat číslem. Zakódujeme ho velice jednoduše: každou levou závorku v řetězci přepíšeme na číslici 1, každou pravou na 2 a výsledek přečteme jako číslo v desítkové soustavě.

Například řetězec `()(())` přeložíme na číslo `121122`, prázdný řetězec zakódujeme jako nulu.

Úkol 2 [4b]: Napište program pro počítadlový stroj, který v registru x dostane kód posloupnosti závorek a až doběhne, sdělí v registru y, zda posloupnost byla ($y = 1$) nebo nebyla ($y = 0$) správně uzávorkovaná.

V programu můžete používat všechny zkratky, které jsme definovali, nebo které jste vymysleli v předchozím úkolu.

O síle počítadel

Jak je vidět z předchozího příkladu, pomocí počítadlového stroje lze odpovídat na netriviální otázky. Ukážeme, že je dokonce stejně silný jako Turingův stroj (a tím pádem i jako přepisovací programy z předchozí série).

Simulace počítadel na Turingově stroji je triviální – stačí každému počítadlu vyhradit jednu pásku stroje.

Úkol 3 [3b]: Vymyslete opačný převod: popište, jak k libovolnému Turingovu stroji sestrojit počítadlový stroj, který spočítá totéž. Zvolte vhodné kódování vstupu a výstupu Turingova stroje čísly.

Nabízí se také otázka, kolik počítadel doopravdy potřebujeme. U Turingova stroje víme, že si (za cenu zpomalení výpočtu) vystačí s jedinou páskou. Jak je to zde? Stačí pevný počet počítadel? A potřebujeme vůbec tolik instrukcí?

Úkol 4 [3b]: Pro co nejmenší konstantu k dokažte, že ke každému počítadlovému stroji existuje ekvivalentní stroj, který použije jen k registrů. Můžete předpokládat, že původní stroj pro vstup i výstup využívá jediný registr x. Dokázali byste k ještě snížit, pokud byste mohli zavést nějaké vlastní kódování vstupu?

Úkol 5 [2b]: Navrhněte ještě menší instrukční sadu, pomocí které půjdou vyjádřit všechny tři instrukce našeho počítadlového stroje. (Nová sada nemusí nutně být podmnožinou té původní.)

26-4-8 Dlaždičky — 16 bodů

V naší zoo výpočetních modelů jsme zatím potkávali volně pasoucí se exemplář. Dnes uvidíme první zdi. Není to ovšem proto, že by náš model potřeboval chránit před větry dešti, nýbrž proto, že tyto zdi bude náš program obkládat dlaždicemi.

Pojďme si nejprve říct, co je to taková dlaždice. *Dlaždice* představuje čtverec jednotkové velikosti, který má každou hranu obarvenou nějakou barvou. Obarvení jednotlivých hran může a nemusí být stejné, ale každá hrana musí být obarvena právě jednou barvou. Dohodněme se také, že barvy budeme označovat nějakými symboly, typicky čísly a písmeny.

Často budeme pro názornost dlaždice zobrazovat opravdu jako čtverce rozdělené na čtyři části, ale formálně dlaždici zavedeme jako uspořádanou čtveřici (ℓ, h, p, d), kde jednotlivé symboly ℓ, h, p, d označují po řadě obarvení levé, horní, pravé a dolní hrany dlaždice.

Z takových dlaždic můžeme skládat *dláždění*. Prostoru, který chceme dlaždičkovat, říkejme *zeď*. Ta je obdélníková a má rozměry $r \times s$ (přičemž jednotkou bude délka hrany dlaždice). Okraje zdi jsou rozděleny na úseky o jednotkové délce, a každý z těchto úseků má podobně jako hrany dlaždic nějaké obarvení.

Jako dláždění označujeme pokrytí zdi dlaždicemi, pokud toto pokrytí splňuje několik podmínek. V první řadě požadujeme, aby v každém z $r \times s$ čtverců

byla umístěna právě jedna dlaždice. Dále každé dvě sousední dlaždice musí mít ty hrany, kterými se dotýkají, obarvené stejnou barvou. Požadavek na stejnou barvu máme i na okrajové dlaždice, tedy dlaždice, které přiléhají k okraji zdi, musí mít příslušnou hranu obarvenou stejnou barvou, jakou je obarvený příslušný úsek okraje. Poslední podmínka, dlaždice nesmíme při tvorbě dláždění otáčet. Dodejme ještě, že každou dlaždici smíme použít libovolně-krát.

Pomocí dláždění, resp. jeho existence nebo neexistence, můžeme snadno rozhodovat úlohy, na které se odpovídá ANO, nebo NE. Jak to uděláme? Musíme sestavit vhodnou množinu dlaždic, z kterých budeme smět vybírat při tvorbě dláždění. Horní okraj zdi obarvíme podle vstupu. Ještě potřebujeme obarvit ostatní okraje, s tím, že všechny jejich úseky obarvíme stejnou barvou (můžeme o tom tedy uvažovat jako o obarvení celého okraje jednou barvou). Dlaždice a obarvení vybíráme tak, aby dláždění existovalo, právě když odpověď na úlohu je ANO.

Možných dláždění (a jim příslušných množin dlaždic a obarvení okrajů) může existovat velmi mnoho, tak si je alespoň trochu omezme. Požadujeme, aby zeď byla vždy široká právě tak, jak dlouhý je vstup. Horní okraj tedy bude vstupu přesně odpovídat. Navíc chceme, aby výška zdi byla nejmenší možná.

To, co jsme před chvilkou popsali, je *dlaždicový program.* Ten se skládá z nějaké konečné množiny dlaždic a nějakého obarvení okrajů zdi. Formálně by se jednalo o uspořádanou čtveřici (D, ℓ, p, d), kde D je množina dlaždic a ℓ, p, d představují obarvení levého, pravého a dolního okraje.

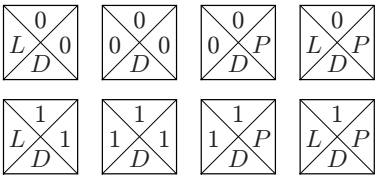
Program na zadaný vstup odpoví ANO, pokud je možné vydláždit nějakou zeď dlaždicemi z množiny D tak, aby horní okraj byl obarven podle vstupu a zbývající okraje barvami ℓ, p a d. Neexistuje-li žádné takové dláždění, výstupem programu je NE.

Dlaždicové programy jsou chráněná zvířátka, a tak jim budeme na vstupu předkládat pouze neprázdné řetězce.

Bývá hezké umět u programů ve výpočetním modelu určit složitost. U dlaždicových programů to zvládneme jednoduše: za dobu výpočtu prohlásíme minimální výšku zdi, pro kterou existuje dláždění (časová složitost je pak maximum z dob výpočtu přes všechny vstupy dané délky), použitou paměť pak představuje plocha vydlážděné zdi. Vstupy, na něž je odpověď NE, takže žádné dláždění neexistuje, složitost neovlivní.

Dost bylo teoretizování, pojďme se podívat, jak se náš model návštěvníkům předvede.

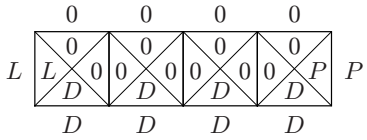
Mějme na vstupu nějakou posloupnost nul a jedniček. Naším úkolem je rozhodnout, jestli je tato posloupnost konstantní, tedy zda obsahuje pouze nuly nebo pouze jedničky. Využijeme k tomu dlaždicový program s dlaždicemi z obrázku vpravo. Může-



me je rozdělit do čtyř typů, každý typ existuje ve dvou barvách.

Levý okraj obarvíme L, pravý P, dolní D. Náš program určitě odpoví správně, a dokonce mu k tomu bude stačit jeden řádek. Takové odvážné tvrzení by se ale slušelo dokázat.

Jelikož všechny dlaždičky mají dolní hranu obarvenou D a žádná nemá barvou D obarvenou hranu horní, buď bude mít dláždění výšku 1, nebo vůbec nepůjde vytvořit. Pro konstantní posloupnost jistě dláždění existuje. V případě jednoprvkové posloupnosti použijeme příslušnou dlaždici čtvrtého typu, v případě posloupnosti delší pomocí vhodné dlaždice prvního typu „zvolíme číslo“ a následně ho „propagujeme“ až k pravému okraji – viz obrázek níže.



Platí i to, že vše, pro co dláždění existuje, je konstantní posloupnost. K levému i k pravému okraji může přiléhat vždy jen jedna konkrétní dlaždice (podle hodnoty na vstupu), a k jejich spojení je potřeba „předávat“ stále stejné číslo.

Úkol 1 [3b]: Sestavte dlaždicový program, který o posloupnosti nul a jedniček na vstupu zjistí, zda je v ní počet jedniček dělitelný třemi.

Úkol 2 [2b]: Mějme nějaký dlaždicový program, který pracuje v čase t, kde t je konstanta. Dokažte, že existuje jiný dlaždicový program, který odpovídá na tutéž otázku, ale stačí mu čas 1.

Závorkování nás stále baví

V jednotlivých dílech seriálu jste si mohli zkoušet v různých výpočetních modelech ověřovat, že zadaná posloupnost je správně uzávorkovaná. Toho jsou schopné i dlaždicové programy, a na rozdíl od počítadlových strojů z minulého dílu jim ani nemusíme posloupnost nějak speciálně kódovat.

Úkol 3 [4b]: Sestavte dlaždicový program, který o posloupnosti otvíracích a zavíracích závorek na vstupu rozhodne, zda je správně uzávorkovaná.

Úkol 4 [3b]: Dokažte, že rozhodnutí uzávorkování nelze pomocí dlaždicových programů dosáhnout v lepší než logaritmické časové složitosti. Kdybyste si nevěděli rady, zkuste alespoň dokázat, že konstantní čas nestačí.

Příbuzní Turingových strojů?

Když jsme se na začátku seriálu zastavili u Turingových strojů, nepřečetli jsme si jednu ceduli o jejich příbuzných.

Připomeňme si, že stroj se v každém kroku výpočtu rozhoduje podle stavu, ve kterém se právě nachází, a podle znaku na aktuálním políčku pásky. A každé kombinaci stavu a znaku jeho program přiřadí instrukci, která se má provést. Instrukce říká, co má stroj dál udělat (na jaký znak přepsat aktuální část vstupu,

kam se posunout, do jakého přejít stavu). Ke každé kombinaci stavu a znaku jsme měli právě jednu možnost.

Ale co kdyby těch možností bylo víc? Co kdybychom jedné kombinaci stavu a vstupu přiřadili hned několik možných reakcí? Přesně tak to totiž mají *nedeterministické Turingovy stroje.* Jak si ale takový stroj z možných reakcí vybere tu, kterou doopravdy provede?

Jedna možnost je představit si, že nedeterministický stroj umí *vracet* svůj výpočet. Pak můžeme říct, že nedeterministický stroj v každém kroku výpočtu vykoná první nevyzkoušenou možnou reakci (nevyzkoušenou v daném kroku) a pokračuje dál. Pokud se někdy dostane do stavu, kdy už nemá další možné reakce, nebo do koncového stavu NE, jednoduše vrací svůj výpočet až do toho kroku, kdy měl naposledy na výběr. Teprve v případě, že se vrátí do počátečního stavu a už nemá co vyzkoušet, zapíše NE.

Nebo si můžeme představit, že je stroj vybaven křišťálovou koulí (neboli orákulem), které mu pokaždé poradí takovou reakci, aby na konci výpočtu stroj odpověděl ANO. Jen pokud taková posloupnost rad neexistuje, odpověď zní NE.

Úplně mimo ale není ani představa, že v každém kroku se vesmír rozštěpí na tolik kopií, kolik má nedeterministický Turingův stroj právě možností, v každém ze vzniklých vesmírů se provede jedna reakce a výpočet pokračuje dál. Důležité pro nás je, jestli alespoň v jednom vesmíru dojde stroj do stavu ANO.

Úkol 5 [4b]: Dokažte, že dlaždicové programy jsou ekvivalentní nedeterministickým Turingovým strojům pracujícím v lineárním prostoru. Tedy máte za úkol dokázat, že jakýkoli nedeterministický Turingův stroj, který má lineární prostorovou složitost, lze reprezentovat jako dlaždicový program, a naopak, každý dlaždicový program (při našich omezeních na rozměry zdi) lze reprezentovat jako nedeterministický Turingův stroj pracující v lineárním prostoru.

Prozradíme vám malou nápovědu k předchozímu úkolu: tvrzení stačí dokázat o Turingových strojích pracujících v prostoru přesně n (kde n je velikost vstupu). Pokud totiž stroj používá prostor cn pro nějakou konstantu $c > 1$, můžeme podobně jako v úkolu 2 vytvořit jiný stroj, kterému bude stačit prostor n.

26-5-8 Automatizovaný graf — 15 bodů

Ve všech čtyřech minulých dílech tohoto seriálu jsme se potulovali po pomyslné zoo výpočetních modelů a zastavovali se u zajímavých exemplářů. Dnes naši letošní pouť zakončíme u jednoho podivného výběhu, ve kterém se nepase jedno velké výpočetní zvířátko, ale spousta maličkých. Řeč bude o *grafových automatech*.

Úvod

Grafový automat se skládá ze spousty stejných jednoduchých automatů (můžeme si je představit třeba jako malé programy, omezení viz níže), které jsou ně-

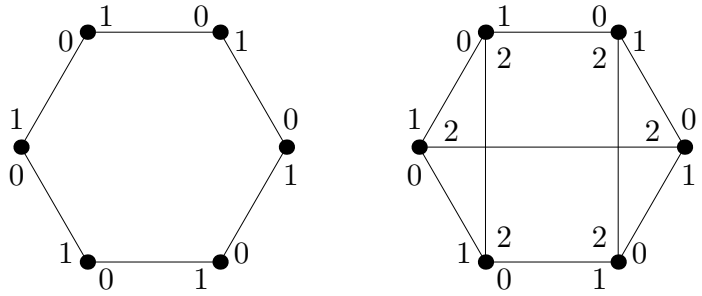
jakým způsobem pospojovány a společně řeší nějaký složitější problém. Abychom si nepletli pojmy, budeme dále grafovému automatu jako celku říkat *grafomat* a pojmem *automat* budeme označovat jednotlivé malé automaty obsažené v grafomatu.

Řečeno formálně, grafomat si můžeme představit jako obyčejný neorientovaný graf[23] tvořený množinou *vrcholů* a množinou *hran* mezi nimi. V každém vrcholu sídlí jeden automat a hrany nám vyjadřují to, jak jsou automaty mezi sebou propojené. Pro účely tohoto seriálu si dovolíme pojmy vrchol a automat ve vrcholu zaměňovat.

Aby navíc mohl každý automat rozpoznat své sousedy, jsou v každém vrcholu všechny hrany, které z něj vychází, očíslovány navzájem různými čísly $0, 1, 2, \ldots$ Jedna hrana přitom na obou koncích může (ale nemusí) dostat různá čísla, takže spíše než hrany číslujeme konce hran. Pokud budeme mluvit o nějaké hraně z pohledu určitého automatu, budeme její konce nazývat *místní* a *protější*.

Navíc budeme pro jednoduchost předpokládat, že ze všech vrcholů vede stejný počet hran. Grafům s touto vlastností se říká *regulární*, nebo přesněji K-regulární, kde K je počet hran vycházejících z vrcholu. To mimochodem znamená, že v celém grafu se nachází právě $N \cdot K/2$ hran, neboť každá hrana má 2 konce a konců napočítáme $N \cdot K$.

Ukázku 2-regulárního a 3-regulárního grafomatu na 6 vrcholech s očíslovanými hranami můžete vidět níže.



Automaty a jejich paměť

Už víme, jak grafomat vypadá jako celek, ale ještě bychom si měli popsat, jak vypadají jednotlivé automaty ve vrcholech. Jak už jsme řekli výše, všechny automaty musí být stejné – navzájem se budou odlišovat jen tím, co který z nich dostane na vstupu, a tím, co uvidí okolo sebe.

Formálně jde o *konečné automaty,* o kterých jsme psali seriál ve 23. ročníku. Abychom je nemuseli precizně definovat, budeme si je raději představovat jako velmi jednoduché programy. V příkladech a v řešeních budeme programovat

[23] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

v Pythonu, svá řešení můžete psát ve svém oblíbeném jazyce, pokud si myslíte, že je na to vhodný. Zavedeme ale několik omezení, aby možnosti našich programů odpovídaly možnostem konečných automatů.

Předně omezíme paměť: Každý automat si může pamatovat jen konstantně mnoho bitů informace nezávisle na velikosti vstupu. Můžeme použít libovolný pevný počet proměnných nějakého libovolného, ovšem omezeného rozsahu. Smíme si například pamatovat číslo od 0 do 42, ale nemůžeme si pořídit proměnnou, ve které bychom si spočítali počet vrcholů grafu – taková proměnná by musela mít horní mez závislou na velikosti vstupu, což nemáme dovoleno.

Druhé omezení vyplývá z prvního: V programech jednotlivých automatů nemůžeme použít *rekurzi* a pro všechny cykly musí existovat konstantní horní mez na počet iterací.

Zbývá určit, jak spolu mohou automaty komunikovat. Kdykoliv jsou dva automaty propojeny hranou, vidí si navzájem do paměti. Mohou si do ní ovšem jen nahlížet, ne ji jeden druhému přepisovat. Pro přístup k paměti sousedů máme v každém automatu k dispozici dvě pole indexovaná místním číslem hrany (tedy od 0 do $K-1$):

- $P[i]$ obsahuje protější číslo hrany s místním číslem i.
- $S[i]$ přistupuje k proměnným souseda připojeného hranou s místním číslem i. Pomocí konstrukce $S[i].\mathit{promenna}$ přečteme libovolnou sousedovu proměnnou. Jen se nesmíme odkazovat na jeho pole P a S, tedy není možné se například zeptat na proměnnou sousedova souseda.

Průběh výpočtu a jeho ukončení

Jak bylo naznačeno výše, výpočet probíhá v taktech. V každém taktu se automat může podívat na stav proměnných svých sousedů a podle nich a svého vlastního stavu provést nějaký výpočet a modifikovat vlastní proměnné. (Pokud během taktu soused své proměnné změnil, stále vidíme jejich stav z počátku taktu.)

Když se automat rozhodne, že už pro něj veškerá práce skončila, může zavolat speciální instrukci `stop`. Od této chvíle až do konce běhu celého grafomatu už tento automat nevykoná žádnou akci. Jeho sousedé stále mohou číst jeho proměnné, ale už není žádný způsob, jak by jeho výpočet mohl být opět nastartován. Poté, co instrukci `stop` zavolají všechny automaty v grafu, končí celý výpočet.

Druhou možností ukončení výpočtu je *ustálení*. Tím se myslí, že se dva takty po sobě nezmění v žádném automatu hodnota jakékoliv proměnné (rozmyslete si, proč potřebujeme dva takty a nestačí nám jeden – souvisí to s tím, že automaty vidí stav proměnných souseda jakoby o tah nazpět).

Pokud se grafový automat nikdy celý nezastaví (ani instrukcí `stop` ani ustálením), je to špatně a takový program je chybný.

Vstup je realizován tak, že se před prvním taktem výpočtu objeví ve smluvených proměnných každého automatu vstupní hodnoty (jaké a v jakých pro-

měnných, to záleží na úloze). Výstup je obdobný, po konci výpočtu by se ve všech automatech měla ve smluvených proměnných nacházet správná výstupní hodnota.

Při počítání složitosti nás bude zajímat jen počet taktů do zastavení celého grafového automatu (na rychlosti výpočtu jednotlivých automatů ve vrcholech nezáleží). Odhad počtu taktů stačí dělat jen asymptoticky (pomocí $\mathcal{O}$-notace), pokud to konkrétní úloha nebude vyžadovat jinak.

Příklad 1 – hledání dosažitelných vrcholů

Zadání: Mějme 5-regulární graf na N vrcholech a v každém vrcholu proměnnou a. Na začátku bude ve všech vrcholech $a = 0$ s jedinou výjimkou vrcholu A, kde bude $a = 1$. Po konci výpočtu by mělo být $a = 1$ ve všech vrcholech, kam lze z vrcholu A po hranách grafu dojít.

Řešení: Použijeme princip prohledávání grafu do šířky – každý vrchol bude sledovat své sousedy a ve chvíli, kdy se v nějakém z nich objeví $a = 1$, sám si také své a nastaví na 1. Až se hodnoty a ustálí, výpočet se přirozeně zastaví. Program pro jednotlivý automat bude vypadat následovně:

```
# Proměnné:
# a - rozsah 0..1
# i - rozsah 0..4, výchozí hodnota 0

for i in range(5):
        if S[i].a == 1:
                a = 1
```

Program vykoná nejvýše N taktů. Nejpomaleji poběží, pokud má graf tvar cesty. Jestliže bude graf „hustší“, program může doběhnout mnohem rychleji – například pro úplný graf vykonáme jen 3 takty: v prvním se všude nastaví $a = 1$ a zbylé dva slouží pro ustálení.

Příklad 2 – nalezení protějšího vrcholu

Zadání: Mějme 2-regulární graf na N vrcholech složený z jediného cyklu sudé délky (graf je tedy sudá kružnice délky N). Na začátku je jeden vrchol označen: má $x = 1$, zatímco ostatní $x = 0$. Chceme najít protější vrchol a také ho označit (nastavit mu $x = 1$).

Řešení: Pošleme si po kružnici signály směrem doleva i doprava a ve chvíli, kdy se potkají, tak víme, že jsme nalezli vrchol přesně naproti. Zde pro ukázku použijeme zastavení pomocí `stop`, i když by šlo napsat i verzi zastavující ustálením.

```
# Proměnné:
# x - rozsah 0..1
# signal - rozsah 0..1, výchozí hodnota 0

if x == 1:
        # Vyšleme úvodní signál na obě
```

```
            # strany a skončíme
            signal = 1
            stop
    if S[0].signal and S[1].signal:
            # Dostali jsme signál z obou stran
            x = 1
            stop
    elif S[0].signal or S[1].signal:
            # Signál přišel alespoň z jedné strany
            signal = 1
            stop
```


Celý výpočet poběží právě $N/2$ taktů.

Úkoly

Úkol 1 [3b]: Mějme 2-regulární graf na N vrcholech (N je dělitelné třemi) složený z jediného cyklu. Na začátku je jeden vrchol označený: má $x = 1$, zatímco ostatní vrcholy mají $x = 0$. Vaším úkolem je označit zbylé dva vrcholy ve třetinách kružnice. Tedy pokud si startovní vrchol označíme indexem 0, tak po konci běhu grafového automatu budou označeny právě vrcholy s indexy 0, $N/3$ a $2N/3$.

Pokud vám to pomůže, můžete předpokládat, že každý vrchol je spojený hranou s místním číslem 1 se sousedem po směru hodinových ručiček a hranou s místním číslem 0 s druhým sousedem (jako na obrázku v úvodu).

Úkol 2 [3b]: Mějme 5-regulární souvislý graf s jedním označeným vrcholem (bude mít na začátku proměnnou $a = 1$, ostatní vrcholy ji budou mít nulovou). Vaším úkolem bude najít nějakou *kostru* tohoto grafu, tedy nějakou podmnožinu hran takovou, že stále spojuje všechny vrcholy, ale neobsahuje žádný cyklus.

Pro výstup použijte pole proměnných *kostra*[i] obsahující pět prvků. Na začátku bude toto pole v každém vrcholu plné nul, na konci by v něm měly být jedničky právě na pozicích, které odpovídají místním číslům hran, které jsou v nalezené kostře (pozor, pamatujte na to, že místní a protější číslo hrany nemusejí být stejné a že je nutné nastavit pole *kostra*[i] na obou koncích hrany).

V předchozích dvou úkolech stačilo odhadovat počet taktů asymptoticky, nebylo by ale zajímavé zkusit si vyrobit program, o kterém budeme vědět úplně přesně, jak dlouho poběží?

Úkol 3 [4b]: Vyrobte program, který bude na K-regulárním grafu na N vrcholech ($N \geq 1$) běžet přesně $C \cdot N$ kroků, kde C bude nějaká konstanta, která může záviset na velikosti K. Pokud se vám však povede C stanovit pevně (bez závislosti na K), bude to ještě lepší. Pokud vám to pomůže, můžete, jako v předchozích úkolech, předpokládat, že právě jeden vrchol bude nějakým způsobem označen.

Možná už začínáte být nervózní, už je skoro konec seriálu a ještě nebylo ani slovo o tradičním uzávorkování. Nebojte se, zde přichází:

Úkol 4 [5b]: Vaším úkolem bude zkontrolovat správnost uzávorkování skládajícího se z levých a pravých závorek. Správné uzávorkování je takové, kde jsou závorky správně spárovány a páry se nekříží.

V řeči grafomatu budou závorky zakódované hodnotami ve vrcholech 2-regulárního grafu (kružnice) na N vrcholech: v prvním vrcholu bude zapsána první závorka, ve druhém druhá, a tak dále. Poslední vrchol pak bude navíc hranou spojen s prvním, aby byla kružnice uzavřená.

Závorky budou zapsány pomocí proměnné *zavorka* a to tak, že hodnota 1 bude odpovídat levé (otevírací) závorce a hodnota -1 pravé (uzavírací) závorce. Navíc bude první vrchol označen pomocí $prvni = 1$, aby šel jednoduše poznat.

Na konci běhu by ve všech vrcholech měla být nastavená proměnná *vystup* na hodnotu 1, pokud bylo uzávorkování korektní, nebo na hodnotu 0, pokud nebylo.

Pár slov závěrem

S grafomaty jste se již mohli potkat při studování starých ročníků Matematické olympiády kategorie P. Naše grafomaty jsou velmi podobné (ale ne identické – třeba se liší v podmínkách zastavování), takže se pro inspiraci můžete podívat i na studijní texty a úlohy 56. ročníku olympiády.[24]

Možná vám grafomaty přijdou jako zajímavý teoretický model, který nemá s praxí pranic společného. Opak je však pravdou. Speciální verzí grafomatů jsou třeba takzvané *buněčné automaty* a nejznámější z nich je asi Conwayova hra Život.[25] Ta se svým fungováním blíží svému biologickému předobrazu – buňky v těle složitějších organismů jsou také často jedna jako druhá (alespoň v rámci stejné tkáně) a každá z nich reaguje jen na to, co „vidí“ okolo sebe.

Mimo to grafomat můžeme považovat za jeden z dosti realistických modelů paralelních počítačů. Z fyzikálních zákonů totiž plyne, že vzdálené procesory spolu nemohou komunikovat okamžitě (kvůli konečné rychlosti světla) a že každý procesor může komunikovat pouze s velmi omezeným počtem sousedů (kvůli omezené dimenzi prostoru). Obě omezení grafomat poměrně věrně zachycuje.

[24] `http://mo.mff.cuni.cz/p/56/zadani-1.html#P-I-4`

[25] `http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`

Recepty z programátorské kuchařky

Kuchařka první série – základní algoritmy

Tato naše kuchařka je nejzákladnější ze základních a je určena hlavně pro začínající řešitele. To však neznamená, že zkušenější řešitelé do ní nahlédnout nemůžou – třeba na nějakou konkrétní programátorskou techniku, kterou by si potřebovali osvěžit.

V první části kuchařky se seznámíme hlavně se základními principy programování, uchovávání dat v počítači a základy rychlé manipulace s nimi. Po přečtení této části bychom měli být schopní převést své myšlenky z hlavy na papír či do počítače a měli bychom vědět, proč je námi zvolený postup rozumný.

Druhá část nás poté seznámí se základními postupy, jak řešit určité konkrétní problémy. Naučíme se například, jak rychle vyhledávat v uspořádané posloupnosti hodnot, nebo jak si pomocí předpočítání usnadnit řešení těžké úlohy.

Většinu klíčových částí se pokusíme též ukazovat v podobě zdrojového kódu v několika různých jazycích (v nízkoúrovňovém C, abychom viděli implementaci blízkou tomu, jak to vidí počítač, a v Pythonu, kde je psaní o něco příjemnější). Nebudeme ale probírat základy syntaxe těchto jazyků, tu si případně můžete nalézt jinde.[26] Pokud žádný z těchto jazyků neumíte, nezoufejte. KSP můžete řešit i bez toho, stačí když svá řešení důkladně slovně popíšete (konkrétní jazyk se pak můžete naučit až během řešení).

Část první: Základní pojmy

Algoritmus a program

Pod tajemným slovem *algoritmus* se skrývá jen jiný výraz pro postup. Můžete si to představit jako příkaz od maminky „Běž do krámu, kup chleba, a když budou mít měkké rohlíky, tak jich vem tucet.“[27]

Takovýto příkaz klidně můžeme nazvat algoritmem, ačkoliv to bude asi znít nezvykle – pojem algoritmus se totiž používá hlavně ve světě počítačů. Je to tedy nějaká posloupnost základních příkazů, která řeší nějaký problém. Výběr konkrétního programovacího jazyka rozhoduje o tom, jaké základní příkazy budeme mít k dispozici. V základu jsou ale skoro stejné.

Mezi základní příkazy patří:

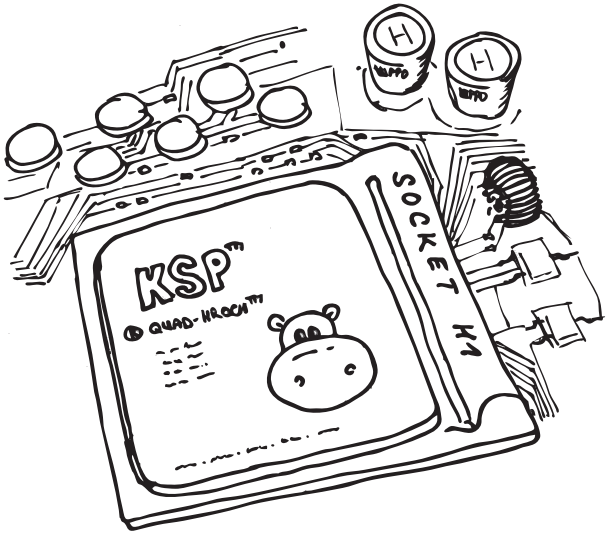
- Manipulace s daty v paměti (uložení či načtení hodnoty, více detailně v další kapitole)
- Provedení nějakého numerického výpočtu $(+, -, *, /)$

[26] `http://ksp.mff.cuni.cz/study/odkazy.html`

[27] A jako slušně vychovaní se tedy vydáte do krámu a koupíte tucet chlebů, protože měli měkké rohlíky :-)

- Vyhodnocení nějaké podmínky a odpovídající větvení programu *(Pokud A, tak proveď B, jinak proveď C)*
- Opakování nějakého příkazu *(Dokud platí A, dělej B)*
- Vstup a výstup programu (nejtypičtější je asi vstup od uživatele z klávesnice či načtení vstupu ze souboru; výstup pak může znamenat vypsání výsledku na obrazovku nebo třeba zapsání dat do souboru)

Z těchto základních stavebních kamenů se pak skládá každý další algoritmus. Programem pak rozumíme realizaci algoritmu v nějakém konkrétním programovacím jazyce.



Reprezentace dat v počítači

Celkem často si v průběhu výpočtu našeho algoritmu potřebujeme pamatovat nějaké hodnoty. K tomu nám programovací jazyky dávají nástroj s názvem *proměnná*. Ta představuje jakési pojmenované místo v paměti (přihrádku), do kterého si můžeme data ukládat a pak je zase načítat.

Typickým příkladem může být počítání součtu čísel, která nám uživatel zadá na vstupu. Na začátku nejdříve do nějakého místa v paměti uložíme hodnotu 0. Poté postupně, jak nám uživatel zadává čísla, tuto proměnnou pokaždé přečteme, k její hodnotě přičteme zadané číslo a výsledek opět uložíme na stejné místo.

Takovéto použití jedné proměnné je velmi jednoduché (tak jednoduché, že ho takto podrobně do řešení KSPčka nepište, není to potřeba), ale také celkem omezené. Co kdybychom si chtěli pamatovat třeba celou zadanou posloupnost

čísel? Mohlo by nám stačit vyrobit si spoustu různě pojmenovaných proměnných, ale nejde to lépe? Jde.

Jednotlivé proměnné se mohou kombinovat do složitějších konstrukcí, které obecně nazýváme *datovými strukturami*. Zkusíme si ty nejzákladnější představit.

Pole

KSP

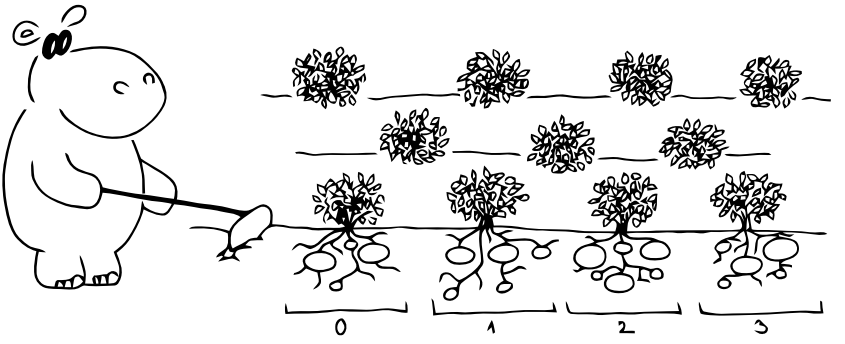
První datovou strukturou, kterou si představíme a která se na výše nastíněnou situaci náramně hodí, je *pole*. To představuje spoustu přihrádek (proměnných) naskládaných v paměti za sebou, ke kterým typicky přistupujeme přes jeden společný název pole a jejich pořadové číslo neboli index (jako `NazevPole[0], NazevPole[1], ...`).[28]

Ve většině základních jazyků je pole jen statické, tedy to znamená, že v okamžiku jeho vytváření musíme počítači říct, jak ho chceme velké. Některé vyšší jazyky ale nabízejí i pole, které se dynamicky zvětšuje, takovou konstrukci si ukážeme ve druhé části kuchařky.

recepty

Abychom nebyli omezeni jen jedním rozměrem, můžeme si klidně vyrobit pole dvourozměrné (případně obecně n-rozměrné). Dvourozměrné pole je vlastně tabulka hodnot, nazýváme ji také někdy *matice*, a může se nám hodit například při reprezentaci různých map (plán bludiště) nebo, jak uvidíme níže, pro reprezentaci dalších datových struktur.

U pole již má smysl přemýšlet, jak dlouho která operace bude trvat. Díky tomu, že jsou jednotlivé prvky v poli naskládané pevně za sebou, tak když se počítače zeptáme na obsah přihrádky `pole[42]`, přesně ví, na které adrese v paměti se její obsah nachází, a proto nám hodnotu vrátí ihned.



Tomu budeme říkat *operace v konstantním čase* a budeme značit, že trvá čas $\mathcal{O}(1)$. Efektivitu programu totiž nepočítáme v sekundách (protože každý

[28] Pozor, ve světě počítačů se velmi často indexuje od nuly, tedy první prvek má v tomto případě index 0

z nás má asi jinak rychlý počítač), ale v počtu základních operací, které musí program řádově vykonat. Více o časové složitosti si můžete přečíst v kuchařce o složitosti,[29] nejdříve však doporučujeme dočíst tuto kuchařku.

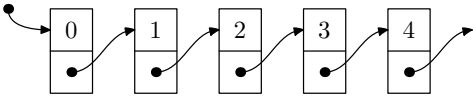
Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu pozici dál, aby se vkládaný prvek vešel na své místo. Taková operace tedy může pro pole délky N prvků trvat řádově až N kroků, což zapisujeme jako $\mathcal{O}(N)$ a říkáme, že je to vzhledem k N *lineární časová složitost*.

To je docela značná nevýhoda oproti struktuře, kterou si ukážeme za chvíli. Určitě ale pole nezavrhujme. Je to základní datová struktura, která nalezne použití ve spoustě programů, a jak si ve druhé části kuchařky ukážeme, můžeme ho použít třeba k rychlému hledání hodnoty metodou *binárního vyhledávání*. Nyní ale již slibovaná další datová struktura...

Spojový seznam a ukazatele

Pole jsme měli v paměti určené jenom tím, že počítač věděl, kde je jeho začátek a kolik místa v paměti zabírají jeho prvky. Při dotazování na konkrétní index pak podle indexu a podle velikosti prvků počítač přesně věděl, kam do paměti se má podívat, aby našel námi požadovaný prvek (to vše zvládl v konstantním čase). Jednotlivé prvky si tedy vůbec nemusely pamatovat, kde se nachází jejich sousedi, protože všechny prvky seděly v paměti za sebou.

Představme si ale teď situaci, kdy by si každý prvek ještě pamatoval pozice sousedů. Pak bychom mohli mít prvky libovolně rozházené v paměti a jen by se na sebe vzájemně odkazovaly (první prvek by tvrdil, že druhý je na pozici X, druhý by tvrdil, že třetí je na pozici Y, a tak dále).



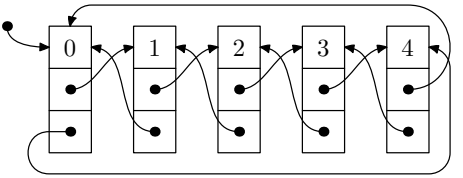
K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každá část paměti v počítači má nějaké své číslo. Když si vytváříme nějakou pojmenovanou proměnnou, tak ta vlastně odkazuje na nějaké místo v paměti a na tomto místě v paměti je její hodnota.

Co kdyby ale hodnota proměnné byla adresa nějakého jiného místa v paměti? Pak takové proměnné říkáme *pointer* a umožňuje nám vytvářet výše popsanou strukturu rozházených prvků v paměti.

Spojový seznam je tedy určený svým prvním prvkem (máme v jedné proměnné pointer na tento prvek, který se často nazývá *kořen*, protože z něj „vyrůs-

[29] `http://ksp.mff.cuni.cz/viz/kucharky/slozitost`

tá“ zbytek struktury) a poté u každého dalšího prvku máme za sebou uloženou hodnotu tohoto prvku a případně odkaz (pointer) na další prvek. Odkazy mezi prvky mohou být i obousměrné, mohou vést dokola (poslední ukazuje na první) či mohou dokonce tvořit nějakou složitější strukturu (pak to ale již nebude čistý spojový seznam).



Co nám takto vystavěná struktura umožňuje v porovnání s polem? Přístup na konkrétní prvek v ní stojí lineárně času, protože ho musíme „odkrokovat“ od prvního prvku (na který máme pointer), tedy musíme udělat až $\mathcal{O}(N)$ kroků. Pokud bychom však pointer na daný prvek už nějak měli, tak na něj samozřejmě můžeme přistoupit v konstantním čase.

Naopak přidávání prvků na konkrétní místo (i jejich odebírání) máme v podstatě zadarmo a spojový seznam můžeme rozšiřovat, dokud na něj máme v počítači paměť. Ve chvíli, kdy chceme přidat nový prvek za prvek, na který máme pointer, tak jen šikovně přepojíme ukazatele. Pokud předtím ukazatele vedly $A \to B$, tak teď povedou $A \to C \to B$ (a při odebírání naopak).

Zde můžete vidět ukázku pointerů a spojových seznamů v jazyce C, kde jsou tyto věci mnohem více nízkoúrovňové (ale zato rychlejší):

```
typedef struct tprvek tprvek;

// Struktura pro prvek obsahující dopředné
// i zpětné odkazy
struct tprvek {
    int hodnota;
    tprvek *dalsi;
    tprvek *predchozi;
};

// Vytvoří nový prvek:
tprvek *novy(int i) {
    tprvek *aktualni =
        malloc(sizeof(tprvek));
    aktualni->dalsi = NULL;
    aktualni->predchozi = NULL;
    aktualni->hodnota = i;
    return aktualni;
}
```

```
// Odstraní prvek a vratí pointer na další
// prvek (vrácení pointeru se hodí při
// odstraňování kořene):
tprvek *odstran(tprvek *aktualni) {
    if (aktualni->predchozi != NULL)
        aktualni->predchozi->dalsi =
            aktualni->dalsi;
    if (aktualni->dalsi != NULL)
        aktualni->dalsi->predchozi =
            aktualni->predchozi;

    tprvek *pomocna = aktualni->dalsi;
    free(aktualni);
    return pomocna;
}

// Vloží a vrátí pointer na nový prvek:
tprvek *vloz_za(tprvek *aktualni, int i) {
    tprvek *pomocna = aktualni->dalsi;
    aktualni->dalsi = novy(i);

    if (pomocna != NULL)
       pomocna->predchozi = aktualni->dalsi;

    aktualni->dalsi->dalsi = pomocna;
    return aktualni->dalsi;
}

// Použití:
int main() {
    tprvek *koren = novy(1);
    tprvek *aktualni = vloz_za(koren, 2);

    aktualni = koren;
    while (aktualni != NULL) {
        printf("%d\n",aktualni->hodnota);
        aktualni = aktualni->dalsi;
    }

    return 0;
}
```

Na následující straně je ukázka spojových seznamů v Pythonu, kdybychom si je podobně jako v C chtěli naprogramovat sami (Python totiž obsahuje spoustu základních struktur již hotových, podívejte se na modul jménem `collections`).

```
class Prvek:
    def __init__(self, hodnota):
        self.hodnota = hodnota
        self.dalsi = None
        self.predchozi = None

class Spojak:
    def __init__(self):
        self.koren = None

    def Vypis(self, aktualni):
        if aktualni is not None:
            print aktualni.hodnota
            self.Vypis(aktualni.dalsi)

    def VlozPo(self, prvek, zaPrvek = None):
        if zaPrvek is not None:
            prvek.dalsi = zaPrvek.dalsi
            prvek.predchozi = zaPrvek
            zaPrvek.dalsi = prvek
        if prvek.dalsi is not None:
            prvek.dalsi.predchozi = prvek
        if self.koren is None:
            self.koren = prvek

    def Odstran(self, prvek):
        if prvek.predchozi is not None:
            prvek.predchozi.dalsi = \
                prvek.dalsi
        if prvek.dalsi is not None:
            prvek.dalsi.predchozi = \
                prvek.predchozi

# Použití:
prvekA = Prvek("A")
prvekB = Prvek("B")
prvekC = Prvek("C")
prvekD = Prvek("D")

seznam = Spojak()

seznam.VlozPo(prvekB)
seznam.VlozPo(prvekD, prvekB)
seznam.VlozPo(prvekC, prvekD)
seznam.VlozPo(prvekA, prvekC)
seznam.Odstran(prvekC)

seznam.Vypis(seznam.koren)
```

Tyto základní struktury už jsou často předpřipravené jako součást nějakých knihoven v daném jazyce, ale je velmi důležité rozumět tomu, jak vnitřně fungují. Protože jedině když budeme vědět, co je jak rychlé a efektivní, tak budeme schopni psát rychlé programy.

Teď již víme, jak reprezentovat nejzákladnější datové struktury v počítači, ale mohlo by se nám hodit zastavit se ještě chvíli nad dalšími strukturami, tentokrát již více teoreticky.

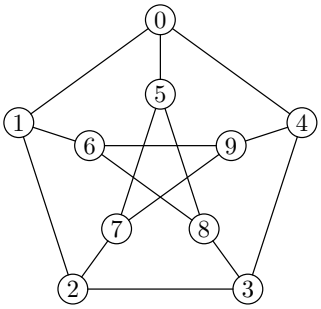
Stromy a grafy v informatice

Grafy

S nějakými grafy jste se již možná potkali, ale tento pojem je bohužel docela přetěžovaný. Jedním jeho významem jsou „koláčové grafy“ a jiné další diagramy znázorňující nějaký poměr (ať už to jsou výsledky voleb, nebo poměr lidí, kteří sledovali v televizi Večerníček).

Další význam můžeme nalézt v analytické matematice, kde se potkáme s grafy průběhu nějakých funkcí. My však nemáme na mysli ani jedno ze zmíněných, my se budeme bavit o *kombinatorických grafech*.

Grafem tedy máme na mysli nějakou množinu objektů, říkejme jim *vrcholy*, a nějaké vztahy mezi nimi. Tyto vztahy nazýváme *hranami* a jsou vyjádřené dvojicemi vrcholů, mezi kterými vedou. Ukázku takového grafu vidíme třeba na následujícím obrázku.



Jako praktickou ukázku grafu si můžeme například představit silniční síť nějakého státu: vrcholy budou města a hrany budou silnice, které mezi nimi vedou.

Graf můžeme doplnit třeba tím, že si v každém vrcholu nebo na každé hraně budeme pamatovat nějakou hodnotu (například cenu nejlevnějšího benzínu ve městech a délku v kilometrech na silnicích), pak graf nazýváme *ohodnocený*. Další možnou úpravou je, že každá hrany povede jen jedním směrem (jednosměrné silnice), takovým grafům říkáme *orientované* (pokud pak v orientovaném grafu chceme silnici oběma směry, prostě do něj přidáme dvě hrany, jednu v každém směru).

Také se můžete setkat s pojmem *souvislý* graf. Ten znamená jen to, že mezi každými dvěma vrcholy existuje nějaká neorientovaná cesta. Pokud tomu tak není, tak je graf *nesouvislý* a dá se rozložit na několik menších grafů, které již souvislé jsou a říká se jim *komponenty souvislosti*.

Poslední, co nám schází k praktickému použití grafů, je naučit se, jak je reprezentovat v počítači. Existuje několik možností (n bude značit počet vrcholů, m počet hran):

- **Seznam sousedů** – vrcholy grafu budeme mít uložené v poli a u každého vrcholu budeme mít (spojový) seznam čísel dalších vrcholů, do kterých z aktuálního vrcholu vede hrana. Zabírá místo $\mathcal{O}(n+m)$ a hodí se pro řídké grafy (tedy grafy, kde je m řádově stejné jako n).
- **Matice sousednosti** – tabulka $n \times n$, kde na souřadnicích $[i,j]$ je jednička (případně jiná hodnota, v případě ohodnoceného grafu), pokud z i do j vede hrana, a nula, pokud tam hrana není (u neorientovaných grafů je navíc matice symetrická – je jedno, jestli vezmeme $[i,j]$ nebo $[j,i]$). Hodí se pro husté grafy, kde $m \sim n^2$.
- **Matice incidence** – řádky reprezentují vrcholy, sloupce hrany. V každém sloupci jsou právě dvě jedničky – indexy vrcholů, mezi kterými hrana vede. Zabírá však $\mathcal{O}(mn)$ a její použití bývá dost neohrabané, takže je většinou lepší dát přednost jiné reprezentaci grafu. Je však dobré o ní vědět.

KSP

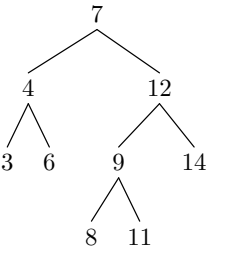
recepty

Grafy jsou velmi široké téma. Můžeme hledat jejich minimální kostry, můžeme v nich hledat nejkratší cesty či skrze ně pouštět pod tlakem vodu. Více o nich si tedy můžete přečíst v některé z našich specializovaných grafových kuchařek, které odkazujeme z našeho kuchařkového rozcestníku.[30]

Stromy

Možná si říkáte, co má informatika u všech elektronů společného s lesnictvím? Kupodivu celkem mnoho a bez stromů bychom se v mnoha případech jen těžko obešli. Informatické stromy sice nejsou většinou tak zelené, mají ale, na rozdíl od svých dřevnatých sourozenců, mnoho jiných pěkných vlastností.

Strom je vlastně speciálním případem souvislého grafu, který neobsahuje žádný cyklus. To znamená, že mezi každými dvěma vrcholy stromu existuje právě jedna cesta. Strom pak můžeme za nějaký zvolený vrchol *zakořenit*. Tím se nám z tohoto vrcholu stane *kořen*, ze kterého směrem dolů (informatické stromy mají tradičně kořen nahoře) vyrůstají nějaké *podstromy*.



U stromu navíc mluvíme o *hloubce*, to je vzdálenost od kořene k danému vrcholu. Hloubka celého stromu pak je nejdelší vzdálenost od kořene k nějakému *listu* (tak říkáme vrcholům, které již nemají žádné *syny*, tedy vrcholy, které by z nich vyrůstaly). Vrcholy stromu také můžeme podle jejich hloubky uspořádat do jednotlivých *hladin*.

Velmi často používáme stromy, které jsou nějak pravidelné. Časté jsou *binární stromy*, které mají v každém vrcholu maximálně dva syny (levý a pravý podstrom). Reprezentovat se dají buď obecně jako každý jiný strom (v každém vrcholu spojový seznam podstromů), nebo velmi pěkně i v poli.

[30] `http://ksp.mff.cuni.cz/study/cooks/`

Stačí si uvědomit, že pokud vrcholy očíslujeme od nuly, tak vrchol s indexem i má jako syny vrcholy s indexy $2i + 1$ a $2i + 2$. Pokud bude strom úplný – to znamená, že bude mít poslední hladinu plně zaplněnou (bez děr) – tak bude i pole plně zaplněné bez prázdných míst.

Speciálním případem je pak ještě *binární vyhledávací strom*. Je to normální binární strom, pro nějž navíc platí, že všechny hodnoty v levém podstromu jsou menší než hodnota ve vrcholu a všechny v pravém podstromu naopak větší.

Na složitější datové struktury stavějící na těchto základních (haldy, intervalové stromy, ...) se můžete podívat do některé z našich dalších kuchařek, na jejichž přehled jsme vás už odkázali o kapitolu výše.

Část druhá: Programátorské techniky

Tato část by měla sloužit jako rychlý přehled a ukázka různých technik, které se dají použít při řešení úloh z KSPčka, nebo při programování obecně.

Rekurze

Rekurze je velmi důležitá programátorská technika. V podstatě znamená definování nějaké věci (ať už je to nějaký objekt či postup výpočtu) pomocí sebe sama.

Rekurzivně může být například zadána nějaká datová struktura. Například stromy jsou pěkným příkladem rekurzivně definované datové struktury – každý vrchol stromu může mít syny, a každý z těchto synů je sám o sobě strom.

Prakticky je to realizováno tak, že každý vrchol má svou hodnotu a pak ještě seznam ukazatelů vedoucích na další případné podstromy. S ukazateli jsme se již potkali a s jejich pomocí jsme si postavili spojový seznam. A přesně tak, spojový seznam je také ve své podstatě rekurzivní datová struktura.

Mimo rekurzivních datových struktur se ale často potkáváme i s rekurzivním postupem výpočtu nějakého programu, nejčastěji realizovaným ve formě *rekurzivní funkce*, která volá sama sebe (s jinými parametry, jinak by to asi nemělo smysl).

U rekurzivních funkcí je nejdůležitější věc definovat nějakou *okrajovou podmínku*, tedy podmínku, při níž už se rekurze zastaví. Jinak by se totiž mohlo stát, že by rekurze běžela donekonečna.[31]

Rekurzivní funkce a převod na nerekurzivní cyklus

Typickým příkladem rekurzivní funkce je výpočet Fibonacciho čísel. Ta jsou definována tak, že $f_0 = 1$, $f_1 = 1$ a n-té Fibonacciho číslo je součtem dvou předchozích ($f_n = f_{n-1} + f_{n-2}$). To nám dává posloupnost čísel $0, 1, 1, 2, 3, 5, 8, \ldots$. Pokud toto přepíšeme do programového kódu, tak dostáváme následující zápisy:


```
// Kód v C:
int fib(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;
        else return fib(n-1) + fib(n-2)
}
```

```
# Kód v Pythonu:
def fib(n):
        if n==0:
                return 0
        elif n==1:
                return 1
        else:
                return fib(n-1) + fib(n-2)
```

Jak vidíme, je přepis celkem přímočarý. Pokud by se nám však rekurze v nějakém případě nelíbila, tak se každé rekurze můžeme zbavit. Rekurzivní volání totiž můžeme šikovně přepsat na nějaký cyklus se *zásobníkem* (spojový seznam, do kterého vkládáme jen z jedné strany a ze stejné strany z něj i odebíráme – tedy prvek přidaný jako první odebereme až jako poslední).

Pak jen v cyklu odebíráme prvky ze zásobníku, dokud není prázdný, a za každé rekurzivní zavolání do zásobníku přidáme parametry, se kterými bychom naší funkci volali. Tímto postupem převedeme každou rekurzivní funkci na nerekurzivní.

Ještě doplníme poznámku, že ve většině programovacích jazyků každé volání funkce stojí nějaký čas, sice malý, ale když se volání provádí opakovaně, tak se to už nasčítá. Pro reálnou implementaci je tedy nejlepší pokusit se rekurzi převést na nerekurzivní volání, pokud to nějak rozumně jde.

Ale občas to jde dokonce i jednodušeji a bez zásobníku. Podívejte se na alternativní variantu výpočtu Fibonacciho čísel níže a rozmyslete si, co dělá.

[31] Tedy přesněji do doby, kdy našemu počítači dojde paměť, do které si ukládá jednotlivá volání funkcí.

```
// Kód v C:
int fib2(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;

        int a = 0; int b = 1;
        for(int i = 2; i<=n; i++) {
                int c = a + b;
                a = b;
                b = c;
        }
        return b;
}

# Kód v Pythonu:
def fib2(n):
        if n==0:
                return 0
        elif n==1:
                return 1

        a = 0; b = 1
        while n>1:
                (a, b) = (b, a+b)
                n-=1
        return b
```


Jak vidíte, je i tato funkce elegantní a navíc běhá mnohem rychleji, než její rekurzivní varianta. Tato funkce běhá v $\mathcal{O}(n)$, kdežto rekurzivní varianta počítala stejné věci mnohokrát dokola (zkuste si nakreslit nějaký strom volání předchozí funkce, případně se podívat dopředu do kapitoly Předpočítané mezivýsledky).

Rekurzivní varianta tedy běžela až v čase $\mathcal{O}(2^n)$, což je pro velká n mnohem pomaleji než $\mathcal{O}(n)$ (avšak šla by celkem snadno zachránit, aby běžela také v $\mathcal{O}(n)$, zkuste si rozmyslet jak).

Backtracking

S rekurzí silně souvisí i pojem *backtracking*, česky by se snad dalo říci „metoda pokusu a omylu." Tímto pojmem označujeme proces, kdy postupně zkoušíme všechny možnosti, jak vyřešit nějaký problém.

Zpětné vyhledávání se tento proces nazývá proto, že pokud již nemůžeme pokračovat dál (třeba v případě, že v bludišti dojdeme do slepé uličky), vrátíme se kus zpět a zkusíme jinou (zatím nevyzkoušenou) možnost. Takto postupně zkusíme každou možnost a buď nalezneme námi hledané řešení, nebo se vrátíme až na výchozí pozici a zjistíme, že řešení neexistuje.

Backtracking bývá často realizován pomocí rekurze, ukážeme si to na příkladu hledání rozkladu zadané částky na mince o hodnotách 5 Kč a 3 Kč (všimněte si, že v takto omezeném peněžním systému nejde složit třeba částka 7 Kč). Naše funkce dostane jako parametr zbývající částku a zkusí rekurzivně provést rozklad na jednotlivé mince:

```
// Kód v C:
bool rozloz(int castka) {
        // Koncová podmínka rekurze
        if(castka == 0) return true;
        else if(castka < 0) return false;
        else if(rozloz(castka-5)) {
                printf(" 5 Kc");
                return true;
        } else if(rozloz(castka-3)) {
                printf(" 3 Kc");
                return true;
        } else return false;
}
```

```
# Kód v Pythonu:
def rozloz(castka):
        if castka == 0:
                return True
        elif castka < 0:
                return False
        elif rozloz(castka-5):
                print " 5 Kc"
                return True
        elif rozloz(castka-3):
                print " 3 Kc"
                return True
        else:
                return False
```

Jak je vidět, tak v každém kroku zkusíme nejdříve použít pětikorunovou minci, a když náš rozklad nevyjde, zkusíme v tomto kroku použít ještě tříkorunu. Takto se rozhodujeme v každém kroku rekurze a případně se vracíme z neúspěšných větví výpočtu a zkoušíme další možnosti.

Takovým postupem vyzkoušíme až exponenciálně mnoho možností ($\mathcal{O}(2^n)$), což není moc rychlé. Proto je doporučováno se backtrackování raději vyhnout, nebo ho nějak chytře vylepšit. Je však dobré o backtrackování vědět, protože existují problémy, které efektivněji řešit neumíme.

Rozděl a panuj

Jednou ze základních technik je rozdělení složitějšího problému na menší části, které opět můžeme rozdělit na menší a tak dále, dokud se nedostaneme k problémům tak malým, že je už umíme triviálně vyřešit.

Binární vyhledávání v poli

Představte si, že máme seřazené pole n prvků a chceme zjistit, jestli se v něm nachází prvek s hodnotou k. Určitě můžeme projít celé pole v lineárním čase (tím, že budeme brát jeden prvek za druhým a kontrolovat, zda je roven hodnotě k), ale to je zbytečně pomalé a nevyužívá toho, že máme pole seřazené.

Můžeme totiž začít s velkým problémem a ten postupně zmenšovat na stále menší a menší. Nejdříve hledáme k v celém poli. Podíváme se na jeho prostřední prvek: Pokud je roven k, jsme hotovi. Je-li větší než k, víme, že se k musí nacházet nalevo od něj. Můžeme tedy hledat znovu, ale tentokrát jen v levé polovině pole. Analogicky když je prostřední prvek menší než k, hledáme v pravé polovině.

Jelikož se nám každým krokem problém zmenší na polovinu, tak se maximálně po $\log n$ krocích dostaneme na pole velikosti jedna. Říkáme, že algoritmus má *logaritmickou časovou složitost*, píšeme $\mathcal{O}(\log n)$.[32]

Prakticky postup provádíme tak, že si udržujeme levý a pravý okraj aktuálně zpracovávaného úseku a postupně je k sobě přibližujeme.

Ukázka hlavní smyčky v C:

```
int pole[] = {1,2,5,7,12,16,42};
int hledane = 8;
int L = 0; int R = 6;

do {
    int prostredni = (L+R)/2;
    int x = pole[prostredni];
    if (x == hledane)
        printf("Pole obsahuje hledane\n");
    else if (x < hledane)
        L = prostredni + 1;
    else
        R = prostredni;
} while(L < R && x != hledane);

if (x != hledane)
    printf("Hledane neni v poli\n");
```

[32] Pokud není řečeno jinak, tak pro nás v informatice značka log znamená *dvojkový logaritmus*, což je funkce opačná k funkci 2^n a roste o hodně pomaleji než funkce lineární. Pro velká n platí: $1 < \log n < n$ a například $\log 2 = 1, \log 8 = 3, \log 1024 = 10$.

Ukázka v Pythonu jako funkce vracející index prvku nebo −1:

```
def bin_vyhled(pole, hledane, L=0, R=None):
    if R is None:
        R = len(pole)
    while L < R:
        prostredni = (L+R)//2
        x = pole[prostredni]
        if x < hledane:
            L = prostredni + 1
        elif x > hledane:
            R = prostredni
        else:
            return prostredni
    return -1

# Zavolání:
print bin_vyhled([1,2,5,7,12,16,42], 8)
```

Další aplikace

Další typickou aplikací postupu rozděl a panuj je například třídění posloupnosti pomocí *Mergesortu*. Ten v základu funguje tak, že každou posloupnost, kterou dostane k setřídění, rozdělí na poloviny a každou z nich setřídí rekurzivním zavoláním sebe sama. Zastaví se ve chvíli, kdy třídí posloupnost délky jedna. Pak jen v každém kroku ze dvou setříděných menších posloupností vyrobí jejich sléváním setříděnou posloupnost dvojnásobné délky.

Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[33]

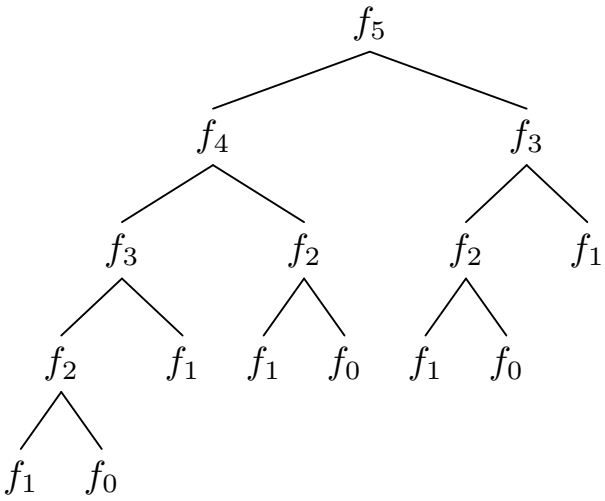
Předpočítané mezivýsledky

Motivací k této kapitole je následující motto: „Proč počítat něco vícekrát, když nám to stačí spočítat jednou a zapamatovat si to?“.

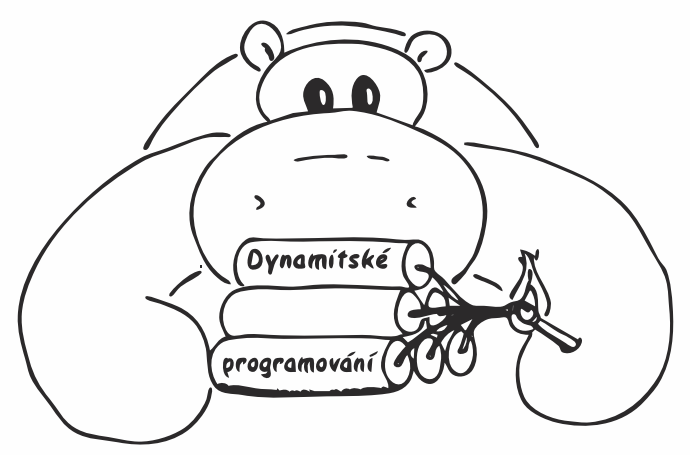
Velmi často se totiž setkáváme s tím, že něco počítáme stále dokola. Jako příklad si můžeme vzít naší rekurzivní implementaci počítání Fibonacciho čísel z kapitolky Rekurze.

Když se podíváme na výpočet čísla `fib(5)`, tak pro něj voláme `fib(4)` a `fib(3)`, `fib(4)` volá `fib(3)` a `fib(2)`, `fib(3)` volá `fib(2)` a `fib(1)` a tak dále. Všimli jste si, kolikrát se nám tyhle výpočty opakují? Některá Fibonacciho čísla spočteme totiž zbytečně mnohokrát.

[33] `http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj`



Kdybychom si je namísto opakovaného počítání někde pamatovali, mohli bychom pak odpověď na dotaz na již vypočtené číslo vytáhnout jako králíka z klobouku v konstantním čase. Zavedením jednoho globálního pole, ve kterém si tyto hodnoty pro jednotlivá n budeme pamatovat, nám sníží časovou složitost z $\mathcal{O}(2^n)$ na pěkných $\mathcal{O}(n)$. Takovému postupu se obecně říká *dynamické programování*.



Dynamické programování

Nejprve uveďme na pravou váhu výraz „dynamické“ v názvu. Nevystihuje tak úplně podstatu této techniky a jeho historické pozadí je celkem složité, avšak dnes je tento název již tak zažitý, že se už pravděpodobně nezmění.

Slovo „dynamické“ částečně odkazuje na to, že se dynamicky (za běhu programu) postupně staví řešení jednodušších problémů, která jsou následně použita pro řešení složitějších. Jeho hlavní podstatou je tedy ukládání a opětovné použití již jednou vypočtených údajů.

Hodí se na úlohy, které se dají dělit na podúlohy, které jsou si podobné a mohou se opakovat. Výsledky takovýchto podúloh si poté ukládáme a při dotazu na stejnou podúlohu vrátíme jen uložený výsledek a výpočet již neprovádíme.

Pro další prohloubení znalostí můžete na našem webu nahlédnout do další kuchařky, tentokrát nesoucí (překvapivě) název dynamické programování.[34]

Prefixové součty

Velmi často se nám však hodí si ještě před samotným výpočtem předvypočítat a uložit nějaké hodnoty, které poté použijeme.

Představme si například problém nalezení souvislého úseku s největším součtem v nějaké posloupnosti kladných i záporných čísel. Že to není úplně jednoduchý příklad, si ukažme na této posloupnosti: $1, -2, 4, 5, -1, -5, 2, 7$

Máme zde dvě ryze kladné souvislé posloupnosti, každou se součtem 9 ($4, 5$ a $2, 7$). Ale přesto je výhodnější vzít i nějaké záporné hodnoty a vytvořit tak souvislou posloupnost o součtu 12 (zkuste ji nalézt).

Tedy by nás mohlo napadnout, že prostě zkusíme vzít všechny možné začátky a všechny možné konce, to nám dává $\mathcal{O}(n^2)$ možných posloupností (máme n možných začátků a ke každému z nich řádově n možných konců), pro každou posloupnost si spočteme součet (to zvládneme v $\mathcal{O}(n)$) a budeme si pamatovat ten největší nalezený. Celý náš postup tak trvá $\mathcal{O}(n^3)$.

To není pro takhle jednoduchou úlohu zrovna ten nejpěknější čas, zkusme ho zlepšit. Ukážeme si, jak vypočítat součet libovolné posloupnosti v konstantním čase. Celý princip je vlastně až kouzelně jednoduchý, ale zároveň velmi mocný. Na začátku výpočtu si do pomocného pole P stejné délky jako posloupnost na vstupu (té říkejme S) uložíme takzvané *prefixové součty: i-tý* prefixový součet je součet prvních $i + 1$ prvků S, neboli $P[i] = S[0] + S[1] + \ldots + S[i]$.

Pro náš ukázkový případ a pro vstupní pole označené S by to dopadlo takto:

i	-1	0	1	2	3	4	5	6	7
$S[i]$		1	-2	4	5	-1	-5	2	7
$P[i]$	0	1	-1	3	8	7	2	4	11

Pole prefixových součtů umíme získat v lineárním čase – prostě jen od začátku procházíme vstupní pole, počítáme si průběžný součet a ten zapisujeme.

[34] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

Součet libovolného úseku $a \ldots b$ pak získáme v konstantním čase jako prefixový součet od začátku do indexu b minus prefixový součet od začátku do indexu a. Zapsáno programově to pak je: `soucet = P[b] - P[a-1];`

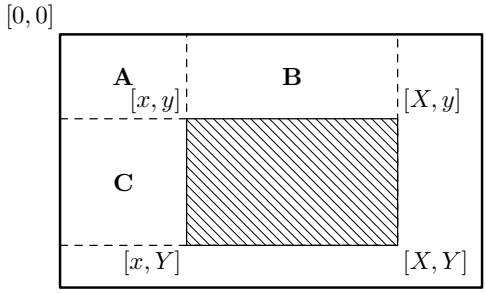
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas, prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

Dvourozměrné prefixové součty

Prefixové součty se však dají zobecnit i do více rozměrů, ale princip je vždy stejný. Například dvourozměrné prefixové součty u matice fungují tak, že si předpočítáme součty podmatic začínajících levým vrchním políčkem a končící na indexu $[x, y]$.

Z toho je vidět, že prefixový součet zpravidla obsadí stejně velký prostor jako původní data, v tomto případě tedy budeme mít matici hodnot prefixových součtů končících na zadaných souřadnicích. Jak ale získat součet nějaké podmatice, která se nachází někde „uprostřed“ naší matice?

Použijeme stejný princip, jako u jednorozměrného případu: Přičteme větší část, kterou chceme započítat, a odečteme od ní části, které započítat nechceme. Pro případ podmatice začínající vlevo nahoře na pozici $[x, y]$ a končící napravo dole na $[X, Y]$ to ilustruje následující obrázek:



Nejdříve přičteme celý prefixový součet končící na pozici $[X, Y]$. Tím jsme ale započítali i části A, B a C z obrázku, které započítat nechceme. Tak odečteme prefixové součty končící na indexech $[X, y]$ a $[x, Y]$. Ale pozor, teď jsme odečetli jednou $A + B$ a jednou $A + C$, tedy část A (prefixový součet končící na pozici $[x, y]$) jsme odečetli dvakrát, musíme ji proto ještě jednou přičíst.

Celý vzorec tedy vypadá takto:

```
soucet = P[X,Y] - P[x,Y] - P[X,y] + P[x,y];
```

Tento princip přičítání a odečítání se dá zobecnit i pro libovolné vyšší rozměry, ale chce to již trošku představivosti, co se má přičíst a kolikrát. Říká se tomu také *princip inkluze a exkluze* a najde použití nejen u vícerozměrných prefixových součtů.

Vyvážení délky předvýpočtu a hlavního výpočtu

Správně vyvážit, kolik času můžeme věnovat na předvýpočet a kolik času na hlavní výpočet, je velice důležitá věc a spousta i zkušenějších řešitelů v tom občas chybuje. Přitom to při troše počítání není vůbec nic složitého.

Jako první je potřeba vědět, kolikrát nám předvýpočet během běhu programu pomůže. Předvýpočtem si totiž vybudujeme za nějaký čas určitou datovou strukturu, pomocí které pak dokážeme rychle odpovídat na zadané dotazy.

Označme si počet takovýchto dotazů, které program za běhu dostane, jako Q. Buď to může být hodnota přímo ze zadání typu „Zkonstruujte datovou strukturu pro n hodnot, která zvládne rychle odpovídat na dotazy daného typu, a očekávejte řádově m dotazů“, nebo se může jednat o nějaký interní dotaz v rámci běhu programu (příklad interního dotazu je například výše uvedený algoritmus na hledání souvislé podposloupnosti s co největším součtem, který se za běhu ptal na součty nějakých podintervalů).

Dále si označme jako $\mathcal{O}_p$ čas, který nám zabere předvýpočet a jako $\mathcal{O}_q$ čas, který nám ušetří každý předvypočítaný dotaz. Celkový čas, který ušetříme, je pak vlastně $Q \cdot \mathcal{O}_q$. Pokud je tento čas řádově větší než $\mathcal{O}_p$, pak má předvýpočet smysl.

Naopak nemá smysl trávit předvýpočtem řádově více času, než by trval samotný výpočet bez použití předpočítaných hodnot.

Jako příklad uvažujme problém o velikosti n, u kterého máme tři možnosti, které můžeme zvolit. Můžeme buď předvýpočet úplně vynechat a na každý dotaz odpovídat v čase $\mathcal{O}(n)$, nebo provést předvýpočet v čase $\mathcal{O}(n \log n)$ a poté odpovídat na každý dotaz v čase $\mathcal{O}(\log n)$, nebo provést předvýpočet v čase $\mathcal{O}(n^2)$ a pak odpovídat v čase $\mathcal{O}(1)$ na dotaz.

Kdy se nám co hodí?

- Pokud bychom dostali jen jeden dotaz, nemá smysl si cokoliv předpočítávat a odpovíme jednou v čase $\mathcal{O}(n)$.
- Pokud bude dotazů řádově n, má smysl použít první předvýpočet. Pak budeme mít čas na předvýpočet i na samotný výpočet $\mathcal{O}(n \log n)$, což je optimum.
- Naopak pokud by dotazů bylo řádově n^2 nebo víc, tak se nám již první předvýpočet nevyplatí, dostali bychom se totiž na čas $\mathcal{O}(n^2 \log n)$. Zde se hodí použít druhý, delší předvýpočet a pak se dostat na časovou složitost $\mathcal{O}(n^2 + n^2 \cdot 1) = \mathcal{O}(n^2)$.

Hladové algoritmy

Věřte nebo ne, ale i počítač se někdy cítí hladový. Po namáhavé práci mu můžeme dopřát to potěšení, aby si ukousl co největší kus dat. A ukážeme, že někdy je to i ku prospěchu. Řeč bude o *hladových algoritmech*.

Takovými algoritmy rozumíme ty, které hledají řešení celé úlohy po jednotlivých krocích a splňují následující dvě podmínky:

- V každém kroku zvolí lokálně nejlepší řešení.
- Provedené rozhodnutí již nikdy neodvolává (tedy nebacktrackuje).

Lokálně nejlepší řešení je takové, které v aktuálním kroku vybere tu možnost, která nám na tomto místě nejvíce pomůže (bez jakéhokoliv ohledu na globální stav). Může to být třeba nejvyšší hodnota, nebo nejkratší cesta v grafu.

Pokud ale od hladového algoritmu chceme, aby nám našel globálně nejlepší řešení, musí naše úloha splnit předpoklad, že si výběrem lokálně nejlepšího řešení nezhoršíme to globální. Tento předpoklad se nedá formulovat obecně a je nutné se nad ním zamyslet zvlášť u každé úlohy.

Příklady hladových algoritmů

První hladovou úlohou bude (jak jinak) automat na jídlo vracející mince. Automat by měl vracet peníze nazpět tak, aby vrátil daný obnos v co možná nejmenším počtu mincí. Pro náš měnový systém (máme mince hodnot 1, 2, 5, 10, 20 a 50 Kč) lze tuto úlohu řešit hladovým algoritmem – v každém kroku algoritmu vrátíme tu největší minci, kterou můžeme (tedy pro vrácení 42 Kč to bude $42 = 20 + 20 + 2$ Kč).

Měnové systémy většiny států jsou postavené tak, aby fungovaly takto pěkně, neplatí to ale obecně. Zkusme si vrátit 42 Kč, pokud bychom měli jen mince hodnoty 20, 10 a 4 Kč. Správným řešením je $42 = 20 + 10 + 4 + 4 + 4$ Kč, hladový algoritmus by ale zkusil vrátit $42 = 20 + 20 + \ldots$ a tady by selhal.

Dále se velmi často dají hladovým algoritmem řešit nějaké úlohy přidávání nebo odebírání skupin prvků. Typickým příkladem je třeba rozvržení naplánovaných přednášek do učeben. Seřadíme si začátky přednášek podle času a postupně bereme jednu za druhou a umísťujeme je do volných učeben s nejnižším číslem.

Tím jsme si určitě nic nerozbili, protože v nějaké učebně přednáška být musí. Určitě budeme potřebovat tolik učeben, kolik je maximálně přednášek v jeden čas, a díky tomu si umístěním přednášky do nějaké učebny nezablokujeme místo pro jinou přednášku, jelikož nám vždy zbude dostatek volných učeben.

Kdybychom ale naopak měli pevně daný počet učeben a chtěli jsme do nich umístit co možná nejvíce přednášek, tak se již nejedná o úlohu řešitelnou hladovým algoritmem, v takovém případě je potřeba zvolit nějaký chytřejší postup.

Závěr

Doufám, že jste si z tohoto rozsáhlého textu odnesli nějaké nové znalosti a poznatky, které vám pomohou nejen v řešení KSP.

Pokud jste začínajícími řešiteli, zkuste s pomocí kuchařky vyřešit několik lehčích úloh a jejich řešení poslat – nově nabyté znalosti je totiž nejlepší co nejdříve protrénovat. Nic si nedělejte z toho, pokud napoprvé nevyřešíte všechno,

s postupným zkoušením se budou vaše znalosti jen zlepšovat. Zkušenější řešitelé možná v kuchařce nalezli nějaké ujasnění pojmů, či si některé techniky osvěžili.

A pokud tento text považujete za dobrý, budeme jen rádi, pokud ho doporučíte svým kamarádům a spolužákům, kteří chtějí s programováním začít.

Úvodním kurzem vaření podle kuchařky vás provedl

Jirka Setnička

Kuchařka druhé série – vyhledávací stromy

V kuchařce první série jsme probrali základní způsoby ukládání dat v počítači, tzv. datové struktury, a také často používané programátorské techniky. Konkrétně jsme se naučili udržovat čísla nebo jiné objekty v poli, ve spojovém seznamu, v grafu nebo ve stromu. Ukázali jsme si rekurzi a její využití v backtrackingu (prostém zkoušení všech možností). Dále jsme nakouli pod pokličku dalším technikám: rozděl a panuj, dynamickému programování, hladovým algoritmům a pár dalším.

Nyní se podíváme podrobněji na binární vyhledávání, které bylo rovněž minule představeno, a pokusíme se ho vylepšit, abychom mohli průběžně měnit data, v nichž vyhledáváme. Zdá-li se vám to na jednu kuchařku málo, zvlášť v porovnání s tou minulou, věřte, že problém není jednoduchý, ale zajímavý a řešení jsou navíc v praxi často používána.

Nejprve však zopakujme binární vyhledávání.

Binární vyhledávání

Stejně jako minule máme obrovské pole setříděných záznamů, třeba identifikačních čísel studentů nejmenované univerzity (záznamy však nemusí být čísla, stačí, když jsou navzájem porovnatelné). Naším úkolem je najít záznam z v poli s N záznamy $x_1 < x_2 < \ldots < x_N$.

Při použití binárního vyhledávání neboli půlení intervalu se podíváme na prostřední záznam x_m a porovnáme s ním naše z. Pokud $z < x_m$, víme, že se z nemůže vyskytovat „napravo“ od x_m, protože tam jsou všechny záznamy větší než x_m a tím spíše než z. Analogicky pokud $z > x_m$, nemůže se z vyskytovat v první polovině pole. V obou případech nám zbude jedna polovina a v ní budeme pokračovat stejným způsobem. Tak budeme postupně půlit interval, ve kterém se z může nacházet, až buďto z najdeme nebo vyloučíme všechny prvky, kde by mohlo být.

Tento algoritmus můžeme naprogramovat buďto rekurzivně nebo pomocí cyklu, v němž si budeme udržovat interval $\langle l, r \rangle$, ve kterém se hledaný prvek ještě může nacházet. My si ukážeme v jazyce C přístup s cyklem:

```
int bin_najdi(int z) {
    int levy, pravy, median;
    // interval, ve kterém hledáme
    levy = 0; pravy = N;
    // dokud interval ještě není prázdný
    while (levy <= pravy) {
        median = (levy+pravy)/2;
        // hledaná hodnota je vlevo
        if (z < x[median])
            pravy = median-1;
        // je vpravo
```

```
            else if (z > x[median])
                levy = median+1;
            else // našli jsme ji
                return median;
        }
        return -1; // hledaná hodnota nebyla nikde
    }
```

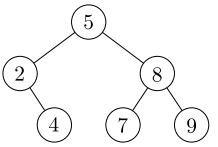

Samozřejmě bychom při vyhledávání záznamu mohli být ještě chytřejší. Víme-li třeba, že čísla jsou z rozsahu 1 až 1000 a dostaneme číslo 900, můžeme se napřed podívat do devíti desetin pole místo do poloviny. Obecně se tedy snažíme odhadovat, kde bude záznam v rámci pole podle jeho hodnoty. Tomuto přístupu se říká *interpolační vyhledávání* a v průměru je lepší než binární (průměrná časová složitost je $\mathcal{O}(\log \log N)$), byť v nejhorším případě je lineární.

Binární vyhledávání je velmi rychlé, pokud máme možnost si data předem setřídit. Jakmile ale potřebujeme za běhu programu přidávat a odebírat záznamy, se zlou se potážeme. Buďto budeme mít záznamy uložené v poli a pak nezbývá než při zatřiďování nového prvku ostatní „rozhrnout“, což může trvat až N kroků, anebo si je budeme udržovat v nějakém seznamu, do kterého dokážeme přidávat v konstantním čase, jenže pak pro změnu nebudeme při vyhledávání schopni najít tolik potřebnou polovinu.

Zkusme ale provést jednoduchý myšlenkový pokus:

Vyhledávací stromy

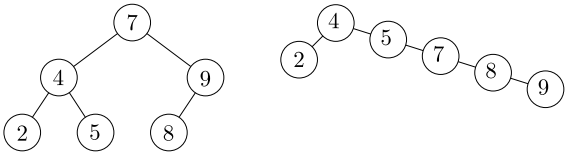
Představme si, jakými všemi možnými cestami se může v našem poli binární vyhledávání ubírat. Na začátku porovnáváme s prostředním prvkem a podle výsledku se vydáme jednou ze dvou možných cest (nebo rovnou zjistíme, že se jedná o hledaný prvek, ale to není moc zajímavý případ). Na každé cestě nás zase čeká porovnání se středem příslušného intervalu a to nás opět pošle jednou ze dvou dalších cest atd. To si můžeme přehledně popsat pomocí stromu:



Jeden vrchol stromu prohlásíme za kořen a ten bude odpovídat celému poli (a jeho prostřednímu prvku). K němu budou připojené vrcholy obou polovin pole (opět obsahující příslušné prostřední prvky) a tak dále. Ovšem jakmile známe tento strom, můžeme náš půlící algoritmus provádět přímo podle stromu (ani k tomu nepotřebujeme vidět původní pole a umět v něm hledat poloviny): začneme v kořeni, porovnáme a podle výsledku se buďto přesuneme do levého nebo pravého podstromu, a tak dále. Každý průběh algoritmu bude tedy odpovídat nějaké cestě z kořene stromu do hledaného vrcholu.

Teď si ale všimněte, že aby hledání hodnoty podle stromu fungovalo, strom vůbec nemusel vzniknout půlením intervalu – stačilo, aby v každém vrcholu platilo, že všechny hodnoty v levém podstromu jsou menší než tento vrchol a naopak

hodnoty v pravém podstromu větší. Hledání v témže poli by také popisovaly následující stromy (např.):



Hledací algoritmus podle jiných stromů samozřejmě už nemusí mít pěknou logaritmickou složitost (kdybychom hledali podle „degenerovaného“ stromu z pravého obrázku, trvalo by to dokonce lineárně). Důležité ale je, že takovéto stromy se dají poměrně snadno modifikovat a že je při troše šikovnosti můžeme udržet dostatečně podobné ideálnímu půlení intervalu. Pak bude hloubka stromu stále $\mathcal{O}(\log N)$, tím pádem i časová složitost hledání, a jak za chvilku uvidíme, i mnohých dalších operací.

Definice

Zkusme si tedy pořádně nadefinovat to, co jsme právě vymysleli:

Binární vyhledávací strom (podomácku BVS) je buď prázdná množina nebo *kořen* obsahující jednu hodnotu a mající dva *podstromy* (levý a pravý), což jsou opět BVS, ovšem takové, že všechny hodnoty uložené v levém podstromu jsou menší než hodnota v kořeni, a ta je naopak menší než všechny hodnoty uložené v pravém podstromu.

Úmluva: Pokud x je kořen a L_x a R_x jeho levý a pravý podstrom, pak kořenům těchto podstromů (pokud nejsou prázdné) budeme říkat *levý a pravý syn* vrcholu x a naopak vrcholu x budeme říkat *otec* těchto synů. Pokud je některý z podstromů prázdný, pak vrchol x příslušného syna nemá. Vrcholu, který nemá žádné syny, budeme říkat *list* vyhledávacího stromu. Všimněte si, že pokud x má jen jediného syna, musíme stále rozlišovat, je-li to syn levý nebo pravý, protože potřebujeme udržet správné uspořádání hodnot. Také si všimněte, že pokud známe syny každého vrcholu, můžeme již rekonstruovat všechny podstromy.

Každý BVS také můžeme popsat velmi jednoduchou strukturou v paměti:

```
struct vrchol {
    struct vrchol *levy, *pravy; // synové
    int x;                       // hodnota ve vrcholu
};
```

Pokud některý ze synů neexistuje, zapíšeme do příslušné položky hodnotu `NULL`.

Hledání

V řeči BVS můžeme přeformulovat náš vyhledávací algoritmus takto:

```
/* Dostane kořen stromu a hodnotu. Vrátí vrchol,
   kde se nachází, nebo NULL, není-li. */
struct vrchol *strom_najdi(struct vrchol *v,
                           int x)
{
    while (v != NULL && v->x != x) {
        if (x < v->x)
            v = v->levy;
        else
            v = v->pravy;
    }
    return v;
}
```

Funkce `strom_najdi` bude pracovat v čase $\mathcal{O}(h)$, kde h je hloubka stromu, protože začíná v kořeni a v každém průchodu cyklem postoupí o jednu hladinu níže.

Vkladání

Co kdybychom chtěli do stromu vložit novou hodnotu (aniž bychom se teď starali o to, zda tím strom nemůže degenerovat)? Stačí zkusit hodnotu najít a pokud tam ještě nebyla, určitě při hledání narazíme na odbočku, která je `NULL`. A přesně na toto místo připojíme nově vytvořený vrchol, aby byl správně uspořádán vzhledem k ostatním vrcholům (že tomu tak je, plyne z toho, že při hledání jsme postupně vyloučili všechna ostatní místa, kde nová hodnota být nemohla). Naprogramujeme opět snadno, tentokráte si ukážeme rekurzivní zacházení se stromy:

```
/* Dostane kořen stromu a hodnotu ke vložení,
   vrátí nový kořen. */
struct vrchol *strom_vloz(struct vrchol *v,
                          int x)
{
    if (v == NULL) { // prázdný strom
        // založíme nový kořen
        v = malloc(sizeof(struct vrchol));
        v->levy = v->pravy = NULL;
        v->x = x;
    } else if (x < v->x) // vkládáme vlevo
        v->levy = strom_vloz(v->levy, x);
    else if (x > v->x)  // vkládáme vpravo
```

```
            v->pravy = strom_vloz(v->pravy, x);
        return v;
    }
```

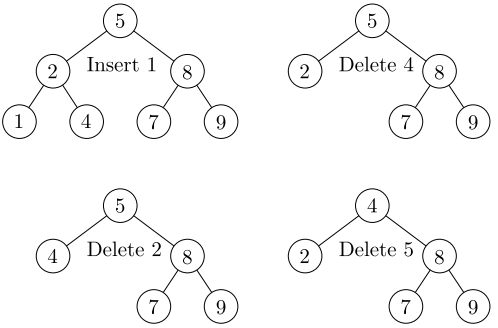
Mazání

Mazání bude o kousíček pracnější, musíme totiž rozlišit tři případy: Pokud je mazaný vrchol list, stačí ho vyměnit za `NULL`. Pokud má právě jednoho syna, stačí náš vrchol v ze stromu odstranit a syna přepojit k otci v. A pokud má syny dva, najdeme největší hodnotu v levém podstromu (tu najdeme tak, že půjdeme jednou doleva a pak pořád doprava), umístíme ji do stromu namísto mazaného vrcholu a v levém podstromu ji pak smažeme (což už umíme, protože má 1 nebo 0 synů). Program následuje:


```
/* Parametry stejně jako strom_vloz */
struct vrchol *strom_vymaz(struct vrchol *v,
                           int x)
{
    struct vrchol *w, *ret = v;
    if (v == NULL) return v; // prázdný strom
    else if (x < v->x)  // hledáme x
        v->levy = strom_vymaz(v->levy, x);
    else if (x > v->x)
        v->pravy = strom_vymaz(v->pravy, x);
    else { // našli jsme x ... jaké má syny?
        if (v->levy == NULL
            && v->pravy == NULL) {  // žádné
            free(v);
            return NULL;
        } else if (v->levy == NULL) {
            // jen pravý syn
            ret = v->pravy;
            free(v);
            return ret;
        } else if (v->pravy == NULL) {
            // jen levý syn
            ret = v->levy;
            free(v);
            return ret;
        } else { // má oba dva syny
            w = v->levy; // hledáme max(L)
            while (w->pravy != NULL)
                w = w->pravy;
            v->x = w->x; // prohazujeme
```

```
            // mažeme původní max(L)
            v->levy = tree_del(v->levy, w->x);
        }
    }
    return v;
}
```


Když do stromu z našeho prvního obrázku zkusíme přidávat nebo z něj odebírat prvky, dopadne to takto:



Jak vkládání, tak mazání opět budou trvat $\mathcal{O}(h)$, kde h je hloubka stromu. Ale pozor, jejich používáním může h nekontrolovatelně růst (v závislosti na počtu prvků ve stromě).

Cvičení

- Zkuste najít nějaký příklad, kdy h dosáhne až N – při postupném budování stromu operacemi vkládání i při mazání ze stromu hloubky $\mathcal{O}(\log N)$.

Procházení stromu

Pokud bychom chtěli všechny hodnoty ve stromu vypsat, stačí strom rekurzivně projít a sama definice uspořádání hodnot ve stromu nám zajistí, že hodnoty vypíšeme ve vzestupném pořadí: nejdříve levý podstrom, pak kořen a pak podstrom pravý. Časová složitost je, jak se snadno nahlédne, lineární, protože strávíme konstantní čas vypisováním každého prvku a prvků je právě N. Program bude opět přímočarý:

```
void strom_ukaz(struct vrchol *v){
    if (v == NULL)
        return; // není co dál dělat
    printf("(");
    strom_ukaz(v->levy);
    printf(")\%d(",v->x);
    strom_ukaz(v->pravy);
```

```
        printf(")");
    }
```

Vyvážené stromy

S binárními stromy lze dělat všelijaká kouzla a prakticky všechny stromové algoritmy mají společné to, že jejich časová složitost je lineární v hloubce stromu. (Pravda, právě ten poslední je výjimka, leč všechny prvky rychleji než lineárně s N opravdu nevypíšeme.)

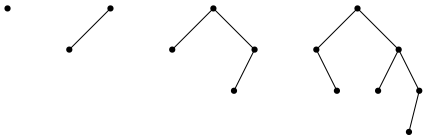
Jenže jak jsme viděli, neopatrným insertováním a deletováním prvků mohou snadno vznikat všelijaké degenerované stromy, které mají lineární hloubku. Abychom tomu zabránili, musíme stromy *vyvažovat.* To znamená definovat si nějaké šikovné omezení na tvar stromu, aby hloubka byla vždy $\mathcal{O}(\log N)$. Možností je mnoho, my uvedeme jen ty nejdůležitější:

Dokonale vyvážený budeme říkat takovému stromu, ve kterém pro každý vrchol platí, že počet vrcholů v jeho levém a pravém podstromu se liší nejvýše o jedničku. Takové stromy kopírují dělení na poloviny při binárním vyhledávání, a proto mají vždy logaritmickou hloubku. Jediné, čím se liší, je, že mohou zaokrouhlovat na obě strany, zatímco náš půlící algoritmus zaokrouhloval polovinu vždy dolů, takže levý podstrom nemohl být nikdy větší než pravý.

Z toho také plyne, že se snadnou modifikací půlícího algoritmu dá dokonale vyvážený BVS v lineárním čase vytvořit ze setříděného pole. Bohužel se ale při Insertu a Deletu nedá v logaritmickém čase strom znovu vyvážit.

AVL stromy

Zkusíme tedy vyvažovací podmínku trochu uvolnit a vyžadovat, aby se u každého vrcholu lišily o jedničku nikoliv velikosti podstromů, nýbrž pouze jejich hloubky. Takovým stromům se říká *AVL stromy* a mohou vypadat třeba takto:



Každý dokonale vyvážený strom je také AVL stromem, ale jak je vidět na předchozím obrázku, opačně to platit nemusí. To, že hloubka AVL stromů je také logaritmická, proto není úplně zřejmé a zaslouží si to trochu dokazování:

Věta: AVL strom o N vrcholech má hloubku $\mathcal{O}(\log N)$.

Důkaz: Označme A_d nejmenší možný počet vrcholů, jaký může být v AVL stromu hloubky d. Snadno zjistíme, že $A_1 = 1$, $A_2 = 2$, $A_3 = 4$ a $A_4 = 7$ (příslušné minimální stromy najdete na předchozím obrázku). Navíc platí, že $A_d = 1 + A_{d-1} + A_{d-2}$, protože každý minimální strom hloubky d musí mít kořen a 2 podstromy, které budou opět minimální, protože jinak bychom je mohli

vyměnit za minimální a tím snížit počet vrcholů celého stromu. Navíc alespoň jeden z podstromů musí mít hloubku $d-1$ (protože jinak by hloubka celého stromu nebyla d) a druhý hloubku $d-2$ (podle definice AVL stromu může mít $d-1$ nebo $d-2$, ale s menší hloubkou bude mít evidentně méně vrcholů).

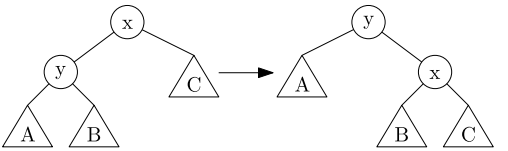
Spočítat, kolik přesně je A_d, není úplně snadné. Nám však postačí dokázat, že $A_d \geq 2^{d/2}$. To provedeme indukcí: Pro $d < 4$ to plyne z ručně spočítaných hodnot. Pro $d \geq 4$ je $A_d = 1 + A_{d-1} + A_{d-2} > 2^{(d-1)/2} + 2^{(d-2)/2} = 2^{d/2} \cdot (2^{-1/2} + 2^{-1}) > 2^{d/2}$ (součet čísel v závorce je ≈ 1.207).

Jakmile už víme, že A_d roste s d alespoň exponenciálně, tedy že $\exists c : A_d \geq c^d$, důkaz je u konce: Máme-li AVL strom T na N vrcholech, najdeme si nejmenší d takové, že $A_d \leq N$. Hloubka stromu T může být maximálně d, protože jinak by T musel mít alespoň A_{d+1} vrcholů, ale to je více než N. A jelikož A_d rostou exponenciálně, je $d \leq \log_c N$, čili $d = \mathcal{O}(\log N)$. *Q.E.D.*

AVL stromy tedy vypadají nadějně, jen stále nevíme, jak provádět Insert a Delete tak, strom zůstal vyvážený. Nemůžeme si totiž dovolit strukturu stromu měnit libovolně – stále musíme dodržovat správné uspořádání hodnot. K tomu se nám bude hodit zavést si nějakou množinu operací, o kterých dokážeme, že jsou korektní, a pak strukturu stromu měnit vždy jen pomocí těchto operací. Budou to rotace a dvojrotace.

Rotace

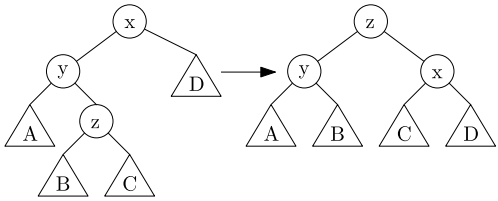
Rotací binárního stromu (respektive nějakého podstromu) nazveme jeho „překořenění“ za některého ze synů kořene. Místo formální definice ukažme raději obrázek (trojúhelník zastupuje podstrom, který může být v některých případech i prázdný):



Strom jsme překořenili za vrchol y a přepojili jednotlivé podstromy tak, aby byly vzhledem k x a y opět uspořádané správně (všimněte si, že je jen jediný způsob, jak to udělat). Jelikož se tím okolí vrcholu y „otočilo“ po směru hodinových ručiček, říká se takové operaci *rotace doprava.* Inverzní operaci (tj. překořenění za pravého syna kořene) se říká *rotace doleva* a na našem obrázku odpovídá přechodu zprava doleva.

Dvojrotace

Také si nakreslíme, jak to dopadne, když provedeme dvě rotace nad sebou lišící se směrem (tj. jednu levou a jednu pravou nebo opačně). Tomu se říká *dvojrotace* a jejím výsledkem je překořenění podstromu za vnuka kořene připojeného „cikcak“. Raději opět předvedeme na obrázku:



Znaménka

Při vyvažování se nám bude hodit pamatovat si u každého vrcholu, v jakém vztahu jsou hloubky jeho podstromů. Tomu budeme říkat *znaménko* vrcholu a bude buďto 0, jsou-li oba stejně hluboké, − pro levý podstrom hlubší a + pro pravý hlubší. V textu budeme znaménka, respektive vrcholy se znaménky značit $\odot$, $\ominus$ a $\oplus$.

Pokud celý strom zrcadlově obrátíme (prohodíme levou a pravou stranu), znaménka se změní na opačná ($\oplus$ a $\ominus$ se prohodí, $\odot$ zůstane). Toho budeme často využívat a ze dvou zrcadlově symetrických situací popíšeme jenom jednu s tím, že druhá se v algoritmu zpracuje symetricky.

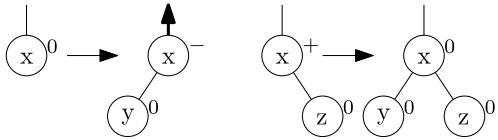
Často také budeme potřebovat nalézt otce nějakého vrcholu. To můžeme zařídit buďto tak, že si do záznamů popisujících vrcholy stromu přidáme ještě ukazatele na otce a budeme ho ve všech operacích poctivě aktualizovat, anebo využijeme toho, že jsme do daného vrcholu museli někudy přijít z kořene, a celou cestu z kořene si zapamatujeme v nějakém zásobníku a postupně se budeme vracet.

Tím jsme si připravili všechny potřebné ingredience, tož s chutí do toho.

Vyvažování po Insertu

Když provedeme Insert tak, jak jsme ho popisovali u obecných vyhledávacích stromů, přibude nám ve stromu list. Pokud se tím AVL vyváženost neporušila, stačí pouze opravit znaménka na cestě z nového listu do kořene (všude jinde zůstala zachována). Pakliže porušila, musíme s tím něco provést, konkrétně ji šikovně zvolenými rotacemi opravit. Popíšeme algoritmus, který bude postupovat od listu ke kořeni a vše potřebné zařídí.

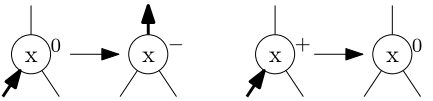
Nejprve přidání listu samotné:



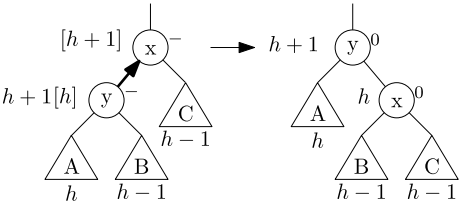
Pokud jsme přidali list (bez újmy na obecnosti levý, jinak vyřešíme zrcadlově) vrcholu se znaménkem $\odot$, změníme znaménko na $\ominus$ a pošleme o patro výš informaci o tom, že hloubka podstromu se zvýšila (to budeme značit šipkou).

Přidali-li jsme list k $\oplus$, změní se na $\odot$ a hloubka podstromu se nemění, takže můžeme skončit.

Nyní rozebereme případy, které mohou nastat na vyšších hladinách, když nám z nějakého podstromu přijde šipka. Opět budeme předpokládat, že přišla zleva; pokud zprava, vyřešíme zrcadlově. Pokud přišla do $\oplus$ nebo $\odot$, ošetříme to stejně jako při přidání listu:

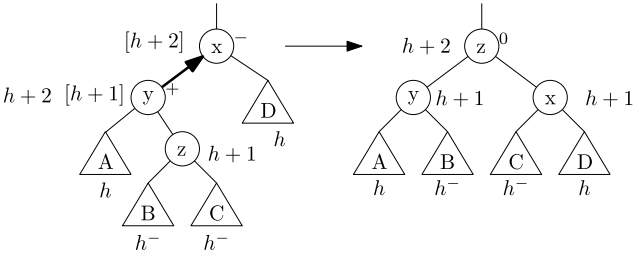


Pokud ale vrchol x má znaménko $\ominus$, nastanou potíže: levý podstrom má teď hloubku o 2 vyšší než pravý, takže musíme rotovat. Proto se podíváme o patro níž, jaké je znaménko vrcholu y pod šipkou, abychom věděli, jakou rotaci provést. Jedna možnost je tato (y je $\ominus$):



Tehdy provedeme jednoduchou rotaci vpravo. Jak to dopadne s hloubkami jsme přikreslili do obrázku – pokud si hloubku podstromu A označíme jako h, B musí mít hloubku $h-1$, protože y je $\ominus$, atd. Jen nesmíme zapomenout, že v x jsme ještě $\ominus$ nepřepočítali (vede tam přeci šipka), takže ve skutečnosti je jeho levý podstrom o 2 hladiny hlubší než pravý (původní hloubky jsme na obrázku naznačili [v závorkách]). Po zrotování vyjdou u x i y znaménka $\odot$ a celková hloubka se nezmění, takže jsme hotovi.

Další možnost je y jako $\oplus$:



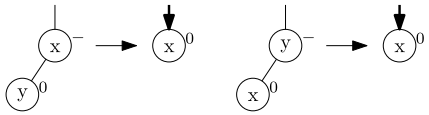
Tehdy se podíváme ještě o hladinu níž a provedeme dvojrotaci. (Nemůže se nám stát, že by z neexistovalo, protože jinak by v y nebylo $\oplus$.) Hloubky

opět najdete na obrázku. Jelikož z může mít libovolné znaménko, jsou hloubky podstromů B a C buďto h nebo $h-1$, což značíme h^-. Podle toho pak vyjdou znaménka vrcholů x a y po rotaci. Každopádně vrchol z vždy obdrží $\odot$ a celková hloubka se nemění, takže končíme.

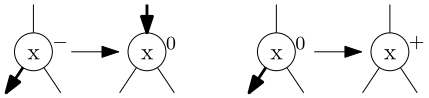
Poslední možnost je, že by y byl $\odot$, ale tu vyřešíme velmi snadno: všimneme si, že nemůže nastat. Kdykoliv totiž posíláme šipku nahoru, není pod ní $\odot$. (Kontrolní otázka: jak to, že $\oplus$ může nastat?)

Vyvažování po Deletu

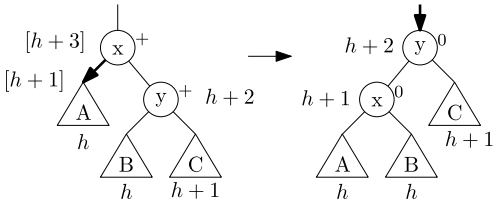
Vyvažování po Deletu je trochu obtížnější, ale také se dá popsat pár obrázky. Nejdříve opět rozebereme základní situace: odebíráme list (bez újmy na obecnosti (BÚNO) levý) nebo vnitřní vrchol s jediným synem (tehdy ale musí být jeho jediný syn listem, jinak by to nebyl AVL strom):



Šipkou dolů značíme, že o patro výš posíláme informaci o tom, že se hloubka podstromu snížila o 1. Pokud šipku dostane vrchol typu $\ominus$ nebo $\odot$, vyřešíme to snadno:

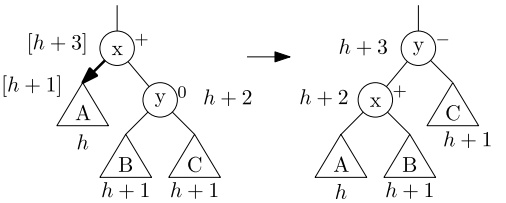


Problematické jsou tentokráte ty případy, kdy šipku dostane $\oplus$. Tehdy se musíme podívat na znaménko *opačného* syna a podle toho rotovat. První možnost je, že opačný syn má $\oplus$:



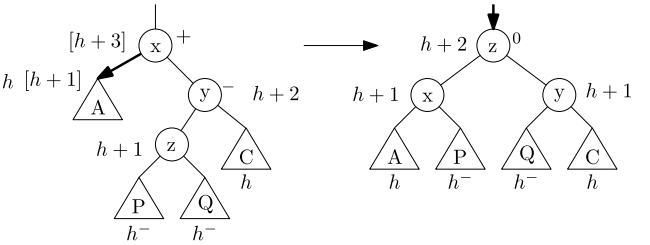
Tehdy provedeme rotaci vlevo, x i y získají nuly, ale celková hloubka stromu se sníží o hladinu, takže nezbývá, než poslat šipku o patro výš.

Pokud by y byl $\odot$:



Opět rotace vlevo, ale tentokráte se zastavíme, protože celková hloubka se nezměnila.

Poslední, nejkomplikovanější možnost je, že by y byl $\ominus$:



V tomto případě provedeme dvojrotaci (z určitě existuje, jelikož y je typu $\ominus$), vrcholy x a y obdrží znaménka v závislosti na původním znaménku vrcholu z a celý strom se snížil, takže pokračujeme o patro výš.

Happy end

Jak při Insertu, tak při Deletu se nám podařilo strom upravit tak, aby byl opět AVL stromem, a trvalo nám to lineárně s hloubkou stromu (konáme konstantní práci na každé hladině), čili stejně jako trvá Insert a Delete samotný. Jenže o AVL stromech jsme již dokázali, že mají hloubku vždy logaritmickou, takže jak hledání, tak Insert a Delete zvládneme v logaritmickém čase (vzhledem k aktuálnímu počtu prvků ve stromu).

Další typy stromů

AVL stromy samozřejmě nejsou jediný způsob, jak zavést stromovou datovou strukturu s logaritmicky rychlými operacemi. Jaké jsou další?

2-3-stromy nemají v jednom vrcholu uloženu jednu hodnotu, nýbrž jednu nebo dvě (a synové jsou pak 2 nebo 3, odtud název.) Přidáme navíc pravidlo, že všechny listy jsou na téže hladině. Hloubka vyjde logaritmická, vyvažování řešíme pomocí spojování a rozdělování vrcholů.

Červeno-černé stromy si místo znamének vrcholy barví. Každý je buďto červený nebo černý a platí, že nikdy nejsou dva červené vrcholy pod sebou a že na každé cestě z kořene do listu je stejný počet černých vrcholů. Hloubka je pak znovu logaritmická.

Po Insertu a Deletu barvy opravujeme rotováním a přebarvováním na cestě do kořene, jen je potřeba rozebrat podstatně více případů než u AVL stromů. (Za to jsme ale odměněni tím, že nikdy neděláme více než 2 rotace.) Počet případů k rozebrání lze omezit zpřísněním podmínek na umístění červených vrcholů – dvěma různým takovým zpřísněním se říká *AA-stromy* a *left-leaning červeno-černé stromy*.

Interpretujeme-li červené vrcholy jako rozšíření otcovského vrcholu o další hodnoty, pochopíme, že jsou červeno-černé stromy jen jiným způsobem záznamu 2-4-stromů. Proč se takový kryptický překlad dělá? S třemi potomky vrcholu a dvěma hodnotami se pracuje nešikovně.

V případě *splay stromů* nezavádíme žádnou vyvažovací podmínku, nýbrž definujeme, že kdykoliv pracujeme s nějakým vrcholem, vždy si jej vyrotujeme do kořene a pokud to jde, preferujeme dvojrotace. Takové operaci se říká Splay a dají se pomocí ní definovat operace ostatní: Find hodnotu najde a poté na ni zavolá Splay. Insert si nechá vysplayovat předchůdce nové hodnoty a vloží nový vrchol mezi předchůdce a jeho pravého syna. Delete vysplayuje mazaný prvek, pak uvnitř pravého podstromu vysplayuje minimum, takže bude mít jen jednoho syna a můžeme jím tedy nahradit mazaný prvek v kořeni.

Jednotlivé operace samozřejmě mohou trvat až lineárně dlouho, ale dá se o nich dokázat, že jejich *amortizovaná* složitost je vždy $\mathcal{O}(\log N)$. Tím chceme říci, že provést t po sobě jdoucích operací začínajících prázdným stromem trvá $\mathcal{O}(t \cdot \log N)$ (některé operace mohou být pomalejší, ale to je vykoupeno větší rychlostí jiných).

To u většiny použití stačí – datovou strukturu obvykle používáte uvnitř nějakého algoritmu a zajímá vás, jak dlouho běží všechny operace dohromady – a navíc je Splay stromy daleko snazší naprogramovat než nějaké vyvažované stromy. Mimo to mají Splay stromy i jiné krásné vlastnosti: přizpůsobují svůj tvar četnostem hledání, takže často hledané prvky jsou pak blíž ke kořeni, snadno se dají rozdělovat a spojovat, atd.

Treapy jsou randomizovaně vyvažované stromy: něco mezi stromem (tree) a haldou (heap). Každému prvku přiřadíme *váhu,* což je náhodné číslo z intervalu $\langle 0, 1\rangle$. Strom pak udržujeme uspořádaný stromově podle hodnot a haldově podle vah (všimněte si, že tím je jeho tvar určen jednoznačně, pokud tedy jsou všechny váhy navzájem různé, což skoro jistě jsou). Insert a Delete opravují haldové uspořádání velmi jednoduše pomocí rotací. Časová složitost v průměrném případě je $\mathcal{O}(\log N)$.

BB-α stromy nabízí zobecnění dokonalé vyváženosti jiným směrem: zvolíme si vhodné číslo α a vyžadujeme, aby se velikost podstromů každého vrcholu lišila maximálně α-krát (prázdné podstromy nějak ošetříme, abychom nedělili nulou; dokonalá vyváženost odpovídá $\alpha = 1$ (až na zaokrouhlování)). V každém vrcholu si budeme pamatovat, kolik vrcholů obsahuje podstrom, jehož je koře-

nem, a po Insertu a Deletu přepočítáme tyto hodnoty na cestě zpět do kořene a zkontrolujeme, jestli je strom ještě stále α-vyvážený.

Pokud ne, najdeme nejvyšší místo, ve kterém se velikosti podstromů příliš liší, a vše od tohoto místa dolů znovu vytvoříme algoritmem na výrobu dokonale vyvážených stromů. Ten, pravda, běží v lineárním čase, ale čím větší podstrom přebudováváme, tím to děláme méně často, takže vyjde opět amortizovaně $\mathcal{O}(\log N)$ na operaci.

Cvičení

- Jak konstruovat dokonale vyvážené stromy?
- Jak pomocí toho naprogramovat BB-α stromy?
- Najděte algoritmus, který k prvku v obecném vyhledávacím stromu najde jeho následníka, což je prvek s nejbližší vyšší hodnotou (zde předpokládejte, že ke každému prvku máte uložený ukazatel na jeho otce).

- Jak vypsat celý strom tak, že začnete v minimu a budete postupně hledat následníky? (I když nalezení následníka může trvat až $\mathcal{O}(h)$, všimněte si, že projití celého stromu přes následníky bude lineární.)
- Jak do vrcholů stromu ukládat různé pomocné informace, jako třeba počet vrcholů v podstromu každého vrcholu, a jak tyto informace při operacích se stromem (při Insertu, Deletu, rotaci) udržovat?
- Ukažte, že lze libovolný interval $\langle a, b\rangle$ rozložit na logaritmicky mnoho intervalů odpovídajících podstromům vyváženého stromu.
- Ukažte, že zkombinováním předchozích dvou cvičení lze odpovídat i na otázky typu „kolik si strom pamatuje hodnot ze zadaného intervalu“ v logaritmickém čase ...

Poznámky

- Představte si, že budujete binární vyhledávací strom vkládáním prvků v náhodném pořadí. Obecně nemusí být vyvážený, ale v průměru v něm bude možné vyhledávat v čase $\mathcal{O}(\log N)$. Žádný div: Stromy, které nám vzniknou, odpovídají přesně možným průběhům QuickSortu, který má průměrnou časovou složitost $\mathcal{O}(N \log N)$.
- Pokud bychom připustili, že se mohou vyskytnout dva stejné záznamy, budou stromy stále fungovat, jen si musíme dát o něco větší pozor na to, co všechno při operacích se stromem může nastat.
- Jakpak přišly AVL stromy ke svému jménu? Podle Adeľsona-Veľského a Landise, kteří je objevili.
- Rekurenci $A_d = 1 + A_{d-1} + A_{d-2}$, $A_1 = 1$, $A_2 = 2$ pro velikosti minimálních AVL stromů je samozřejmě možné vyřešit i přesně. Žádné překvapení se nekoná, objeví se totiž stará známá Fibonacciho čísla: $A_n = F_{n+2} - 1$.

Martin „Medvěd“ Mareš & Tomáš Valla

Kuchařka třetí série – haldy a Dijkstrův algoritmus

Dnešní povídání o algoritmech a datových strukturách se bude zabývat jedním z nejznámějších algoritmů: Dijkstrovým algoritmem pro hledání nejkratších cest v grafech. K tomu (a nejenom k tomu) se nám bude hodit šikovná datová strukturka zvaná halda, tak si předvedeme nejdříve ji.

Halda

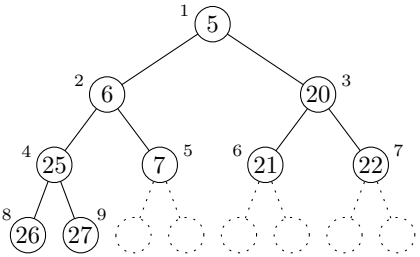
Halda je datová struktura pro uchovávání množiny čísel (či jakýchkoliv jiných prvků, na kterých máme definováno uspořádání, tj. umíme pro každou dvojici prvků říci, který z nich je menší). Tato datová struktura obvykle podporuje následující operace: Přidání nového prvku, nalezení nejmenšího prvku a odebrání nejmenšího prvku. My si ukážeme jednoduchou implementaci haldy, která bude při uložení N prvků potřebovat čas $\mathcal{O}(\log N)$ na přidání či odebrání jednoho prvku a $\mathcal{O}(1)$ (tj. konstantní) na zjištění hodnoty nejmenšího prvku.

Naše implementace bude vypadat následovně: Pokud halda obsahuje N prvků, uložíme její prvky do pole na pozice 1 až N. Prvek na pozici k bude mít dva *následníky,* a to prvky na pozicích $2k$ a $2k + 1$; samozřejmě, pokud je k velké, a tedy např. $2k + 1 > N$, má takový prvek jen jednoho či dokonce žádného následníka. Naopak prvek na pozici $\lfloor k/2 \rfloor$ nazveme *předchůdcem* prvku na pozici k. Ti z vás, kteří znají binární stromy, v tomto jistě rozpoznali způsob, jak v poli uchovávat úplné binární stromy (následníci jsou synové a předchůdci otcové v obvyklé stromové terminologii, prvek č. 1 je kořen stromu).

Prvky haldy však v poli neuchováváme v úplně libovolném pořadí. Chceme, aby platilo, že každý prvek je menší nebo roven všem svým následníkům. Naše halda tedy může vypadat např. takto:

1	2	3	4	5	6	7	8	9
5	6	20	25	7	21	22	26	27

Tomu odpovídá tento strom:



Z toho, co jsme si právě popsali, je jasné, že nejmenší prvek je uložen na pozici s indexem 1, a tedy můžeme snadno v konstantním čase zjistit jeho hodnotu. Ještě prozradíme, jak lze prvky do haldy rychle přidávat a odebírat:

Jestliže halda obsahuje N prvků, pak nový prvek, říkejme mu třeba x, přidáme na konec pole, tj. na pozici s indexem $N + 1$. Nyní x porovnáme s jeho předchůdcem. Pokud je jeho předchůdce menší, je vše v pořádku a jsme hotovi. V opačném případě x s jeho předchůdcem prohodíme. Tím jsme problém napravili, ale nyní může být x menší než jeho nový předchůdce. To lze napravit dalším prohozením a tak budeme pokračovat dále, než se buďto dostaneme do situace, kdy už je x větší nebo rovno svému předchůdci, nebo „vybublá“ až do kořene haldy, kde už žádného předchůdce nemá. Protože se v každém kroku pozice, na níž se prvek x právě nachází, zmenší alespoň na polovinu, provedeme dohromady nejvýše $\mathcal{O}(\log N)$ výměn, a tedy spotřebujeme čas $\mathcal{O}(\log N)$.

Odebírání nejmenšího prvku probíhá podobně: Prvek z poslední pozice (tj. z pozice N) přesuneme na pozici 1, tedy místo minima. Místo s předchůdci jej však porovnáme s jeho následníky a v případě, že je větší než některý z jeho následníků, opět je prohodíme (pokud je větší než oba následníci, prohodíme ho s menším z nich). A protože se nám v každém kroku index „bublajícího“ prvku v poli alespoň zdvojnásobí, opět spotřebujeme čas $\mathcal{O}(\log N)$.

Jako cvičení si rozmyslete, že v čase $\mathcal{O}(\log N)$ lze z haldy smazat dokonce libovolný prvek, pokud si ovšem pamatujeme, kde se v haldě nachází. Také můžeme prvek ponechat a jen změnit jeho hodnotu.

Ještě si předvedeme program:

```
var halda: array[1..MAX] of integer;
    N: integer; { počet prvků v haldě }

function nejmensi: integer;
begin
  nejmensi:=halda[1]
end;

procedure vloz(prvek: integer);
var i, x: integer;
begin
  N:=N+1; i:=N;
  halda[i]:=prvek;
  while (i>1) and (halda[i div 2]>halda[i])
  do begin
    x:=halda[i div 2];
    halda[i div 2]:=halda[i];
    halda[i]:=x;
    i:=i div 2
  end
end;
```

```
procedure smaz_nejmensi;
var i, j, x: integer;
begin
  halda[1]:=halda[N];
  N:=N-1; i:=1;
  while 2*i<=N do begin
    j:=i;
    if halda[j]>halda[2*i] then j:=2*i;
    if (2*i+1<=N) and (halda[j]>halda[2*i+1])
       then j:=2*i+1;
    if i=j then break;
    x:=halda[i]; halda[i]:=halda[j];
    halda[j]:=x;
    i:=j
  end
end;
```


HeapSort

Když už máme k dispozici haldu, můžeme pomocí ní například snadno třídit čísla. Máme-li N čísel, která chceme setřídit, vytvoříme si z nich nejprve haldu o N prvcích (například postupným vkládáním do prázdné haldy), načež z ní budeme postupně N-krát odebírat nejmenší prvek. Tím získáme prvky původního pole v rostoucím pořadí. Celkově provedeme N vložení, N nalezení minima a N smazání. To vše dohromady stihneme v čase $\mathcal{O}(N \log N)$.

Než si ukážeme program, přidáme ještě dva triky, které nám implementaci značně usnadní. Předně si vše uložíme do jednoho pole – to bude při plnění haldy obsahovat na svém začátku haldu a na konci zbytek vstupního pole, přitom zbytek pole se bude postupně zmenšovat a uvolňovat tak místo haldě; naopak v druhé polovině algoritmu budeme zmenšovat haldu a do volného prostoru ukládat setříděné prvky. K tomu se nám bude hodit získávat prvky v opačném pořadí, proto si upravíme haldu tak, aby udržovala nikoliv minimum, nýbrž maximum.

Druhý trik spočívá v tom, že nebudeme haldu vytvářet postupným vkládáním, nýbrž naopak zabubláváním prvků (podobným, jako děláme při mazání minima) od konce. Všimněte si, že takto také získáme správné nerovnosti mezi prvky a jejich následníky, a dokonce tak zvládneme celou haldu vytvořit v lineárním čase (proč to tak je, si zkuste dokázat sami, stačí si uvědomit, kolikrát zabubláváme které prvky). Zbytek třídění bohužel nadále zůstává $\mathcal{O}(N \log N)$.

Tomuto algoritmu se obvykle říká *HeapSort* (čili třídění haldou) a je jedním z mála známých rychlých třídících algoritmů, které nepotřebují pomocnou paměť.

```
type Pole = array[1..MAXN] of Integer;

procedure HeapSort(var A: Pole);
var i, x: integer;
   procedure bublej(m, i: integer);
   { zabublání prvku: m je velikost haldy,
   i je index zabublávaného prvku }
   var j, x: integer;
   begin
      while 2*i<=m do begin
         j:=2*i;
         if (j<m) and (A[j+1]>A[j]) then j:=j+1;
         if A[i]>=A[j] then break;
         x:=A[i]; A[i]:=A[j]; A[j]:=x;
         i:=j;
      end;
   end;
begin
   for i:=N div 2 downto 1 do bublej(N,i);
   { vybírej maximum }
   for i:=N downto 2 do begin
      x:=A[1]; A[1]:=A[i]; A[i]:=x;
      bublej(i-1, 1);
   end;
end;
```

Dijkstrův algoritmus

Nyní již konečně k slíbenému *Dijkstrovu algoritmu.* Tento algoritmus dostane orientovaný graf s hranami ohodnocenými nezápornými čísly (viz kuchařka o grafech)[35] a nalezne v něm nejkratší cestu mezi dvěma zadanými vrcholy. Ve skutečnosti tento algoritmus dělá o malinko více: Najde totiž nejkratší cesty z jednoho zadaného vrcholu do všech ostatních.

Nechť v_0 je vrchol grafu, ze kterého chceme určit délky nejkratších cest. Budeme si udržovat pole délek zatím nalezených cest z vrcholu v_0 do všech ostatních vrcholů grafu. Navíc u některých vrcholů budeme mít poznamenáno, že cesta nalezená do nich je už ta nejkratší možná. Takovým vrcholům budeme říkat *definitivní.* Na začátku inicializujeme v poli všechny hodnoty na ∞ kromě hodnoty odpovídající vrcholu v_0, kterou inicializujeme na 0 (délka nejkratší cesty z v_0 do v_0 je 0). V každém *kroku* algoritmu pak provedeme následující: Vybereme vrchol w, který ještě není definitivní, a mezi všemi takovými vrcholy je délka zatím nalezené cesty do něj nejkratší možná. Vrchol w prohlásíme za definitivní.

[35] http://ksp.mff.cuni.cz/viz/kucharky/grafy

Dále otestujeme, zda pro nějaký vrchol v cesta z vrcholu v_0 do w a pak po hraně z w do v není kratší, než zatím nalezená cesta z v_0 do v, a je-li tomu tak, upravíme délku zatím nalezené cesty do v. Toto provedeme pro všechny takové vrcholy v. Celý algoritmus skončí, pokud jsou už všechny vrcholy definitivní nebo všechny vrcholy, co nejsou definitivní, mají délku cesty rovnou ∞ (v takovém případě se graf skládá z více nesouvislých částí).

Předtím než dokážeme, že právě představený algoritmus opravdu nalezne délky nejkratších cest z vrcholu v_0, se zamysleme nad jeho časovou složitostí.

Pro každý z N vrcholů si délku dosud nalezené cesty uchováme v poli. Celý algoritmus provede nejvýše N kroků, protože v každém kroku nám přibyde jeden definitivní vrchol. Ten vybíráme jako minimum z délky aktuální cesty přes všechny dosud nedefinitivní vrcholy, kterých je $\mathcal{O}(N)$. V každém kroku musíme zkontrolovat tolik vrcholů v, kolik hran vede z vrcholu w. Počet takových změn pro všechny kroky dohromady je pak nejvýše $\mathcal{O}(M)$, kde M je počet hran vstupního grafu. Z toho vyjde časová složitost $\mathcal{O}(N^2 + M)$, čili $\mathcal{O}(N^2)$, jelikož M je nejvýše N^2. Tuto implementaci Dijkstrova algoritmu najdete na konci naší kuchařky.

K uchovávání délek dosud nalezených nejkratších cest můžeme ovšem použít haldu. Ta bude na začátku obsahovat N prvků a v každém kroku se počet jejích prvků sníží o jeden: Nalezneme a smažeme nejmenší prvek, to zvládneme v čase $\mathcal{O}(\log N)$, a případně upravíme délky nejkratších cest do sousedů právě zpracovávaného vrcholu. To pro každou hranu trvá rovněž $\mathcal{O}(\log N)$, celkově za všechny hrany tedy $\mathcal{O}(M \log N)$. Z toho vyjde celková časová složitost algoritmu $\mathcal{O}((N + M) \log N)$, a to je pro „řídké“ grafy (tedy grafy s $M \ll N^2$) výrazně lepší.

Vraťme se nyní k důkazu správnosti Dijkstrova algoritmu. Ukážeme, že po každém kroku algoritmu platí následující tvrzení: Nechť A je množina definitivních vrcholů. Pak délka dosud nalezené cesty z v_0 do v (v je libovolný vrchol grafu) je délka nejkratší cesty $v_0 v_1 \ldots v_k v$ takové, že všechny vrcholy $v_0, v_1, \ldots, v_k$ jsou v množině A. Tvrzení dokážeme indukcí dle počtu kroků algoritmu, které již proběhly. Tvrzení zřejmě platí před a po prvním kroku algoritmu. Nechť w je vrchol, který byl v předchozím kroku prohlášen za definitivní. Uvažme nejprve nějaký vrchol v, který je definitivní. Pokud $v = w$, tvrzení je triviální. V opačném případě ukážeme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w. Označme D délku cesty z v_0 do v přes vrcholy A bez vrcholu w. Protože v každém kroku vybíráme vrchol s nejmenším ohodnocením a ohodnocení vybraných vrcholů v jednotlivých krocích tvoří neklesající posloupnost (váhy hran jsou nezáporné!), tak délka cesty z v_0 do w přes vrcholy z A je alespoň D. Ale potom délka libovolné cesty z v_0 do v přes w používající vrcholy z A je alespoň D. Z volby D pak víme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w.

Nyní uvažme takový vrchol v, který není definitivní. Nechť $v_0 v_1 \dots v_k v$ je nejkratší cesta z v_0 do v taková, že všechny vrcholy $v_0, v_1, \dots, v_k$ jsou v množině A. Pokud $v_k = w$, pak jsme ohodnocení v změnili na délku této cesty v právě proběhlém kroku. Pokud $v_k \neq w$, pak $v_0 v_1, \dots, v_k$ je nejkratší cesta z v_0 do v_k přes vrcholy z množiny A a tedy můžeme předpokládat, že žádný z vrcholů $v_1, \dots, v_k$ není w (podle toho, co jsme si rozmysleli na konci minulého odstavce). Potom se ale délka cesty do v rovnala správné hodnotě už před právě proběhlým krokem.

Vzhledem k tomu, že po posledním kroku množina A obsahuje právě ty vrcholy, do kterých existuje cesta z vrcholu v_0, dokázali jsme, že náš algoritmus funguje správně.

Na závěr ještě poznamenejme, že Dijkstrův algoritmus je možné snadno upravit tak, aby nám kromě určení délky nejkratší cesty i takovou cestu našel: U každého vrcholu si v okamžiku, kdy mu měníme ohodnocení, poznamenáme, ze kterého vrcholu do něj přicházíme. Nejkratší cestu do nějakého vrcholu pak zrekonstruujeme tak, že u posledního vrcholu této cesty zjistíme, který vrchol je předposlední, u předposledního, který je předpředposlední, atd.

Poznámka pro zvídavé: existují i jiné druhy hald, například k-regulární haldy, v nichž má každý prvek k následníků (rozmyslete si, jaká je v takové haldě časová složitost operací a jak nastavit k v závislosti na M a N, aby byl Dijkstrův algoritmus co nejrychlejší), nebo tzv. Fibonacciho halda, která dokáže upravit hodnotu prvku v konstantním čase. S tou pak umíme hledat nejkratší cesty v čase $\mathcal{O}(M + N \log N)$.

Vzorovou implementaci Dijkstrova algoritmu lze nalézt na následující straně.

Dnešní menu Vám servírovali

Dan Kráľ, Martin Mareš a Petr Škoda

Implementace Dijkstrova algoritmu

```
var N: word;                                    { počet vrcholů }
    vahy: array[1..MAX, 1..MAX] of integer;
        { váhy hran, -1 = hrana neexistuje }
    delky: array[1..MAX] of integer;
        { délky zatím nalezených cest, -1 = nekonečno }
    def: array[1..MAX] of boolean;              { definitivní? }

procedure Dijkstra(odkud: word);
var i, w, v: word;
begin
  for i:=1 to N do begin
    def[i]:=false;  delky[i]:=-1;
  end;
  def[odkud]:=true;
  delky[odkud]:=0;
  repeat
    w:=0;
    for i:=1 to N do
      if not def[i] and ((w=0) or (delky[i]<delky[w])) then
        w:=i;
    if w<>0 then begin
      def[w]:=true;
      for i:=1 to N do
        if (vahy[w][i]<>-1) and (delky[w]+vahy[w][i]<delky[i])
          then delky[i]:=delky[w]+vahy[w][i]
    end
  until w=0;
end;
```

Kuchařka čtvrté série – intervalové stromy

Představme si, že máme posloupnost celých čísel

$$p_0, p_1, \ldots p_{N-1},$$

se kterou budeme průběžně provádět tyto dvě operace:

1. Změna jednoho čísla v posloupnosti.
2. Zjištění součtu čísel na nějakém intervalu $[a, b]$, tedy $p_a + p_{a+1} + \ldots + p_b$.

Nejdříve se zkusíme zamyslet, jak bychom úlohu řešili, kdybychom měli jen druhou operaci, tj. dotazy na součty na konkrétních intervalech. K řešení využijeme pole *prefixových součtů*.

Pole prefixových součtů je pole délky $N+1$, ve kterém na indexu i leží součet prvků posloupnosti od indexu 0 až do indexu $i-1$. Tedy

$$\mathit{pref}[i] = p[0] + \ldots + p[i-1], \quad \mathit{pref}[0] = 0$$

Není těžké si rozmyslet, že toto pole dokážeme jednoduše spočítat v čase $\mathcal{O}(N)$.

Nyní, když už známe všechny prefixové součty posloupnosti, umíme snadno spočítat součet na libovolném intervalu $[a, b]$:

$$s[a, b] = \mathit{pref}[b+1] - \mathit{pref}[a]$$

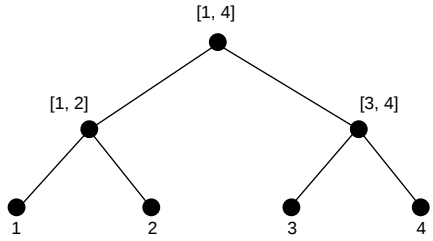
Každý dotaz dokážeme zodpovědět v konstantním čase. Celý algoritmus má tedy složitost $\mathcal{O}(N + D)$, kde N je délka posloupnosti a D je počet dotazů.

Když si do úlohy přidáme i operaci č. 1 (změna čísla v posloupnosti), tak se nám pokazí časová složitost. S prefixovými součty stále dokážeme dotaz č. 2 provádět v konstantním čase, ale při operaci č. 1 se nám může stát, že musíme změnit až všechny prefixové součty, takže složitost této operace je $\mathcal{O}(N)$ a celková složitost pro Z změn a D dotazů je v nejhorším případě $\mathcal{O}(NZ + D)$.

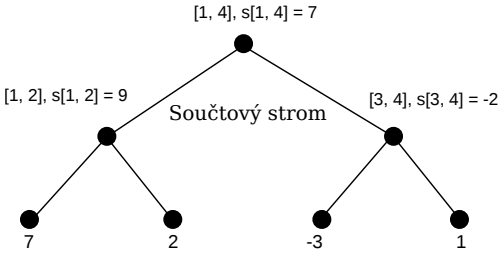
Tato složitost nám samozřejmě stačit nebude a pokusíme se, abychom výsledné intervaly uměli co nejrychleji skládat z předpočítaných hodnot a abychom při změně posloupnosti museli změnit co nejméně hodnot. K tomu se nám bude hodit datová struktura jménem intervalový strom.

Zavedení intervalového stromu

Intervalový strom je dokonale vyvážený binární strom, jehož každý list představuje nějaký interval a všechny ostatní vrcholy reprezentují interval, který vznikne složením intervalů jejich synů. Zároveň intervaly vrcholů jedné hladiny na sebe navazují (vždy směrem zleva doprava). Z toho vyplývá, že složením intervalů z vrcholů jedné hladiny dostaneme interval, který si pamatujeme v kořeni.



Intervalových stromů existuje více druhů. Obvykle je rozlišujeme podle toho, jaké informace si v nich pamatujeme. Například ve stromě pro součty si každý vrchol pamatuje součet na svém intervalu, ve stromě pro maxima si pamatuje maximum na intervalu, apod. Můžeme ale klidně mít strom, který si pamatuje, jestli celý jeho interval obsahuje jen jednu hodnotu a pokud ano, tak jakou.



My se teď zaměříme na intervalový strom pro součty a pomocí něj vyřešíme úvodní úlohu.

Na začátku budeme chtít, aby v listech intervalového stromu byly hodnoty původní posloupnosti, přičemž první a poslední list stromu necháme volné, později uvidíme, proč. Zároveň ale chceme, aby tento strom byl dokonale vyvážený.

Posloupnost tedy prodloužíme tak, aby její velikost byla mocnina dvojky minus dva (na její konec přidáme nějaké prvky). Všimněte si, že tím jsme strom nezvětšili více než dvakrát a že nám nezáleží na tom, jaké prvky jsme do stromu přidali, protože s nimi nikdy nebudeme pracovat. Nyní k jednotlivým operacím.

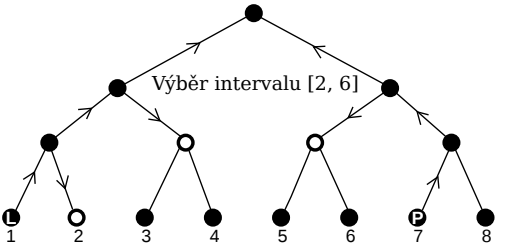
Změnu čísla v posloupnosti uděláme jednoduše. Zjistíme, o kolik se hodnota prvku posloupnosti změní, najdeme odpovídající list a k tomuto listu a ke všem jeho předkům přičteme daný rozdíl. Tím jsme upravili všechny intervaly, do kterých tento prvek patří.

Nyní se podívejme, jak ze stromu zjistíme součet na nějakém intervalu $[a, b]$. Jinými slovy: potřebujeme ze stromu vybrat takové vrcholy, aby sjednocení jejich intervalů byl náš dotazovaný interval, a zároveň chceme, aby těchto vrcholů bylo co nejméně.

Součet intervalu $[a, b]$ zjistíme tak, že si ve stromě najdeme listy reprezentující pozice $a - 1$ a $b + 1$ posloupnosti a jejich nejbližšího společného předka p.

Nyní budeme postupovat z listu od $a-1$ až do p a vždy když do nějakého vrcholu přijdeme z levého syna, tak do výsledku přidáme interval pravého syna. Stejně tak postupujeme od $b+1$ k p a pokud do vrcholu přijdeme z pravého syna, tak přidáme jeho levého syna.

Všimněte si, že při takovémto průchodu složíme celý interval. Vše je vidět na nasledujícím obrázku:



Způsobů, jak pracovat z intervalovým stromem a zjišťování informací z něj, je více. Toto byl jeden z nich.

Změna prvku posloupnosti má časovou složitost $\mathcal{O}(\log N)$, protože jsme na každé hladině změnili pouze jeden interval a strom má $\mathcal{O}(\log N)$ hladin. Zjištění součtu na intervalu má také složitost $\mathcal{O}(\log N)$, jelikož jsme do výsledku přidali maximálně $2 \log N$ intervalů: nejvýše $\log N$ při cestě z listu $a-1$ a $\log N$ při cestě z $b+1$.

Implementace intervalového stromu

Při implementaci intervalového stromu využijeme jeho dokonalé vyváženosti a budeme jej implementovat v poli (stejně jako se do pole ukladá halda). Kořen stromu bude v poli na indexu 1, vrcholy z druhé hladiny budou mít postupně indexy 2 a 3, ..., až listy budou mít indexy N, ..., $2N-1$. V této reprezentaci platí pro vrchol s indexem i následující pravidla:

1. $2i$ a $2i+1$ jsou jeho synové.
2. $\lfloor i/2 \rfloor$ je jeho předek (pro $i > 1$).
3. Pokud je i sudé, tak je vrchol levým synem, jinak pravým.
4. Pro sudé i je $i+1$ pravý bratr, pro liché i je $i-1$ levý bratr.

Nyní víme vše potřebné, tak se podívejme na implementaci v jazyce C:

```
int N = 100;    // velikost posloupnosti
int posl[100];  // posloupnost
int *strom;     // intervalový strom

// Deklarace funkcí
void inic(int N);
void pricti(int index, int hodnota);
int soucet(int A, int B);
```

```
// Inicializace intervalového stromu
// Pozor: prvky posloupnosti indexujeme 1, ..., N
void inic(int N) {
  // Najdeme nejbližší vyšší mocninu dvojky
  int listy = 1;
  while (listy<N+2) listy = listy*2;
  // Pro strom potřebujeme 2*(počet listů) vrcholů
  // (nepoužíváme strom[0])
  strom = (int*)malloc(sizeof(int)*2*listy);
  N = listy;
  for (int i=0; i<2*listy; i++) strom[i] = 0;
  // Na příslušná místa přičteme hodnoty posloupnosti
  for (int i=0; i<N; i++)
    pricti(i, posl[i]);
}

// Přičtení hodnoty na dané místo posloupnosti
void pricti(int index, int hodnota) {
  int k = N + index;
  while(k>0) {
    strom[k] = strom[k] + hodnota;
    k = k/2;
  }
}

// Zjištění součtu na intervalu
int soucet(int A, int B) {
  int souc = 0;
  int a = N + A - 1;
  int b = N + B + 1;
  while (a!=b) {
    // Pokud je a levý syn, tak přičti pravého bratra
    if (a%2==0) souc = souc + strom[a+1];
    // Pokud je b pravý syn, tak přičti levého bratra
    if (b%2==1) souc = souc + strom[b-1];
    // Přesun na otce
    a = a/2; b = b/2;
  }

  // Navíc jsme přičetli syny společného předka.
  souc = souc - strom[2*a] - strom[2*a+1];
  return souc;
}
```

V této implementaci jsme strom upravovali zdola směrem nahoru. Existuje ještě rekurzivní implementace, v níž se strom upravuje od kořene směrem dolů, ale tu si zde ukazovat nebudeme.

Cvičení

- Naprogramujte rekurzivní implementaci operací (strom se prochází shora dolů).
- Jak by vypadala implementace intervalového stromu pro maxima?

Použití intervalového stromu

Intervalový strom je silný nástroj, kterým se dá vyřešit spousta úloh. Ale než ho začnete používat, tak si vždy rozmyslete, zda úloha nelze řešit elegantněji bez intervalového stromu. Ne všechny druhy intervalových stromů se dobře implementují.

Intervalový strom obvykle použijeme, pokud potřebujme průběžně zjišťovat informace o intervalech a zároveň je i měnit. Pokud používáme jen jednu z těchto operací (a tu druhou jen zřídka), existuje často lepší řešení než intervalový strom – viz úvodní příklad.

Fenwickův strom

Fenwickův strom, někdy také nazývaný jako *finský strom,* je v podstatě jen strom reprezentovaný v poli. Jeho používání je podobné jako používání intervalového stromu pro součty. Rozdíl je jen v implementaci daných funkcí. My si Fenwickův strom opět ukážeme na úvodním příkladu. Zase tedy budeme potřebovat funkci pro změnu hodnoty v posloupnosti a funkci pro zjištění součtu na intervalu. (Ve skutečnosti zjistíme dva prefixové součty a z nich pak spočítáme výsledný interval.)

Fenwickův strom je poněkud magická datová struktura. Abychom si tuto magii mohli užít, zvolíme trochu netradiční způsob vysvětlování a nejdříve si ukážeme, jak se Fenwickův strom implementuje a teprve pak si vysvětlíme, jak to všechno funguje.

Fenwickův strom bude pole velikosti $N+1$, kde index 0 nebudeme používat. Používat budeme pouze prvky $1, \ldots, N$, které všechny na začátku nastavíme na 0. Pokud v posloupnosti změníme hodnotu, stejně jako u intervalového stromu, ve Fenwickově stromě na některá místa přičteme rozdíl oproti předchozí hodnotě.

```
void pricti(unsigned int index, int rozdil) {
  while (index<=N) {
    strom[index] += rozdil;
    index = index + (index & -index);
    // "&" značí bitový AND
  }
}
```

A zde je funkce pro zjištění prefixového součtu:

```
int pref_soucet(unsigned int index) {
  int soucet = 0;
  while (index>0) {
    soucet = soucet + strom[index];
    index = index & (index-1);
  }
  return soucet;
}
```


Toť celá implementace. No, nevypadá na první pohled magicky? Pokud chcete vědět, jak tohle celé funguje, tak čtěte dál.

Ve Fenwickově stromě je na indexu 1 uložen první prvek, na indexu 2 součet prvního a druhého, na indexu 3 třetí prvek na indexu 4 součet prvních čtyř, ... na indexu N je uložen součet posledních 2^K hodnot, kde K je pozice prvního jedničkového bitu v binárním zápise čísla N. Ve stromě máme tedy uloženou takovou pravidelnou strukturu intervalů.

Nyní se podíváme, co dělají naše magické funkce na posouvání ve stromě a pak najednou bude všechno jasné. Ve výrazu `index & (index-1)` z funkce `pref_soucet()` se neděje nic jiného než, že se vynuluje nejpravější jedničkový bit v indexu. Tím se dostaneme na první interval, který jsme ještě nepřičetli. V momentě, kdy se dostaneme na index 0, tak už máme dotazovaný interval kompletní a výpočet můžeme ukončit.

Výraz `index + (index & -index)` dělá to, že se v pomyslném stromě intervalů posune o úroveň výš.[36] Pokud jsme tedy v intervalu o velikosti 2, tak se dostaneme do intervalu velikosti 4, který daný interval obsahuje (tento interval je jednoznačný). Samotný výpočet dělá to, že v čísle `index` vezme nejpravější jedničku a znova ji přičte.

Fenwickův strom se používá hlavně kvůli jednoduchosti jeho naprogramování a také kvůli efektivitě samotného výpočtu a nevelké náročnosti na paměť. Při jeho implementaci doporučujeme dávat si pozor na správnost bitových funkcí.

Cvičení

- Rozmyslete si, že oba magické výpočty opravdu dělají to, co mají, a také, proč vše vlastně funguje.

Karel Tesař

[36] Magická operace `index & -index` funguje jen v případě, že se jako reprezentace záporného čísla používá tzv. dvojkový doplněk: `-k == ~k + 1`, neboli všechny bity čísla se znegují a pak se přičte jednička.

Kuchařka páté série – hledání v textu

Řetězec je v podstatě jakákoli posloupnost symbolů zapsaná za sebou a s nimi budeme v této kapitole pracovat. Každého napadne „vyhledávání v textu“ nebo „hledání jmen v telefonním seznamu“, ale řetězce najdeme i na nižších úrovních informatiky. Například celé číslo zakódované v binární soustavě, které dostaneme na vstupu programu, je také jen řetězec nul a jedniček.

Jiný příklad použití řetězců (a jejich algoritmů) najdeme v biologii. DNA není o mnoho více, než chytré uložení posloupnosti čtyř znaků/nukleových bazí – a chceme-li hledat vzory anebo konkrétní podposloupnosti, bude se nám hodit znalost základních algoritmů pro práci s řetězci.

Nemáme bohužel šanci vysvětlit *všechny* algoritmy s řetězci, protože je příliš mnoho možných věcí, co s řetězci dělat. Převáděním řetězců na čísla (hešováním) jsme se věnovali v jiné kuchařce, v této se budeme soustředit na algoritmy, které se objevují spíše v práci s textem.

Kromě úvodu popíšeme dva *stavební kameny* textových algoritmů, což bude jedna datová struktura pro slovníky (trie) a jedno vyhledání v textu s předzpracováním hledaného slova (a jeho rozšíření pro více slov). S jejich znalostí se pak mnohem snáze vymýšlí řešení složitějších, reálnějších problémů.

Jak řetězce chápat

Když programátor dělá první krůčky, často moc netuší, co s těmi řetězci vlastně může a nesmí dělat. V programovacím jazyce to je jasné – něco mu jazyk dovolí a na něco nejsou prostředky. Ale jak to je na úrovni ryze teoretické?

Jak jsme si řekli na začátku, řetězec bude posloupnost nějakých symbolů, kterým říkáme *znaky*. Tyto znaky jsou z nějaké množiny, které říkáme *abeceda*. Abeceda může být jen {`0`, `1`} pro čísla v binárním zápisu, klasické {`A-Z`, `a-z`} pro anglickou abecedu anebo plný rozsah univerzální znakové sady Unicode, která má až 2^{31} znaků. Nezapomínejme, že nejenom písmena a číslice, ale i mezery a interpunkce jsou znaky!

Vidíme, že zanedbat velikost abecedy při odhadu složitosti by bylo příliš troufalé, a tak budeme velikost abecedy označovat $|\Sigma|$. Abeceda sama se v textech o řetězcích často značí řeckým Σ.

O znacích samotných předpokládáme, že jsou dostatečně malé, abychom s nimi mohli pracovat v konstantním čase, podobně jako s celými čísly v ostatních kapitolách.

Nyní hlavní otázka – máme chápat řetězec jako pole znaků, nebo jako spojový seznam? Šalamounská odpověď: můžeme s ním pracovat tak i tak. Když budeme potřebovat převést řetězec na spojový seznam (protože se nám hodí rychlé přepojování řetězců), tak si jej převedeme. Tento převod nás samozřejmě bude stát čas lineárně závislý na *délce* řetězce. Budeme ji dále značit L; časová složitost převodu bude $\mathcal{O}(L)$.

Standardně se ale počítá s tím, že řetězec je uložen v poli někde v paměti (již od začátku algoritmu), takže ke každému znaku můžeme přistupovat v konstantním čase.

Jelikož jsme řetězce definovali jako posloupnosti, nesmíme zapomínat ani na *prázdný řetězec* ε. A když už máme řetězec, určitě máme i *podřetězec* – souvislou podposloupnost znaků jiného řetězce. Například `BAR`, `RET`, ε i `KABARET` jsou podřetězce slova (řetězce) `KABARET`; `KAT` však podřetězcem není.

Často nás budou zajímat dva zvláštní druhy podřetězců. Pokud ze slova odstraníme nějaký souvislý úsek na konci, vznikne podřetězec, kterému říkáme *prefix* (česky předpona), a pokud odstraníme nějaký souvislý úsek ze začátku, dostaneme *suffix* neboli příponu. `RET` je suffix slova `KABARET`, `KABA` je zase jeho prefixem.

Terminologie dovoluje zepředu i zezadu odstranit prázdný řetězec – to znamená, že slovo je samo sobě prefixem i suffixem. Pokud chceme mluvit o prefixech, suffixech nebo obecně podřetězcích, kde jsme museli alespoň jeden znak odtrhnout, označíme takové podřetězce jako *vlastní*.

Pro některá použití řetězců je důležité, abychom je mohli porovnávat – když máme řetězce R a S, chceme rozhodnout, který je menší, a který je větší. Jaké přesně toto uspořádání bude, závisí na naší aplikaci, ale mnohdy se používá tzv. *lexikografické uspořádání*.

Pro lexikografické uspořádání potřebujeme nejprve zadané lineární uspořádání na znacích. Tím se myslí takové, kde jsou všechny prvky navzájem porovnatelné a v podstatě to znamená, že jsou uspořádány v jedné řadě za sebou (kromě binárního $\texttt{0} < \texttt{1}$ se často používá „telefonní“ $\texttt{A} = \texttt{a} < \texttt{B} = \texttt{b} < \ldots < \texttt{Z} = \texttt{z}$, které je ovšem lineární až na velikost znaků).

Když máme zadané uspořádání na znacích, na všechny řetězce je rozšíříme následovně: Nejkratší je prázdný řetězec. Ostatní řetězce třídíme podle jejich prvního znaku. Jestliže se první znak shoduje, tak podle druhého znaku, atd. Pokud přitom znaky jednoho z řetězců dojdou dřív, prohlásíme tento řetězec za menší.

Platí tedy třeba $\varepsilon < \texttt{A} < \texttt{AUTO} < \texttt{AUTOBUS} < \texttt{AUTOGRAM} < \texttt{AUTOR} < \texttt{BAMBITKA} < \texttt{BARNABAS} < \texttt{Z}$.

Adresář pomocí trie

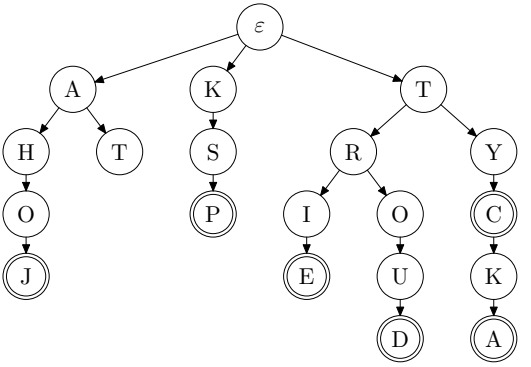
Typický „textový“ problém je udržování množiny řetězců – můžete si představit třeba slovník. Slova ve slovníku si chceme šikovně předzpracovat, abychom pak mohli efektivně odpovídat na otázky typu: „Je slovo S obsaženo ve slovníku?“ Můžeme také po předzpracování chtít přidávat nové položky, nebo i odebírat staré.

Pokud bychom nemuseli odebírat slova, můžeme použít hešování, které je rychlé a účinné. Více o něm najdete v hešovací kuchařce.[37] Má však tu nevýhodu, že při velkém zaplnění se začne chovat pomaleji a mírně nepředvídatelně.

Ukážeme si jiné řešení, které je také asymptoticky rychlé a není ani příliš náročné na paměť. Využívá stromové struktury a říká se mu *trie* (vyslovujeme česky „tryje“ a anglicky jako část slova „retrieval“, z něhož slovo trie vzniklo). V češtině se občas používá také označení „písmenkový strom“.

Trie bude zakořeněný strom. V prvním patře se bude větvit podle prvního písmene slova, ve druhém podle dalšího, a tak dále.

Obrázek vydá za tisíc definic, pojďme se podívat, jak vypadá trie pro slova `AHOJ`, `AT`, `KSP`, `TRIE`, `TROUD`, `TYC`, `TYCKA`. Pro přehlednost písmena místo na hrany kreslíme do následujících vrcholů:



Všimněte si, že vrcholy v hloubce h (tedy v h-tém patře trie) odpovídají prefixům délky h zadaných slov. Například prefixy délky 2 jsou `AH`, `AT`, `KS`, `TR` a `TY`. Hrana mezi prefixy vede právě tehdy, lze-li jeden z druhého získat připsáním písmene na konec.

Jak bychom takovou trii postavili algoritmem? Přesně, jak jsme ji definovali: každé slovo ze slovníku budeme procházet znak po znaku a bude-li nějaká hrana chybět, tak ji vytvoříme a pokračujeme dále podle slova.

Z takto popsané trie bohužel nepoznáme, kde končí slovo ze slovníku a kde končí jen jeho prefix. Standardní způsoby, jak to vyřešit, jsou dva: buď si do každého vrcholu přidáme informaci o tom, je-li koncem celého slova nebo ne (jak je to naznačeno dvojitými kroužky v obrázku), anebo si rozšíříme abecedu o speciální znak, který se v ní předtím nevyskytoval – třeba `$` – a pak všem slovům přilepíme tento `$` na konec.

Budeme-li se později ptát, bylo-li slovo ve slovníku, po průchodu trií zkontrolujeme ještě, jestli z konečného vrcholu vede hrana odpovídající znaku `$`.

[37] `http://ksp.mff.cuni.cz/viz/kucharky/hesovani`

Ještě jsme si nerozmysleli, jak budeme v jednotlivých vrcholech trie reprezentovat hrany do delších prefixů. Abychom mohli vyhledávat skutečně lineárně, potřebovali bychom umět v konstantním čase odpovědět na otázku „má vrchol P potomka přes hranu se znakem c?“.

Abychom zajistili konstantní čas odpovědi, museli bychom mít v každém vrcholu pole indexované znaky abecedy. To ovšem znamená, že takové pole budeme muset vytvořit, a tedy alokovat $|\Sigma|$ políček v každém znaku.

To zvýší paměťovou náročnost trie (a časovou náročnost její stavby) na $\mathcal{O}(D \cdot |\Sigma|)$, kde D značí velikost vstupu, čili součet délek všech slov ve slovníku. To je naprosto přijatelné pro malé abecedy, ale už pro {`A-Z`, `a-z`} je tento faktor roven 52 a pro Unicode je taková alokace nemyslitelná.

Jak z toho ven? Můžeme oželet konstantní rychlost dotazu a použít namísto pole třeba binární vyhledávací strom nebo hešovací tabulku všech znaků, kterými aktuální prefix může pokračovat. Nebo můžeme každý znak velké abecedy zapsat pomocí několika znaků menší abecedy. Tou menší abecedou může být třeba {`0`, `1`}. Tehdy nahradíme každý znak původní abecedy $\lceil \log_2 |\Sigma| \rceil$ novými (jeho zápisem ve dvojkové soustavě). Tím se časová slozitost konstrukce zlepší na $\mathcal{O}(D \cdot \log |\Sigma|)$ a časová složitost dotazu na slovo délky L zhorší na $\mathcal{O}(L \cdot \log |\Sigma|)$.

A jsme hotovi! S trií můžeme v lineárním čase odpovídat na dotazy „Vyskytuje se dané slovo ve slovníku?“, přidávat a odebírat další položky za běhu a nejen to – víc o tom ve cvičeních.

Poznámky

- Chcete-li algoritmus konstrukce trie vidět napsaný v Pascalu, podívejte se do knihy *Algoritmy a programovací techniky*.
- Triím se také říká *prefixové stromy*, což popisuje, že každý vrchol odpovídá prefixu některého slova ve slovníku.
- Kdybychom chtěli, mohli bychom pomocí trie vyhledávat v textu v lineárním čase. Můžeme přeci postavit slovník ze všech slov v daném textu, a pak procházet příslušnou trii. Má to ale pár háčků: jednak je často hledaný řetězec krátký, ale text se nevejde do paměti. Druhak, pokud bychom použili jako oddělovač mezery, mohli bychom hledat jen jednotlivá slova, a nikoli jejich konce nebo delší kusy věty.
- Asi se po poslední poznámce ptáte – existuje nějaká modifikace trie, která umí hledat libovolnou část textu? Ano, jmenuje se *suffixový strom* a jdou s ní dělat spousty krásných kousků. Říká se, že každou řetězcovou úlohu lze řešit v lineárním čase pomocí suffixových stromů. Víc se o nich dočtete třeba v knížce *Krajinou grafových algoritmů*.[38]

[38] `http://mj.ucw.cz/vyuka/ga/`

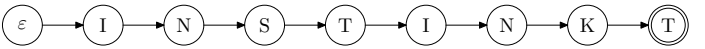
Cvičení

- Řekněme, že chceme slovník na vstupu setřídit v lexikografickém pořadí (definovaném v sekci „Jak řetězce chápat“). Problémem pro klasické třídící algoritmy je to, že porovnání dvou řetězců není bohužel konstantně rychlé. Vymyslete způsob, jak setřídit takový slovník rychle pomocí trie.
- *Komprese trie.* Co kdybychom chtěli odstranit přebytečné vrcholy trie, tedy ty, v nichž se slova nevětví? Rozmyslete si, jestli by něčemu vadilo místo takovýchto cest mít jen jednotlivé hrany. Zesložití se konstrukce nebo vyhledávání? Mimochodem, je celkem jasné, že takováto *komprimovaná trie* přinese jen konstantní zrychlení dotazů i prostoru, a tak na soutěžích apod. stačí použít základní variantu.

Vyhledávání v textu

Začátek situace je asi zřejmý – máme na vstupu zadán dlouhý text a krátké slovo. Slovo si můžeme nějak předzpracovat, načež projdeme co nejrychleji text a nahlásíme jeden nebo všechny výskyty slova. Zajímají nás při tom i výskyty, které se navzájem překrývají: v textu `NANANA` se slovo `NANA` vyskytuje dvakrát. Často se hovoří o „hledání jehly v kupce sena“, pročež se textu přezdívá *seno* a hledanému slovu *jehla*. Délku jehly označíme J a délku textu S.

Představme si nejdříve hledané slovo jako spojový seznam, třeba slovo `INSTINKT`:



Mohli bychom text začít procházet znak po znaku a kontrolovat, zda se text shoduje s naším slovem/spojovým seznamem. Pokud by si znaky odpovídaly, skočíme na další znak z textu a i na další znak v seznamu. Co když se ale neshodují? Pak nemůžeme jen skočit na další znak textu – co kdybychom v textu narazili na slovo `INSTINSTINKT`?

Musíme se tedy vrátit nejen na začátek spojového seznamu, ale i zpátky v textu na druhý znak, který jsme označili jako odpovídající, a zkoušet porovnávat s jehlou znovu od začátku. To už naznačuje, že takto získaný algoritmus nebude lineární, protože se musí vracet zpět v textu o délku jehly.

Sice je předchozí popis skutečně v nejhorším případě složitý $\mathcal{O}(S \cdot J)$, avšak stačí malá úprava a složitost přejde na lineární $\mathcal{O}(S + J)$. Ve skutečnosti algoritmus nezpomalovalo vracení se – za špatnou složitost mohl fakt, že jsme se vraceli *příliš zpátky*.

Třeba v našem příkladu s textem `INSTINSTINKT` se nemusíme vracet ve spojovém seznamu na začátek, jakmile načteme `INSTINS`. Mohli jsme se vrátit jen na druhý znak, tedy do prvního `N`, a pak kontrolovat, jaký znak pokračuje dál. Když následuje `S` jako v našem případě, můžeme pokračovat dále v čtení a nevracíme se v textu. Kdyby text byl jiný, třeba `INSTINB`, vrátili bychom se po

načtení `B` na začátek spojového seznamu a v textu bychom pokračovali dále bez zastavení.

Pro každý znak ve spojovém seznamu si tedy určíme políčko spojového seznamu, na které skočíme, pokud se následující znak v textu liší od toho očekávaného. Pořadové číslo tohoto políčka nám poradí tzv. *zpětná funkce* F, což bude funkce definovaná pomocí pole, kde $F[i]$ bude pořadové číslo políčka, na které se má skočit z políčka číslo i. Porovnávat pak budeme s následujícím znakem. Pokud $F[i] = 0$, znamená to, že máme začít porovnávat úplně od prvního znaku jehly.

Pokud máte rádi grafovou terminologii, můžete se na náš spojový seznam dívat jako na graf a hovořit o *zpětných hranách*.

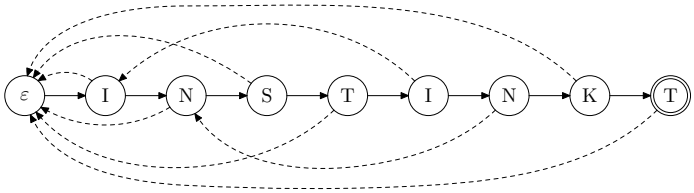
Zatím jsme ale přesně nepopsali, na které políčko přesně bude zpětná funkce ukazovat. Nechť chceme určit zpětné políčko pro druhé `N` ve slově `INSTINKT`. Pracujeme teď s prefixem `INSTIN`. Selsky řečeno, chceme najít „konec slova `INSTIN` takový, že je stejný, jako začátek slova `INSTIN`".

Abychom náš požadavek upřesnili, zamyslíme se nad zpětným políčkem pro jiné slovo. Co kdyby jehlou bylo slovo `ABABABC` a my určovali zpětné políčko pro `ABABAB`? Kdybychom ukázali na první písmenko `B`, nebylo by to správně, protože pak bychom pro text `ABABABABC` nezahlásili výskyt jehly, což je jasná chyba. Musíme se vrátit už na `ABAB`!

Zajímá nás tedy ne libovolný suffix, který je stejný jako začátek, ale nejdelší takový konec/suffix. A ještě navíc ne jen ten nejdelší, ale nejdelší „netriviální" – slovo `INSTIN` je samo sobě prefixem a suffixem, ale zpětná funkce pro `N` by se neměla cyklit, měla by vést zpátky.

Řekněme to tedy znova, zcela formálně: pokud bychom právě určovali hodnotu zpětné funkce pro znak číslo i, kterému odpovídá prefix P, pak její hodnota bude *délka nejdelšího vlastního suffixu* slova P, pro který ještě platí, že *je zároveň prefixem* P.

Pro slovo `INSTINKT` vypadá spojový seznam se zpětnou funkcí (zakreslenou pomocí ukazatelů) takto:



Nyní vyvstávají dvě otázky: Jakou to má celé časovou složitost? A jak spočítat zpětnou funkci?

Poperme se nejdříve s tou první. Pro každý znak vstupního textu mohou nastat dva případy: Buď znak rozšiřuje aktuální prefix, nebo musíme použít

zpětnou funkci. První případ má jasně konstantní složitost, druhý je horší, neboť zpětná funkce může být pro jeden znak volána až J-krát.

Při každém volání však klesne pořadové číslo aktuálního stavu (políčka) alespoň o jedna, zatímco kdykoliv stav prodlužujeme, roste jen o jeden znak. Proto všech zkrácení dohromady může být nejvýše tolik, kolik bylo všech prodloužení, čili kolik jsme přečetli znaků textu. Celkem je tedy počet kroků automatu lineární v délce textu, $\mathcal{O}(S)$.

Konstrukci zpětné funkce provedeme malým trikem. Všimněme si, že $F[i]$ je přesně číslo stavu, do nějž se dostaneme při spuštění našeho vyhledávacího algoritmu na řetězec, který tvoří prefix délky i z jehly bez prvního znaku.

Proč to tak je? Zpětná funkce říká, jaký je nejdelší vlastní suffix daného stavu, který je také stavem, zatímco políčko, ve kterém po i krocích skončíme, označuje nejdelší suffix textu, který je stavem. Tyto dvě věci se přeci liší jen v tom, že ta druhá připouští i nevlastní suffixy, a právě tomu zabráníme odstraněním prvního znaku.

Takže F získáme tak, že spustíme vyhledávání na část samotné jehly. Jenže k vyhledávání zase potřebujeme funkci F. Jak z toho ven? Budeme zpětnou funkci vytvářet postupně od nejkratších prefixů. Zřejmě $F[1] = 0$. Pokud již máme $F[i]$, pak výpočet $F[i+1]$ odpovídá spuštění automatu na slovo délky i a při tom budeme zpětnou funkci potřebovat jen pro stavy délky i nebo menší, pro které ji již máme hotovou.

Navíc nemusíme pro jednotlivé prefixy spouštět výpočet vždy znovu od začátku – $(i+1)$-ní prefix je přeci prodloužením i-tého prefixu o jeden znak. Stačí tedy spustit algoritmus na jehlu bez prvního znaku a sledovat, jakými stavy bude procházet – to budou přesně hodnoty zpětné funkce.

Vytvoření zpětné funkce se nám tak nakonec zredukovalo na jediné vyhledávání v textu o délce $J-1$, a proto poběží v čase $\mathcal{O}(J)$. Časová složitost celého algoritmu tedy bude $\mathcal{O}(S+J)$. Dodáme už jen, že tento algoritmus poprvé popsali pánové Knuth, Morris a Pratt a na jejich počest se mu říká KMP. Naprogramovaný bude vypadat následovně (čtení vstupu jsme si odpustili):

```
var
  Slovo: array[1..J] of char;        { jehla }
  Text: array[1..S] of char;          { seno }
  F: array[1..J] of integer;    { zpětná fce }
  I, J: integer;          { pomocné proměnné }

  function Krok(I: integer; C: char): integer;
  begin
    if (I < J) and (Slovo[I+1] = C) then
      Krok := I + 1
    else if I > 0 then
      Krok := Krok(F[I], C)
```

```
      else
        Krok := 0;
    end;

  begin { konstrukce zpětné funkce }
    F[1] := 0;
    for I := 2 to J do
      F[I] := Krok(F[I-1], Slovo[I]);
    { procházení textu }
    J := 0;
    for I := 1 to S do begin
      J := Krok(J, Text[I]);
      if J = J then
        writeln(I);
    end;
  end.
```


Poznámky

- Pro anglický nebo český text je použití takto sofistikovaného algoritmu skoro škoda, protože v obou jazycích se stává jen málokdy, že bychom měli několik slov spojených dohromady. Prakticky bude stačit i na začátku zmíněný naivní algoritmus. Na soutěžích a olympiádách ale pište raději algoritmus KMP.
- Hešování lze použít i na vyhledávání řetězce v textu. Obzvláště vhodné jsou na to *rolling hash functions* („okénkové hešovací funkce“), které umí v konstantním čase přepočítat heš, ubereme-li nějaký znak na začátku a přidáme-li jiný na konci – jako kdybychom se dívali na text skrz posouvající se okénko.

Cvičení

- Rozmyslete si, že když vyhledáváme více slov, ne jen jedno, a algoritmus musí vypsat všechny výskyty na výstup, můžeme se dobrat vyšší než lineární složitosti v závislosti na vstupu. Na čem potom taková časová složitost také záleží?
- Vymyslete nějakou vhodnou okénkovou hešovací funkci pro vyhledávání jedné jehly.

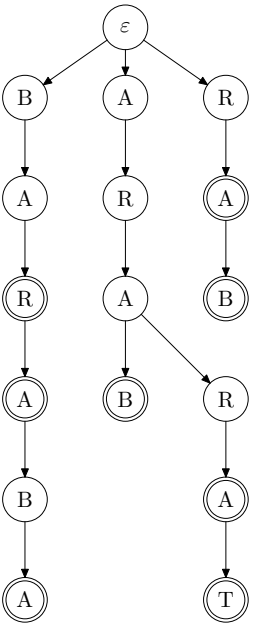
Vyhledávání jehelníčku

Co kdybychom neměli jen jednu jehlu/hledané slovo, ale celý jehelníček, čili seznam hledaných slov? I to lze řešit podobnou metodou, jakou jsme hledali jedno slovo. Tento algoritmus se nazývá po tvůrcích *algoritmus Aho-Corasicková* a spočívá v tom, že jednoduchý spojový seznam nahradíme trií a do trie opět přidáme zpětné hrany.

Budeme postupovat podobně jako u KMP. Nejprve naskládáme jehelníček do trie. Pro příklady v této kuchařce použijeme jehelníček `ARAB`, `ARARA`, `ARARAT`, `BAR`, `BARA`, `BARABA`, `RA` a `RAB`.

Dalším krokem v KMP bylo sestrojení zpětných hran. Nejprve jsme sestrojili zpětnou hranu pro první znak slova, pak pro druhý atd. Ve trii to bude o něco složitější.

Na první pohled se může zdát, že bychom mohli automat sestrojit tak, že bychom vyrobili KMP pro první slovo, pak KMP pro druhé slovo s využitím struktury prvního atd., ale to má háček.



Zpětné hrany totiž nemusí vést do předka. Například pro slovo `BARAB` povede zpětná hrana do slova `ARAB`, z toho do slova `RAB` a z toho do `B`. Kdybychom ale zkonstruovali automat výše popsaným způsobem (a začali slovem `BARAB`), nebude existovat v trii ani `ARAB`, ani `RAB`, takže bychom vedli zpětnou hranu chybně do `B`.

Můžeme se ale opřít o stejný trik, jako při konstrukci KMP. Budeme opět vyhledávat nejdelší vlastní suffix. Kam dojde výpočet po jeho vyhledání, tam povede zpětná hrana.

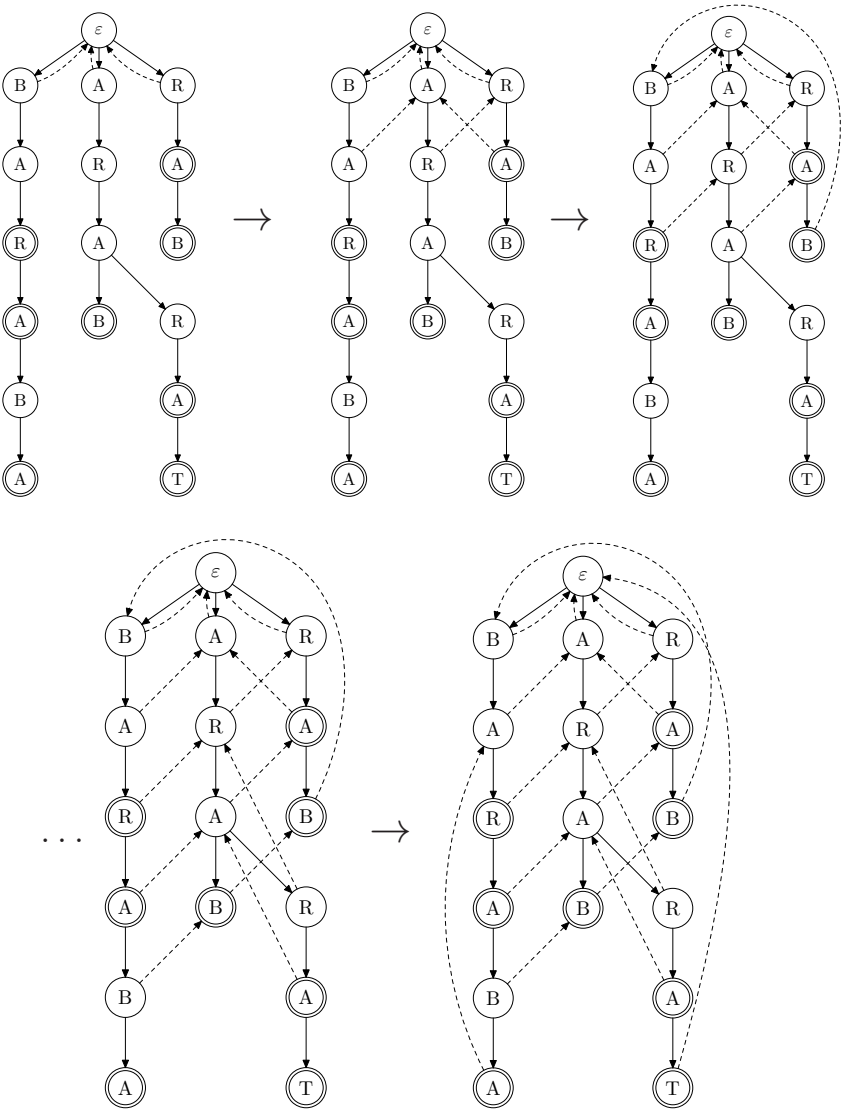
Zkusíme tedy nejprve sestrojit celou trii a pak postupně vyhledat nejdelší vlastní suffix pro každé slovo. Ouha, to také nefunguje. Když začneme slovem `BARABA` a budeme vyhledávat `ARABA`, nalezneme v trii úspěšně prefix `ARAB`, ale `ARABA` již v trii není. Měli bychom přejít ze slova `ARAB` po zpětné hraně, ale tu ještě nemáme zkonstruovanou.

Rozdělíme si trii na *vrstvy* – první znaky slov budou první vrstva, druhé znaky budou tvořit druhou vrstvu atd., až i-té znaky slov budou tvořit i-tou vrstvu.

Zpětná hrana jistě povede do kratšího slova. Z i-té vrstvy tedy povede do vrstvy s nižším pořadovým číslem. Pokud tedy budeme zpětné hrany konstruovat po vrstvách, dojdeme kýženého výsledku.

Ještě zbývá otázka, jak konstruovat zpětné hrany efektivně, když je musíme vyrábět po vrstvách. Mohli bychom prostě vzít slovo, pro které hledáme zpětnou hranu, utrhnout mu první znak a vyhledat. Jenže to budeme dělat spoustu práce zbytečně.

Například pro slovo `BARABA` bychom mohli vyhledávat `ARABA` v již zkonstruované části automatu, ale proč to dělat celé, když jsme při konstrukci předchozí vrstvy vyhledávali `ARAB` při konstrukci zpětné hrany pro `BARAB`?

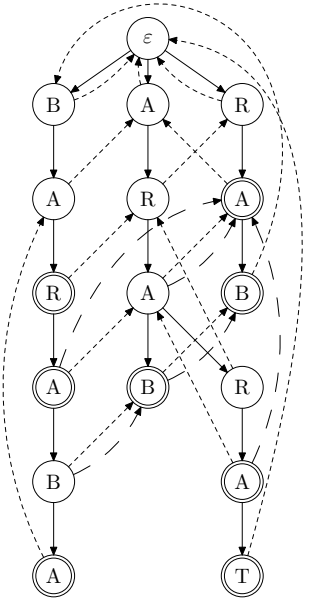


Při konstrukci další zpětné hrany tedy najdeme akorát, kde jsme minule skončili, a odtamtud pokračujeme dál. Jak to najdeme? Z otce našeho vrcholu tam přece vede zpětná hrana.

Postup můžeme shrnout do bodů:

1. c = poslední znak slova (znak stavu P, pro který hledáme zpětnou hranu);
2. přesuneme se do otce;
3. přesuneme se po zpětné hraně;
4. dokud neexistuje syn se znakem c nebo nejsme v kořeni, přesouváme se po zpětných hranách;
5. pokud existuje syn se znakem c, natáhneme do něj zpětnou hranu z P, jinak ji natáhneme do kořene.

Automat je zkonstruován. Časová složitost konstrukce sestává z konstrukce trie v $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$ (pokud použijeme binární strom ve vrcholech) a z předpočítání zpětných hran. Při předpočítávání uděláme nějaký konstantní počet operací pro každý vrchol (celkem tedy $\mathcal{O}(J)$) a také paralelně vyhledáváme všechny jehly z jehelníčku, jejichž vyhledání nás stojí $\mathcal{O}(J)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$.



Tedy konstrukce trvá celkem $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$, paměťová náročnost je stejná jako u trie – $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J)$, přidali jsme jen $\mathcal{O}(J)$ zpětných hran.

Projdeme tedy automatem text `BARABARARAT`. Ohlásí postupně nález slov `BAR`, `BARA`, `BARABA`, `BAR`, `BARA`, `ARARA` a `ARARAT`.

Nenalezl však všechno. Chybí mu např. `ARAB`, který začíná druhým znakem a končí pátým. Dále chybí několik výskytů `RA` a jeden `RAB`.

Když byl na pátém znaku, byl ve stavu `BARAB`, jehož suffixem je `ARAB`. Obecně na suffixy zapomínáme – narozdíl od KMP, kde suffix aktuálního stavu nikdy nebyl jehla, tady jehlou být může.

V každém stavu bychom tedy měli projít veškeré suffixy a zkontrolovat je, jestli náhodou nejsou jehlami. Jak najdeme všechny suffixy? Projdeme postupně po zpětných hranách až do kořene. Má to jen jeden problém – je to pomalé.

Představme si například slovník obsahující `A` a `AAAA...A` (délky $J-1$). Budeme-li jím vyhledávat v textu `AAAA...A` délky $S > J$, projdeme prakticky pro každý znak až $J-1$ zpětných hran, čímž složitost naroste až na nepoužitelných $\mathcal{O}(S \cdot J)$.

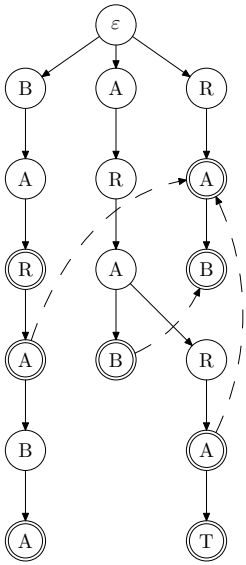
Všimněme si však, že většinou zpětných hran jsme prošli úplně zbytečně. Předpočítáme si tedy *zkratky* – z vrcholu vede zkratka do nejdelšího jeho suffixu, který je jehlou. Na obrázku jsou vyznačeny dlouze čárkovanými šipkami.

Předpočítání zpětných hran časovou složitost konstrukce automatu jistě nezhorší, neboť vyžaduje v nejhorším případě projít všechny zpětné hrany ještě jednou.

Potřebujeme-li ohlásit všechny výskyty slov včetně pozice, kde se nacházejí, jsme hotovi. Výsledná časová složitost prohledávání bude $\mathcal{O}(S + O)$, resp. $\mathcal{O}(S \cdot \log|\Sigma| + O)$, kde O je velikost výstupu – počet výskytů všech slov.

Celková časová složitost prohledávání včetně stavby automatu tedy bude $\mathcal{O}(O + S + J \cdot |\Sigma|)$, resp. $\mathcal{O}(O + (S + J) \cdot \log|\Sigma|)$.

Jak velký může být výstup? Obecně až S^2. Extrémně velký výstup je možné vygenerovat třeba slovníkem obsahujícím všechny prefixy slova `AAAA...A` délky S a senem taktéž `AAAA...A` délky S. Automat pak hlásí výskyt pro každé podslovo, kterých je řádově S^2.



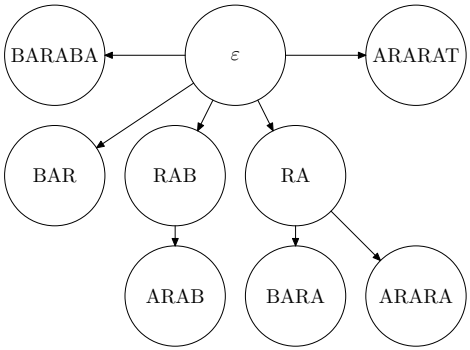
Pokud nám stačí u každého slova jen počet výskytů, nemusíme zoufat – závislost na počtu výskytů umíme odstranit.

Použijeme trik – na každé pozici započítáme pouze nejdelší jehlu, která tam končí (u každé jehly si budeme udržovat čítač). Nebudeme tedy v každém kroku poskakovat po zkratkách až do aleluja, ale maximálně jednou. Díky tomu nám z časové složitosti zmizí velikost výstupu.

V našem příkladu se senem `BARABARARAT` tedy na konci budeme mít uloženo, že `ARAB` se vyskytnul 1×, `ARARA` 1×, `ARARAT` 1×, `BAR` 2×, `BARA` 2× a `BARABA` 1×. `RA` a `RAB` nemají hlášený žádný výskyt.

Nyní si zkonstruujeme strom jenom ze zkratek a pro každý vrchol spočítáme součet celého jeho podstromu.

Tedy po přepočtu bude mít `RA` tři výskyty a `RAB` jeden výskyt; celkový počet výskytů pak bude 12.



Poznámky

- Dalším krokem po KMP a Aho-Corasickové jsou konečné automaty a regulární výrazy, o kterých jsme měli seriál ve 23. ročníku.
- Není moc rozumné snažit se implementovat Aho-Corasickovou v rozumné době například při soutěži, pokud tento algoritmus nemáte opravdu pod kůží. Radši zkuste použít hešování, pokud budete něco takového potřebovat.

Cvičení

- Redukci o velikost výstupu můžeme provést i pro případ, kdy výstup nebudeme vypisovat, ale stačí nám mít jej uložený v paměti. Vymyslete vhodnou úpravu triku s čítačem.
- Zkuste si naimplementovat Aho-Corasickovou vlastnoručně ve svém oblíbeném jazyce, abyste si byli jisti, že doopravdy chápete všechny záludnosti tohoto algoritmu.

Martin Böhm, Jan Matějka, Martin Mareš a Petr Škoda

Vzorová řešení KSP

26-1-1 Blokující signály

Řekněme, že dokážeme zastavit nějaký signál v uzlu X. Co to znamená? To znamená, že můžeme z hlavního počítače poslat blokující signál, který do uzlu X dorazí ve stejný moment jako ten vadný.

Všimněme si, že když můžeme poslat z hlavního počítače signál, který dorazí do uzlu X v čase T, tak můžeme poslat i signál, který dojde kdykoliv potom. Můžeme totiž signál z hlavního počítače jednoduše vyslat později.

Ke každému uzlu X tedy existuje nějaký minimální čas T_0 takový, že dokážeme zablokovat každý vadný signál, který do uzlu X dorazí nejdříve v čase T_0. Speciálně pro hlavní počítač je T_0 rovno nule: pokud vadný signál prochází hlavním počítačem, stačí si na něj počkat.

Na každý uzel X, který je nějak spojený s hlavním počítačem, můžeme nejdříve dosáhnout v čase, za který stihneme přejít nejkratší cestou z hlavního počítače do uzlu X. Stačí si tedy zjistit délku nejkratších cest z počátku do všech ostatních uzlů, čímž získáme všechny časy T_0.

Tahle úloha je (jak si jistě zkušenější řešitelé všimli na první pohled) grafová. Na zjištění délek všech nejkratších cest z nějakého daného vrcholu se v obecném grafu dá použít třeba Dijkstrův algoritmus, který jde naprogramovat tak, aby seběhl v čase $\mathcal{O}(M + N \log N)$ na grafu s N vrcholy a M hranami. Protože ale v této síti jsou všechny hrany (neboli spojení mezi počítači) stejně dlouhé, můžeme nejkratší cesty najít prostým prohledáváním do šířky, což se stihne za $\mathcal{O}(M+N)$. Prohledávání do šířky si můžete představit jako postupné „oloupávání“ sítě: nejdříve utrhneme hlavní počítač, pak všechny počítače na něj napojené (tedy ve vzdálenosti 1), pak všechny napojené na ně (2 kroky daleko), a tak dále. Na detaily implementace se můžete podívat ve zdrojáku vzorového řešení.

Program (C): `http://ksp.mff.cuni.cz/viz/26-1-1.c`

Michal Pokorný

26-1-2 Přeskládání nákladu

Problém rozdělení kontejnerů do skladišť můžeme převést na obarvování neorientovaného grafu. Kontejnery budou vrcholy, doporučení budou hrany mezi nimi a obarvení vrcholů dvěma barvami bude znázorňovat jejich rozdělení do skladišť.

Naším cílem je obarvit vrcholy grafu tak, aby hran vedoucích mezi vrcholy stejné barvy (tedy nesplněných doporučení) byla nejvýše polovina.

Ve speciálním případě, kdy bychom měli slíbeno, že graf je čistě bipartitní (tedy že se dá rozdělit na dvě množiny, kde hrany vedou jen mezi množinami a ne uvnitř), bude obarvení vrcholů dvěma barvami triviální.

Dokud budeme mít nějaký neobarvený vrchol, budeme opakovat toto: obarvíme ho první barvou, všechny jeho sousedy druhou, všechny sousedy sousedů opět první a tak dále. Protože každý vrchol sousedí pouze s vrcholy z opačné partity, povedlo by se nám takto splnit všechna doporučení u každého z vrcholů.

KSP

Pro bipartitní grafy je to tedy snadné. Jak to ale bude v obecném případě? Už se nám asi nepovede splnit všechna doporučení, ale můžeme se pokusit splnit jich alespoň polovinu.[39]

Náš algoritmus bude pracovat po krocích a v každém kroku se bude lokálně pokoušet splnit alespoň polovinu doporučení. Nejdříve se podíváme, jak bude vypadat jeden jeho krok, a potom si dokážeme, že tím splníme alespoň polovinu doporučení i globálně.

Krok algoritmu:

řešení

1. Vezmi libovolný neobarvený kontejner.
2. Spočítej, kolik má sousedů které barvy.
3. Pokud má sousedů jedné barvy méně, obarvi ho touto barvou. Jinak libovolně.
4. Dokud nejsou všechny kontejnery obarveny, pokračuj bodem 1.

Pro účely dokazování přiřadíme každé doporučení jen jednomu z dvojice kontejnerů – tomu obarvenému později (tím si určitě nic nepokazíme, neboť jednotlivá doporučení ani jejich počet se nijak nezmění).

Každý z kontejnerů bude tedy mít svou vlastní množinu doporučení. Ale to jsou přesně ta doporučení, která jsme uvažovali v bodu 2 algoritmu a obarvením kontejneru jsme jich splnili alespoň polovinu. U každého kontejneru je tedy alespoň polovina doporučení splněna, takže v součtu přes všechny kontejnery musí být splněna také alespoň polovina doporučení. A tím je splněno i zadání.

Zbývá ještě časová a prostorová složitost. Vše, co si musíme pamatovat, je nějaký seznam vrcholů a ke každému vrcholu seznam jeho hran, takže paměťová složitost je $\mathcal{O}(N + M)$.

Časová složitost je mírně složitější. Provedeme N kroků algoritmu a v každém se můžeme podívat až na M hran (což by mohlo vést na $\mathcal{O}(MN)$), ale stačí si uvědomit, že celkově se za celý běh programu podíváme maximálně na $2M$ konců hran a tedy výsledná časová složitost bude jen $\mathcal{O}(N + M)$.

Program (C): `http://ksp.mff.cuni.cz/viz/26-1-2.c`

Jirka Setnička & Petra Pelikánová

[39] Pokud bychom jich však chtěli splnit co nejvíc, už by šlo o NP-úplnou úlohu.

26-1-3 Plynové kapsy

Jednoduché řešení

Pro každý dotaz prostě projdeme příslušný interval zleva doprava a pokud se aktuální znak bude shodovat s předchozím, přičteme jedničku. Toto řešení je ale pomalé.

Rychlé řešení

Připravíme si pomocné pole. Na pozici i budeme mít uložený počet bezpečných míst v intervalu $\langle 0, i\rangle$. Tomu se říká prefixový součet. Potom při dotazu na $\langle i, j\rangle$ stačí od j-té pozice odečíst $(i-1)$-tou. Tím od všech bezpečných pozic nalevo od pravého konce intervalu odečteme ty nalevo od levého konce intervalu, tedy zbudou nám jen ty pozice uvnitř intervalu.

A jak si toto pomocné pole spočítat? Velmi podobně tomu, jak jsme počítali výsledek při jednoduchém řešení. Vypravíme se od levého konce a pokaždé, když bude aktuální znak stejný, jako předchozí, zvětšíme si průběžný počet o jedna. A v každém kroku si aktuální průběžný součet uložíme.

Proč to funguje, je vidět z vysvětlení v druhém odstavci. Pomocné pole nám zabere lineární množství paměti s velikostí vstupní posloupnosti. Co se týče času, pak předvýpočet je lineární s délkou posloupnosti na vstupu (projdeme jej jednou zleva doprava a v každém políčku uděláme konstantní množství práce). Jeden dotaz zodpovíme v konstantním čase, protože jen odečteme dvě čísla.

Program (C++): `http://ksp.mff.cuni.cz/viz/26-1-3.cpp`

Lucka Mohelníková & Michal „Vorner“ Vaner

26-1-4 Oprava databáze

Vyřešme nejprve jednodušší variantu, totiž dvojné prvky. Pro každý prvek p_k v zadané posloupnosti můžeme vyzkoušet všechny dvojice předchozích prvků p_i, p_j a ověřit, zda náhodou jejich součet není p_k. Tím dostaneme řešení s časovou složitostí $\mathcal{O}(n^3)$. To ale není nejlepší řešení této úlohy.

Můžeme si všimnout, že zbytečně několikrát provádíme stejné součty. Co kdybychom si místo toho dané součty systematicky pamatovali a pak v nich jen vyhledávali? To můžeme udělat například binárním vyhledávacím stromem.

V něm si budeme uchovávat všechny možné součty dvojic prvků před aktuálním prvkem p_k. Tedy v moment zpracovávání prvku p_k v něm bume mít hodnoty součtů dvojic p_i, p_j pro $i, j < k$. Takže jen ověříme, zda je hodnota p_k obsažená ve stromě a pokud ano, tak p_k je dvojným prvkem. Nakonec přidáme do stromu všechny součty $p_k + p_i$ pro $i < k$ a pokračujeme prvkem p_{k+1}.

Toto řešení má časovou složitost $\mathcal{O}(n \cdot (\log n + n \log n)) = \mathcal{O}(n^2 \log n)$ a paměťovou složitost $\mathcal{O}(n^2)$.

Nyní pojďme vyřešit úlohu pro trojné prvky. Postupovat budeme velmi podobně. Opět budeme mít binární vyhledávající strom uchovávající součty zatím potkaných dvojic, akorát budeme rozdílně zpracovávat prvek p_k. Pro něj budeme předpokládat, že je třetím prvkem v součtu a pro všechny $j > k$ ověříme, zda je možné pomocí součtu dvou prvků před p_k dostat hodnotu $p_j - p_k$.

Pokud ano, tak prvek p_j je trojným prvkem. To zjistíme dotazem na binární vyhledávací strom. Pak stejně jako předtím do stromu přidáme všechny součty tvořené prvkem p_k a některým předchozím prvkem a přesuneme se s výpočtem na další prvek posloupnosti.

Řešení má časovou složitost $\mathcal{O}(n^2 \log n)$, protože provádíme $\mathcal{O}(n^2)$ operací s binárním vyhledávacím stromem. Při programování použijeme knihovní implementaci binárního vyhledávacího stromu, například v jazyce C++ to je `set` z knihovny STL. Celá realizace řešení je pak kratší, než tento slovní popis. :-)

Program (C++): `http://ksp.mff.cuni.cz/viz/26-1-4.cpp`

Karel Tesař & Mark Karpilovskij

Medvědí poznámky

Dvojné prvky jde hledat o něco rychleji. Postupně procházíme přes všechna j a zjišťujeme, zda je součet prvku p_j s nějakým prvkem nalevo od něj roven nějakému prvku napravo od něj. Za tímto účelem si budeme udržovat dva setříděné seznamy: L bude obsahovat hodnoty ležících nalevo od aktuálního prvku, P ty napravo.

Pro každé j spočítáme $S_j = L + p_j$ (seznam vzniklý přičtením p_j ke každému prvku z L). To je také setříděný seznam, takže jeho sléváním s P můžeme snadno zjistit, zda S_j a P mají nějaký společný průnik, čili dvojný prvek. Všechny tyto operace zvládneme pro jedno j provést v lineárním čase, celkově tedy v $\mathcal{O}(n^2)$. Paměti spotřebujeme pouze $\mathcal{O}(n)$.

Vyhledávací stromy jsou mocná zbraň, kterou je dobré ovládat, ale občas lze věci řešit i jednodušeji. Jako třeba zde. Pro hledání trojných prvků postačí předpočítat si všechny možné součty dvojic, setřídit si je a pro každou hodnotu součtu si zapamatovat její nejlevější výskyt. Pak můžeme namísto ve stromu binárně vyhledávat v této setříděné posloupnosti a podle pozice nejlevějšího výskytu snadno ověřit, zda součet leží před zkoumaným p_k, anebo až za ním.

Pokud bychom se ovšem spokojili s algoritmem, který je rychlý v průměru a ne nutně v nejhorším případě, hodí se místo stromu použít hešovací tabulku – ta pracuje v průměrně konstantním čase na operaci, čímž se časová složitost hledání trojných prvků sníží na $\mathcal{O}(n^2)$. Najdeme ji i v STL pod názvem `unordered_set`.

Martin „Medvěd“ Mareš

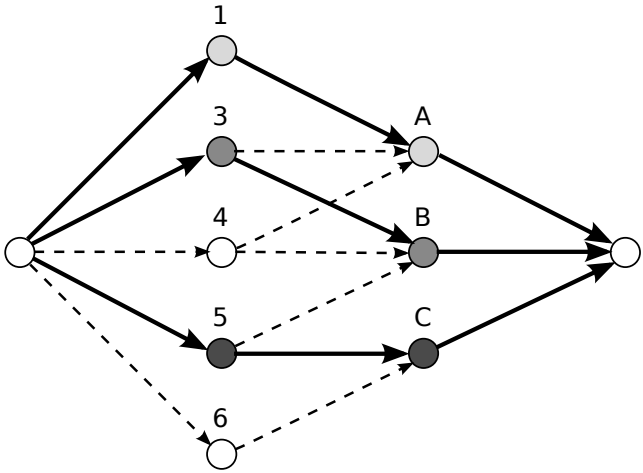
26-1-5 Senzory

Zkoušení všech možností tady moc efektivní nebude. Pojďme úlohu trochu rozebrat, aby se na ni jednodušeji útočilo.

První učiněné pozorování bude následující: můžeme věže rozmístit do řádků a do sloupečků nezávisle. Když se totiž věže ohrožují, tak se ohrožují buď v řádku, nebo ve sloupečku, a naopak pokud je rozmístění věží správné v řádcích i ve sloupečcích, je správné i celkově. Zredukovali jsme si tedy zadání na jeho jednodušší verzi, ve které se věže staví do jednoho řádku, dvě nesmí stát ve stejném sloupci, a každá může stát jenom v nějakém vymezeném intervalu. Vyřešíme tyhle podúlohy pro sloupce a pro řádky zvlášť a výsledky spojíme.

Řešení pomocí systému různých reprezentantů

Jednodušší podúloha je speciální případ hledání takzvaného *systému různých reprezentantů*. Mějme třeba množiny $A = \{1, 3, 4\}, B = \{3, 4, 5\}, C = \{5, 6\}$. Systém různých reprezentantů je takové přiřazení prvků množin k jejich množinám, že všechny vybrané prvky jsou různé. Příklad systému různých reprezentantů pro tyto množiny A, B, C je třeba $A \to 1, B \to 3, C \to 5$ (ale třeba $A \to 3, B \to 3, C \to 6$ nebo $A \to 5, B \to 3, C \to 6$ už ne). Obecně se systémy různých reprezentantů dají hledat přes toky v sítích. Vytvoříme si bipartitní graf, ve kterém jedna partita budou prvky množin $(1, 3, 4, 5, 6)$ a druhá budou množiny (A, B, C) a natáhneme hrany s kapacitou 1 mezi prvkem x a množinou M tam, kde $x \in M$.



Potom si vytvoříme zdroj, ze kterého povedou hrany kapacity 1 do všech prvků, a stok, do kterého povedou hrany kapacity 1 ze všech množin. V této síti najdeme maximální tok například pomocí Fordova-Fulkersonova algoritmu. Ma-

ximální tok nám ukáže hledaný systém různých reprezentantů (pokud existuje). Kapacity hran a podmínky, které musí tok splňovat, zajišťují, že všechny prvky i všechny množiny budou použity maximálně jednou.

Je to zcela správný způsob řešení úlohy o rozmisťování věží. Podrobnosti hledání maximálního toku tu však nenajdete, protože úloha jde řešit jednodušeji a rychleji. Máte-li o ně zájem, můžete je najít v kuchařce o tocích.[40]

Jednodušší řešení

Zkusíme to takzvaně hladově: můžeme nejdříve vybrat věž, kterou umístíme na první sloupeček, pak ze zbylých vybrat tu, kterou umístíme na druhý, a tak dále (samozřejmě přeskakujeme sloupečky bez věží). Jak si ale budeme věže vybírat? Překvapivě tu bude fungovat metoda „do prvního sloupečku vyber tu věž, se kterou jde nejméně hýbat“ (pojmenujme ji třeba *minimální věž*). To, že takový algoritmus bude fungovat, nahlédneme indukcí.

Důkaz správnosti

Indukci začneme třeba od triviálního případu s jednou věží: tu můžeme umístit hned na první sloupeček, na který může přijít, takže tam algoritmus funguje.

Indukční krok bude schematicky takovýhle: „Když nějaká věž má prázdný interval, je jasné, že žádné řešení neexistuje. Řekněme teď bez újmy na obecnosti, že na první sloupeček jde umístit nějaká věž. Vybereme z věží, které jdou dát do prvního sloupečku, libovolnou minimální, a položíme ji tam. Tím si zmenšíme zadání o jednu věž a jeden sloupeček. Necháme si od indukce přihrát řešení menšího problému. Pokud existuje, přidáme k němu tuhle věž a máme výsledek. Pokud neexistuje, pak neexistuje ani řešení problému včetně minimální věže.“

Zbývá teď dokázat, že pokud budeme věže takhle umisťovat, tak o žádné řešení nepřijdeme. (Důkaz toho, že žádné řešení nepřidáme, je jednoduchý.) Jinými slovy: pokud jdou věže nějak rozmístit podle zadání, tak jdou rozmístit i tak, že v prvním sloupečku bude z věží, které tam šly umístit, libovolná minimální.

Vezměme si nějaké řešení a vyberme si z věží, které můžou dostat první sloupeček, nějakou minimální. Označíme ji třeba M. Nechť M nedostane první sloupeček, ale sloupeček S_M. Pokud je první sloupeček v řešení volný, můžeme do něj M přemístit, čímž dostaneme řešení, ve kterém první sloupeček drží vybraná minimální věž. Pokud není první sloupec volný, znamená to, že nějaká věž X jej drží. Protože M je minimální věž, tak interval věže X obsahuje mimo jiné sloupeček S_M. Můžeme tedy věže X a M beztrestně prohodit. Po prohození opět dostaneme řešení, které má v prvním sloupci minimální věž.

Implementace

Máme dokázáno, že to bude fungovat, a zbývá to „jenom“ implementovat. Budeme postupně ukrajovat věže a sloupečky. V proměnné si budeme držet po-

[40] `http://ksp.mff.cuni.cz/viz/kucharky/toky-v-sitich`

slední sloupeček, do kterého jsme už umístili věž (a další věže budeme umisťovat jenom za něj).

Věže chceme brát v takovém pořadí, abychom pokaždé umisťovali tu, která je v nezpracovaném prostoru minimální. Jak toho dosáhneme? Mohli bychom si v každém kroku najít minimální věž průchodem všech zbylých věží, ale to by nás stálo kvadratický čas. Existuje lepší řešení: ukládat si věže do haldy. Halda konkrétně bude minimová a budeme v ní třídit podle konce intervalu, do kterého můžeme věž umístit. Také do ní věže nebudeme přidávat všechny hned, ale teprve okamžikem, kdy narazíme na začátek jejich intervalu. Kdykoliv z téhle haldy tedy odebereme věž s nejmenším koncem intervalu, bude to právě ta minimální pro sloupeček, který budeme zabydlovat.

Jako první krok věže v $\mathcal{O}(N \log N)$ setřídíme vzestupně podle minima intervalu. Potom budeme postupně zleva zaplňovat sloupečky a přidávat věže do haldy.

Každou věž zpracujeme takto: jde-li umístit hned za poslední umístěnou věž, učiníme to. Nejde-li to, zkusíme ji umístit na začátek jejího intervalu. Pokud ani to nejde, znamená to, že poslední umístěná věž je za koncem intervalu právě zpracovávané věže, takže zpracovávaná věž nejde umístit nikam – řešení neexistuje. Nakonec nesmíme zapomenout zvýšit proměnnou s poslední umístěnou věží.

A co nás to bude stát? Paměť je jednoduchá: stačí $\mathcal{O}(N)$ na uložení vstupu a haldy. Co se času týče: setřídění věží zabere $\mathcal{O}(N \log N)$. Každou věž také jednou uložíme do haldy a jednou z ní vybereme, což obojí trvá $\mathcal{O}(\log N)$ za věž, tedy celkem $\mathcal{O}(N \log N)$. Všechno ostatní trvá kratší dobu.

Program (C++): `http://ksp.mff.cuni.cz/viz/26-1-5.cpp`

Michal Pokorný

26-1-6 Hydroponie

Zahrada skýtá přihrádky na rostliny označené čísly 0 až $N-1$. K dispozici máme M nápajecích okruhů, i-tý okruh je intervalem $I_i = [a_i, b_i) = \{a_i, a_i + 1, \ldots, b_i - 1\}$. Délka intervalu v tomto značení je $b_i - a_i$.

Pokoušíme se určit počet všech možných osazení rostlin do přihrádek tak, aby každý interval obsahoval alespoň jednu rostlinu. Protože mohou být výsledné hodnoty extrémně vysoké, počítáme pouze zbytek po dělení této hodnoty číslem 1 000 000 007.

Rozklad na schodiště

Proveďme úvodní pozorování. Platí-li pro navzájem různá i a j inkluze $I_i \subseteq I_j$, můžeme interval I_j ze svých úvah vypustit. Podmínku na obsazení rostliny v tomto intervalu nám zajistí interval I_i.

Dále budeme uvažovat pouze intervaly, které v sobě neobsahují žádný celý interval. Intervaly si navíc uspořádáme vzestupně dle jejich počátku. Pro libo-

volné $i < j$ budou nyní platit nerovnosti $a_i < a_j$ a $b_i < b_j$. (První nerovnost je dána uspořádáním, druhá tím, že I_j nemůže celý ležet v I_i.)

Získali jsme jakousi schodovitou strukturu. Tuto strukturu si rozebereme na jednotlivá schodiště. Schodiště S_i bude množinou intervalů I_u až I_v takovou, že každé dva po sobě jdoucí intervaly mají neprázdný průnik, navíc S_i bude vždy největší takovou možnou množinou. Tedy I_{u-1} a I_{v+1}, pokud existují, jsou disjunktní s I_u a I_v. Počet schodišť označme jako T.

Povšimněme si, že úloha se nám nyní rozpadla na $T + 1$ zcela nezávislých podúloh. Pro přihrádky neobsažené v žádném intervalu nejsme ničím vázáni, tedy máme 2^K možností jejich obsazení, je-li K počtem těchto přihrádek. Zbývajícími podúlohami jsou jednotlivá schodiště.

Výpočet pro jedno schodiště

Mějme jediné schodiště s intervaly $J_0, \ldots, J_{\ell-1}$. Nyní označujme $J_i = [a_i, b_i)$.

Zužitkujeme myšlenku dynamického programování a budeme si postupně počítat hodnoty $D(0)$ až $D(\ell - 1)$, kde $D(i)$ udává počet možných rozmístění květin, umisťujeme-li je pouze do intervalů 0 až i.

Počáteční hodnota $D(0)$ je zřejmě $2^{|J_0|} - 1$, neb nám je zakázáno pouze to obsazení květin v intervalu J_0, při kterém neumístíme ani jednu květinu.

Předokládejme nyní, že hodnoty $D(0), \ldots, D(i-1)$ již byly spočteny, a my chceme určit $D(i)$.

Buď p nejmenší takové, že $b_p > a_i$. Interval J_i rozdělíme na podintervaly $[a_i, b_p)$, $[b_p, b_{p+1})$, $[b_{p+1}, b_{p+2}), \ldots, [b_{i-1}, b_i)$.

Možnosti osazení květinami si rozdělíme podle toho, do kterého z těchto podintervalů umístíme nejpravější květinu. Nejprve se podívejme na intervaly tvaru $[b_j, b_{j+1})$ pro nějaké j splňující $p \leq j < i$. Později se ještě podíváme na speciální interval $[a_i, b_p)$. Výsledky pro jednotlivé intervaly pak sečteme a získáme $D(i)$.

Máme interval $[b_j, b_{j+1})$, ve kterém bude umístěna alespoň jedna rostlina, a víme, že na pozicích od b_{j+1} dále už žádná květina nebude. Možností, jak umístit květiny do tohoto podintervalu, je $2^{b_{j+1}-b_j} - 1$. Kolika způsoby lze korektně osadit zbytek schodiště udává $D(j)$, což dává celkem $\left(2^{b_{j+1}-b_j} - 1\right) \cdot D(j)$ možností pro tento interval.

Zbývá nám interval $[a_i, b_p)$. Je-li $p = 0$, pak je mezivýsledkem hodnota $2^{a_0-a_i} \cdot (2^{a_i-b_0} - 1)$. Musili jsme obsadit alespoň jednu květinu do intervalu $[a_i, b_0)$, zbytek intervalu J_0 jsme mohli obsadit libovolně.

Pro $p > 1$ se ještě podíváme na interval J_{p-1}. Pro ten už platí $b_{p-1} \leq a_i$. Počet možných osazení intervalu $[a_i, b_p)$ je opět $2^{a_i-b_0} - 1$. Interval $[b_{p-1}, a_i)$ lze obsadit libovolně, tedy $2^{a_i-b_{p-1}}$ způsoby. Jak obsadit všechny pozice před tímto intervalem už udává $D(p-1)$. Celkem tedy $\left(2^{a_i-b_0} - 1\right) \cdot 2^{a_i-b_{p-1}} \cdot D(p-1)$.

Celkový počet možných osazení tohoto schodiště je $D(\ell - 1)$.

Plody našeho snažení

Protože jsou umístění do jednotlivých T schodišť a do přihrádek mimo schodiště nezávislá, získáme výsledek jako součin počtu možností pro tyto jednotlivé případy.

Zbývá se zamyslet, s jakou složitostí umíme celé naše řešení implementovat. Abychom si intervaly správně uspořádali a zbavili se přebytečných, stačí je setřídit a vhodně projít. (Detaily si rozmyslete sami, nebo si je přečtěte ve vzorové implementaci.)

Asymptotická složitost algoritmu bude funkcí dvou proměnných, jmenovitě N a M. V takovém případě nemusí být jednoznačné, která časová složitost je tou optimální. Může to záviset na vztahu mezi těmito proměnnými. (Optimální algoritmus by asi měl podle vstupních hodnot N a M zvolit správnou variantu implementace).

Předvedeme si dvě možné složitosti řešení. V prvním případě intervaly setřídíme v čase $\mathcal{O}(M \log M)$, třeba pomocí Quicksortu.

Během počítání možností pro všechna schodiště mocníme dvojku a to exponentem z rozsahu 0 až N. Tuto operaci jsme schopni provést v čase $\mathcal{O}(\log N)$.

Při výpočtu možností pro daný interval ve schodišti sčítáme možnosti podintervalů. Všimněme si, že tento součet není potřeba počítat vždy od nuly, ale stačí jej aktualizovat při zvyšování i a p. To znamená lineární počet operací vzhledem k počtu intervalů. Nejdražší operací je už zmíněné mocnění dvojky.

První řešení má tedy časovou složitost $\mathcal{O}(M \log N)$ a paměťovou složitost $O(M)$. (Všimněte si, že $\mathcal{O}(\log M)$ a $\mathcal{O}(\log N)$ je totéž, protože M je nejvýše N^2.)

Protože okraje intervalů jsou z rozsahu 0 až $N-1$, můžeme je také setřídit přihrádkovým tříděním v čase $\mathcal{O}(N)$. Všechny mocniny dvojky si můžeme dopředu předpočítat a pak na dotaz odpovídat v konstantním čase. Tak získáváme druhé řešení s časovou i paměťovou složitostí $\mathcal{O}(N + M)$.

Při implementaci nesmíme zapomenout všechny hodnoty průběžně nahrazovat jejich zbytkem po dělení 1 000 000 007, abychom nedostávali ohromná čísla.

Vzorová implementace ukazuje řešení v čase $\mathcal{O}(M \log N)$.

Program (C): `http://ksp.mff.cuni.cz/viz/26-1-6.c`

Lukáš Folwarczný

26-1-7 Mravenci

Pokusme se soustředit na srážku určitých dvou mravenců. Lze si všimnout, že pokud si je neoznačíme, vypadá situace tak, jako by se míjeli a k žádné srážce nedošlo. Mravenec, který spadne z klacku poslední, je tedy ten, který je nejvzdálenější od okraje klacku, ke kterému je otočený. Pozor, takový mravenec je vždy aspoň jeden, ale můžou být i dva!

U druhé části nám přibylo několik těžkostí. Nemůžeme už totiž zanedbat označení mravenců. Nahlédneme ale, že mravenci se na klacku nemohou přeskakovat. Speciálně je tedy prvních n mravenců spadlých z klacku na levé straně totožných s těmi n mravenci, kteří jsou na začátku nejblíže levému konci klacku. Pro pravou stranu platí analogické tvrzení.

Z první části již víme, do jakého směru je na začátku orientován mravenec, jenž spadne jako poslední (pro jednoduchost dále předpokládejme, že doleva, jinak analogicky). Nyní zjistíme počet mravenců, kteří na začátku míří doleva. Řekněme, že je jich k. Pak mravenec, který z klacku spadne jako poslední, je k-tý mravenec zleva v pořadí, v jakém stojí mravenci na začátku.

KSP

Jan Bok

26-1-8 Turingova strojovna

řešení

První tři úkoly byly snadné, čtvrtý trochu těžší. Ukázalo se ale, že všechny čtyři skrývají nečekané hlubiny. Začneme proto snadnými řešeními a pak se spolu vydáme na cestu do hlubin.

Úkol 1

Nejprve si všimneme, že každá neprázdná správně uzávorkovaná posloupnost obsahuje po sobě jdoucí dvojici `()`. Smazáním této dvojice vytvoříme jiné správné uzávorkování, v němž opět najdeme takovou dvojici, a tak dále, až posloupnost zredukujeme na prázdnou. Naopak začneme-li se špatným uzávorkováním, smazáním `()` z něj nikdy nevytvoříme správné. Zjistili jsme, že správná uzávorkování jsou právě ta, která lze zredukovat na prázdnou posloupnost.

Přesně o to se bude pokoušet náš stroj. Pracovat bude nad abecedou $\{(,), *\}$ a jeho program bude vypadat následovně:

stav/znak	(	)	*	␣
S	$(\,(, \rightarrow, S)$	$(*, \leftarrow, P)$	$(*, \rightarrow, S)$	$(\,␣, \leftarrow, K)$
P	$(*, \rightarrow, S)$	—	$(*, \leftarrow, P)$	NE
K	NE	—	$(*, \leftarrow, K)$	ANO

Začne ve stavu S, bude procházet řetězcem zleva doprava a hledat první pravou závorku. Jakmile ji najde, přepíše ji na `*` a přepne se do stavu P, v němž se bude vracet zpátky a hledat nejbližší levou. Tu také vyhvězdičkuje, načež se přepne opět do stavu S (všimněte si, že vlevo od aktuální pozice už žádné pravé závorky nejsou, takže se k nim není potřeba vracet). Až vstupní řetězec dojde, stroj přejde do stavu K, v němž bude kontrolovat, zda nezbyly nějaké nespárované levé závorky.

Časová složitost tohoto stroje je $\mathcal{O}(n^2)$, jelikož až řádově n-krát potřebujeme najít párovou závorku, což trvá až řádově n kroků. Kvadraticky dlouho bude

počítat například na vstupu `(((...)))`. Kromě prostoru na pásce, kde byl napsán vstup, nepotřebuje žádnou další paměť.

Úkol 2 s lineární pamětí

Sledujme, jak probíhá výpočet stroje z předchozího úkolu. Vyhvězdičkovaná políčka pásky můžeme ignorovat, ta stroj vždy přeskakuje. Pak platí, že vlevo od aktuální pozice jsou samé levé závorky – to jsou ty, ke kterým jsme ještě nenašli pravou závorku do páru. Každá další závorka je buďto levá (a pak ji přeskočíme, čímž se přesune do levé části), nebo pravá (a tehdy naopak z levé části jednu levou závorku smažeme).

Jinými slovy levou část používáme jako zásobník dosud neuzavřených závorek. Na vícepáskovém stroji si ho můžeme uložit na samostatnou pásku. Tím zařídíme, že další znak v zásobníku bude dostupný v konstantním čase.

Program stroje bude vypadat takto:

$$
\begin{aligned}
(S, (, \sqcup) &\to (((, \to), ((, \to), S)\\
(S,), \sqcup) &\to ((), \to), (\sqcup, \leftarrow), P)\\
(S, \sqcup, \sqcup) &\to ((\sqcup, \bullet), (\sqcup, \leftarrow), K)\\
(P, \alpha, ( ) &\to ((\alpha, \bullet), (\sqcup, \bullet), S)\\
(P, \alpha, \sqcup) &\to \text{NE}\\
(K, \sqcup, \sqcup) &\to \text{ANO}\\
(K, \sqcup, ( ) &\to \text{NE}
\end{aligned}
$$

Stroj začne ve stavu S. Na první pásce se bude pohybovat zleva doprava a postupně číst závorky ze vstupu. Na druhé pásce si bude udržovat zásobník otevřených závorek a hlava se v klidovém stavu bude nacházet vpravo od poslední uložené závorky.

Pokud stroj přečte levou závorku, uloží ji na zásobník.

Pokud přečte pravou závorku, přepne se do stavu P a zkontroluje, jestli na zásobníku má nějakou levou. Pakliže ano, odstraní ji a vrátí se do počátečního stavu. Pokud ne, závorkování není správné.

Skončí-li vstup, je ještě potřeba zkontrolovat, že na zásobníku nezůstala žádná neuzavřená závorka. O to se stará stav K.

Tento stroj pracuje v lineárním čase s délkou vstupu (každý znak zpracuje v konstantním čase) a spotřebuje lineárně mnoho prostoru na pracovní pásce.

Úkol 2 s logaritmickou pamětí

Zadání vybízelo k co nejmenší spotřebě paměti. Vskutku, předchozí řešení prostorem vysloveně plýtvá. Do zásobníku ukládá samé levé závorky, takže by úplně stačilo pamatovat si jejich počet ve dvojkové soustavě. Uložíme ho jako posloupnost znaků `0` a `1` na pracovní pásce. Nejnižší řád se bude nacházet vpravo a v klidovém stavu bude hlava stát na mezeře napravo od něj.

Obsluha vstupní pásky bude vypadat následovně:

$$\begin{aligned}(S, (, \sqcup) &\to ((\,(, \to), (\sqcup, \leftarrow), I)\\ (S,), \sqcup) &\to ((\,), \to), (\sqcup, \leftarrow), D)\\ (S, \sqcup, \sqcup) &\to ((\sqcup, \bullet), (\sqcup, \leftarrow), K)\end{aligned}$$

Stavy I, D a K znamenají „zvyš počítadlo o 1“ (inkrementace), „sniž počítadlo o 1“ (dekrementace) a „závěrečná kontrola, zda počítadlo je 0“. V nich už se budeme zabývat jen pracovní páskou s počítadlem, takže je popíšeme jako jednopáskový Turingův stroj. Navíc jsme přidali stav Z, který slouží k návratu hlavy doprava.

KSP

stav/znak	`0`	`1`	$\sqcup$
I	$(\mathtt{1}, \to, Z)$	$(\mathtt{0}, \leftarrow, I)$	$(\mathtt{1}, \to, Z)$
D	$(\mathtt{1}, \leftarrow, D)$	$(\mathtt{0}, \to, Z)$	NE
K	$(\mathtt{0}, \leftarrow, K)$	NE	ANO
Z	$(\mathtt{0}, \to, Z)$	$(\mathtt{1}, \to, Z)$	$(\sqcup, \bullet, S)$

řešení

Nyní spotřebujeme pouze logaritmické množství prostoru, neboť dvojkové číslo v rozsahu 0 až n zapíšeme pomocí $\lfloor \log_2 n \rfloor + 1$ bitů. Ovšem zvyšování a snižování počítadla trvá logaritmicky dlouho, takže jsme si časovou složitost pokazili na $\mathcal{O}(n \log n)$.

Úkol 3

Pokud si můžeme dovolit přidávat pásky, vystačíme si dokonce s jediným stavem. Rozšíříme abecedu o tolik symbolů, kolik měl původní stroj stavů. Přidáme novou pásku, z níž budeme používat jen jediné políčko. Na něm budeme udržovat informaci o tom, jaký stav původního stroje právě simulujeme. A aby se nový stroj správně rozběhl, určíme, že prázdné políčko odpovídá počátečnímu stavu původního stroje.

Úkol 4

Na jednopáskovém stroji se nabízí rozšířit abecedu na uspořádané dvojice (z, s), kde z je znak původní abecedy a s stav původního stroje. Jenže: pokud posuneme hlavu na sousední políčko, musíme tam přenést informaci o stavu, která byla zakódovaná do stávajícího políčka. To nejde udělat najednou (protože do stavu nového stroje zakódujeme jen 1 bit informace), ale s trochou šikovnosti poskytují dva stavy dost manévrovacího prostoru na to, abychom informaci přenesli po částech.

Stavy nového stroje nazveme X a Y. Abecedu rozšíříme na trojice (z, s, m), přičemž z a s budou odpovídat znaku a stavu původního stroje (stavy očíslujeme) a m bude *mód,* na němž bude záviset, co zrovna X a Y znamenají. Módů budeme rozeznávat pět: *klid, vysílání doprava, vysílání doleva, příjem zprava, příjem zleva.*

Představme si, že nový stroj právě odsimuloval jednu instrukci starého stroje. Na aktuálním políčku změnil s i z a teď se potřebuje posunout na sousední políčko, řekněme doprava. Udělá toto:

- Na aktuální políčko zapíše mód *vysílání doprava,* přepne se do stavu X a posune hlavu doprava.
- Sousední políčko bylo v módu *klid,* takže si stav X vyloží jako požadavek, který přišel zleva. Nastaví $s = 0$, přejde do módu *příjem zleva* a ve stavu X se posune zpět doleva.
- Nyní je předávání informací nastartováno. Vysílací políčko pokaždé sníží své s o 1 a dokud nevyjde 0, přesouvá se na přijímací políčko ve stavu X. Přijímací políčko zvýší své s o 1 a vrátí hlavu zpět (stále stav X). Pokračujeme v předávání.
- Jakmile vysílací políčko dopočítá do nuly, přepne svůj mód na *klid* a posune hlavu na přijímací políčko, tentokrát ve stavu Y. Podle toho přijímací políčko pozná, že přenos je u konce, a odsimuluje další instrukci původního stroje.

Pokud chceme přenášet informaci doleva místo doprava, postupujeme obdobně, jen v prvním kroku použijeme stav Y, podle čehož přijímací políčko pozná, že přenášíme zprava.

Každou instrukci původního stroje tedy umíme odsimulovat konstantně mnoha instrukcemi stroje nového.

Úkol 4: start výpočtu

Právě předvedené řešení 4. úkolu má jeden malý, leč podstatný háček: jak se vlastně celý výpočet rozběhne? Potřebujeme přeci, aby byl ve znaku na prvním políčku pásky zakódován počáteční stav S_0 původního stroje.

Jak to zařídit? Máme možnost určit pro každý znak původní abecedy, jaká trojice mu bude odpovídat v nové abecedě, a také si můžeme vybrat počáteční stav nového stroje.

Hned se nabízí zapisovat znaky původní abecedy jako trojice (z, S_0, klid). Jenže ani pro počáteční stav X, ani pro Y to nedopadne dobře: stroj se bude snažit kopírovat stav ze sousedního políčka, které na to vůbec není připraveno. Tak raději necháme nový stroj, ať svůj výpočet zahájí zapsáním stavu S_0. Jak ale pozná, kdy to má udělat?

Půjdeme na to menší oklikou. Nejprve si rozmyslíme, že každý Turingův stroj můžeme předělat tak, aby nikdy nevyužíval políčka pásky nalevo od počáteční polohy hlavy (tedy aby jeho páska byla jen jednostranně nekonečná). Zařídíme to „přeložením pásky napůl“. Políčka původní pásky si očíslujeme $\ldots, -3, -2, -1, 1, 2, 3, \ldots$ a na i-té políčko nové pásky uložíme uspořádanou dvojici znaků z původních políček i a $-i$.

Předělaný stroj bude simulovat instrukce původního stroje a navíc si bude ve svém stavu pamatovat, zda se nachází v kladné či záporné části původní pásky. Podle toho bude používat buď první, anebo druhou složku dvojice a případně obracet směr pohybu hlavy. Navíc si na políčko 1 umístíme značku, abychom poznali, že jsme přešli přes rozhraní kladné a záporné části.

Tato transformace zpomalí výpočet pouze konstanta-krát a má jeden příjemný důsledek, kterého vzápětí využijeme: vstoupíme-li na jakékoli políčko poprvé, je to vždy zleva.

Nyní se vraťme zpět k redukci počtu stavů. Každý znak z původní abecedy zakódujeme jako trojici (z, S_0, init). Nový mód *init* se chová stejně jako *klid* a navíc říká, že jsme na políčku poprvé. Proto víme, že během výpočtu nového stroje nemůžeme na takové políčko přijít ve stavu Y (ten by totiž znamenal „jdeme kopírovat zprava"). Y tedy prohlásíme za počáteční stav nového stroje a kombinaci mód *init* + stav Y využijeme k rozjezdu stroje.

Heuréka, problém vyřešen. Každý jednopáskový Turingův stroj umíme upravit tak, aby počítal totéž (až na změnu abecedy), zpomalil se jenom konstantakrát a vystačil si přitom s pouhými dvěma stavy. (Přesněji řečeno, předvedli jsme to pro stroje, které odpovídají stavem ANO nebo NE. Pokud by výstup vydávaly na pásce, museli bychom ještě na závěr výpočtu pásku „vyčistit" a překódovat zpět do vstupní abecedy. Ale to už je maličkost.)

Pro přehlednost ještě ukážeme tabulku, co který stav znamená ve kterém módu:

mód	**stav** X	**stav** Y
init	začne příjem zleva	start stroje
klid	začne příjem zleva	začne příjem zprava
vysílání	chci pokračování	—
příjem	zvýšit číslo stavu	konec vysílání

Poznámka: Dodejme ještě, že inicializaci lze provést i jinak. Využijeme toho, že jsme ve vysílacích módech nepotřebovali stav Y. Pojďme i tam stavem rozlišovat, zda jsme přišli zleva nebo zprava. Navíc máme možnost v každém módu poprohazovat, co znamená X a co Y. Pak zafunguje následující trik.

Na počáteční políčko přijdeme ve stavu X, takže si políčko myslí, že má přijímat zleva. Požádá proto políčko vlevo od sebe o pokračování vysílání. Jenže levé políčko o žádném vysílání neví, takže to interpretuje jako žádost o příjem zprava. Přejde tedy doprava s žádostí o vysílání. Můžeme zařídit, aby si pravé políčko tuto žádost vyložilo jako konec vysílání, čili přijalo stav 0. Ten zpracujeme speciálně: skočíme doleva a také předáme konec vysílání. I levé políčko ukončí příjem přijetím stavu 0, ale umí to rozlišit, protože přišel zprava, takže ho může zpracovat jinak speciálně: zahájením přenosu skutečného počátečního stavu stroje doprava. Kouzlo se zdařilo.

Zpět k úkolu 1: rychlejší řešení

Trik s počítadlem z úkolu 2 se dá využít i na jednopáskovém stroji. Jen si musíme dát pozor, kam počítadlo uložíme: pokud na začátek pásky (před vstup), budeme ke konci vstupu potřebovat spoustu kroků na přesuny mezi vstupem a počítadlem; počítadlo za koncem vstupu se bude chovat podobně špatně na začátku výpočtu. Kam tedy? Pořídíme si počítadlo stěhovavé: bude umístěno

těsně před dosud nezpracovanou částí vstupu a po zpracování každé další závorky ho celé přestěhujeme o jednu pozici doprava.

Program stroje bude vypadat následovně:

stav/znak	`(`	`)`	`0`	`1`	␣
S	$(\texttt{(}, \leftarrow, I)$	$(\texttt{)}, \leftarrow, D)$	—	—	$(\sqcup, \leftarrow, K)$
I	—	—	$(\texttt{1}, \leftarrow, L)$	$(\texttt{0}, \leftarrow, I)$	$(\texttt{1}, \leftarrow, L)$
D	—	—	$(\texttt{1}, \leftarrow, D)$	$(\texttt{0}, \leftarrow, L)$	NE
L	—	—	$(\texttt{0}, \leftarrow, L)$	$(\texttt{1}, \leftarrow, L)$	$(\sqcup, \rightarrow, c_s)$
c_s	—	—	$(\sqcup, \rightarrow, c_0)$	$(\sqcup, \rightarrow, c_1)$	—
c_0	$(\texttt{0}, \rightarrow, S)$	$(\texttt{0}, \rightarrow, S)$	$(\texttt{0}, \rightarrow, c_0)$	$(\texttt{0}, \rightarrow, c_1)$	—
c_1	$(\texttt{1}, \rightarrow, S)$	$(\texttt{1}, \rightarrow, S)$	$(\texttt{1}, \rightarrow, c_0)$	$(\texttt{1}, \rightarrow, c_1)$	—
K	—	—	$(\texttt{0}, \leftarrow, K)$	NE	ANO

Stroj opět začíná ve stavu S. Jakmile zmerčí levou závorku, posune se těsně před aktuální pozici, kde je uloženo počítadlo, a začne ho inkrementovat. Přitom setrvává ve stavu I, dokud dochází k přenosu do vyšších řádů. Jakmile přenosy ustanou, přejde do L a pohybuje se směrem k levému okraji počítadla. Nakonec pomocí stavů c_0, c_1 a c_s celé počítadlo přesune o znak doprava a vrátí se zpět do S.

Podobně reaguje na pravou závorku, jen k tomu používá dekrementovací stav D. Na konci výpočtu jako obvykle pomocí stavu K zkontroluje, že počítadlo vyšlo nulové.

Časová složitost tohoto řešení je $\mathcal{O}(n \log n)$, protože pro každý znak vstupu strávíme $\mathcal{O}(\log n)$ kroků zvyšováním či snižováním a následně přesunem logaritmicky dlouhého počítadla. Paměti zabíráme stále $\mathcal{O}(n)$.

Zpět k úkolu 2: trocha naděje

Také vám vrtá hlavou, jestli by nešlo nějak zkřížit naše dvě řešení druhého úkolu a dosáhnout současně lineárního času a logaritmické paměti? Jistá naděje tu je. Pozorujme, jak se mění číslice počítadla, když ho opakovaně inkrementujeme:

$$0000 \to 000\mathbf{1} \to 00\mathbf{10} \to 001\mathbf{1} \to 0\mathbf{100} \to 010\mathbf{1} \to 01\mathbf{10} \to$$
$$\to 011\mathbf{1} \to \mathbf{1000} \to 100\mathbf{1} \to 10\mathbf{10} \to 101\mathbf{1} \to \ldots$$

Pro přehlednost jsme změněné číslice vyznačili tučně.

Pokaždé se jedničky na konci čísla změní na nuly a nula před nimi na jedničku. Nebo jinak: jedna jednička vznikne a možná nějaké zaniknou. Pokud provedeme m inkrementů, vzniklo celkem m jedniček. Každá z nich zanikla nejvýš jednou, takže všech zániků dohromady je také nejvýš m. Všech m inkrementů tedy zabralo čas $\mathcal{O}(m)$.

Tehdy říkáme, že jeden inkrement má konstantní *amortizovanou časovou složitost.* Hezky se to popisuje pomocí *penízkové metody.* Jelikož čas jsou jak

známo peníze, zavedeme mezi nimi směnný kurs. Jeden *penízek* bude představovat dostatek času na provedení jedné operace našeho stroje: tedy přepsání nuly na jedničku či opačně, včetně případného pohybu hlavy.

Nyní prohlásíme, že na provedení jednoho inkrementu po uživateli chceme 2 penízky. Jeden z nich použijeme na vytvoření jedničky, druhý dáme nově vzniklé jedničce do vínku a ona si z něj časem zaplatí své smazání. Za m inkrementů tedy zaplatíme $2m$ penízků a každou operaci stroje jsme naúčtovali některému z inkrementů.

KSP

Dokud tedy při kontrole závorek potkáváme jen ty levé, časová složitost stroje je lineární. Jenže... pravá závorka způsobí snížení počítadla a to nám celý elegantní amortizační argument zboří: počítadlo může libovolně dlouho střídat hodnoty `0111...1` a `1000...0`, což nutně zabere logaritmický čas.

Úkol 2: dvojková soustava vrací úder

Tak snadno se přeci nevzdáme. Dvojkovou soustavu rozšíříme, aby byla *vyvážená*. Vedle číslic 0 a 1 použijeme navíc -1 (pro zkrácení zápisu budeme místo 1 psát `+` a místo -1 prostě `-`). Váhy řádů budou stále mocniny dvojky, takže číslo $c_k c_{k-1} \ldots c_1 c_0$ bude mít hodnotu $\sum_{i=0}^{k} 2^i \cdot c_i$.

řešení

Dvojkový zápis čísla funguje i ve vyvážené dvojkové soustavě, ale totéž číslo může mít i jiné zápisy. Například 6 (`110` dvojkově) se dá zapsat jako `++0`, ale i `+-+0` nebo `+-0-0`, jakož i mnoha dalšími způsoby.

Přesto z první číslice poznáme, zda je číslo kladné nebo záporné. Nechť první číslice je `+` a má váhu 2^k. Potom ani kdyby byly všechny ostatní číslice záporné, nepřispějí dohromady tolik, aby se celé číslo vynulovalo, nebo dokonce přehouplo do záporna. Platí totiž $2^0 + 2^1 + \ldots + 2^{k-1} = 2^k - 1$. Podobně je-li první číslice `-`, musí být číslo nutně záporné.

Z toho speciálně plyne, že jediné možnosti, jak zapsat nulu, jsou řetězce číslic `0`. Cokoliv jiného je buď kladné, nebo záporné.

Inkrementování čísla provedeme takto: půjdeme zprava doleva. Pokud potkáme `0`, změníme ji na `+` a zastavíme se. Potkáme-li `-`, změníme ho na `0` a zastavíme se. Narazíme-li na `+`, přepíšeme ho na `0` a stejně jako v klasické dvojkové soustavě provedeme přenos do vyššího řádu (o číslici vlevo).

Dekrementování je symetrické: z `0` uděláme `-`, z `+` uděláme `0`, z `-` vytvoříme `0` a přenos.

Posloupnost inc, inc, inc, inc, dec, dec, inc tedy projde hodnotami `0`, `+`, `+0`, `++`, `+00`, `+0-`, `+-0`, `+-+`.

Nyní nahlédneme, že inkrementování i dekrementování má konstantní amortizovanou složitost. Penízky budeme tentokrát přiřazovat všem nenulovým číslicím. Inkrementování si nechá od uživatele zaplatit 2 penízky. Pokud přepíše `0` na `+`, zaplatí to z uživatelova penízku a ten druhý dá do vínku vzniklému `+`. Změní-li `+` na `0`, zaplatí to z penízku toho `+` a pokračuje ve výpočtu. A pokud pře-

píše - na 0, zaplatí to z penízku toho - a dva uživatelovy penízky může prohýřit. Dekrementování se chová obdobně, též si vystačí se dvěma penízky.

K vyřešení úlohy použijeme Turingův stroj, který bude fungovat obdobně jako naše předchozí řešení s dvojkovým počítadlem, jen použijeme vyváženou dvojkovou soustavu. Popíšeme jen obsluhu počítadla, zbytek stroje zůstane stejný. V klidovém stavu budeme opět udržovat hlavu vpravo od počítadla. Navíc aby se nám snadno testovala nulovost počítadla, budeme nevýznamné nuly ze začátku čísla při každé příležitosti mazat.

Program stroje vypadá takto:

stav/znak	0	+	-	␣
I	$(+, \rightarrow, Z)$	$(0, \leftarrow, I)$	$(0, \leftarrow, N)$	$(+, \rightarrow, Z)$
D	$(-, \rightarrow, Z)$	$(0, \leftarrow, N)$	$(0, \leftarrow, D)$	NE
Z	$(0, \rightarrow, Z)$	$(+, \rightarrow, Z)$	$(-, \rightarrow, Z)$	$(␣, \bullet, S)$
N	$(0, \rightarrow, Z)$	$(+, \rightarrow, Z)$	$(-, \rightarrow, Z)$	$(␣, \rightarrow, N')$
N'	$(␣, \rightarrow, Z)$	—	—	—
K	NE	NE	NE	ANO

Stav I je jako obvykle inkrementovací, stav D dekrementovací. V obou stroj upravuje počítadlo tak dlouho, dokud dochází k přenosu. Pak se přepne do stavu Z, v němž se vrací na konec pásky. Pokud na pásku zapíše 0, odskočí si ještě do stavu N, v němz zkontroluje, zda tato nula neleží na začátku čísla, a případně ji smaže. Stav K slouží k závěrečné kontrole nulovosti počítadla – jelikož nevýznamné nuly průběžně mažeme, postačí testovat prázdnost pásky.

Dosáhli jsme tedy lineární časové a logaritmické prostorové složitosti. Na závěr dodejme, že logaritmický prostor je skutečně zapotřebí, ale důkaz je trochu pracnější a do okraje této stránky by se nevešel ☺

Úkol 2: kočkopsí řešení

Ukážeme ještě jedno optimální řešení druhého úkolu. Je technicky pracnější, ale myšlenkově prostší: Šikovně zkřížíme první řešení (počítadlo v jedničkové soustavě, lineární čas, lineární paměť) s druhým (dvojková soustava, čas $\mathcal{O}(n \log n)$, paměť $\mathcal{O}(\log n)$).

Opět budeme udržovat počítadlo rozdílu levých a pravých závorek. Počítadlo bude dvojkové, ale budeme ho aktualizovat po skocích. Nejprve spočítáme $\ell = \lceil \log_2 n \rceil$.

Vstup budeme zpracovávat po blocích velikosti ℓ. Pro každý blok budeme udržovat počítadlo v jedničkové soustavě. Zajímat nás bude jeho hodnota na konci bloku a také nejnižší záporná hodnota během bloku. To vše zjistíme v čase $\mathcal{O}(\ell)$ a prostoru taktéž $\mathcal{O}(\ell)$.

Hodnotu počítadla na konci bloku převedeme do dvojkové soustavy a přičteme ji ke globálnímu počítadlu. Před tím ještě ověříme, že nejnižší záporná hodnota nepřesáhla předchozí hodnotu globálního počítadla. Všechny tyto ope-

race stihneme v $\mathcal{O}(\ell)$ – tolik bitů mají dvojková čísla, s nimiž pracujeme. Převod z jedničkové do dvojkové soustavy zařídíme postupným přičítáním jedničky, které, jak už víme, trvá $\mathcal{O}(1)$ amortizovaně.

Každý blok tedy zpracujeme v čase $\mathcal{O}(\ell)$ a prostoru $\mathcal{O}(\ell)$. Všech $\mathcal{O}(n/\ell)$ bloků v čase $\mathcal{O}(n)$ a prostoru opět $\mathcal{O}(\ell) = \mathcal{O}(\log n)$.

Martin „Medvěd“ Mareš

26-2-1 Zamotané provazy

Prvním postřehem může být, že počet křížení musí být vždy sudý, jinak nemůže být provaz 2 nahoře na obou stromech. Dále si lze všimnout, že když je na dvou po sobě jdoucích kříženích nahoře stejný provaz, jde jen o přehozenou „vlnu“, a lze ji bez problémů rozmotat.

Pokud budeme postupně odstraňovat dvojice stejných křížení, dobereme se k jednomu ze dvou stavů. Pokud nám zbyly dva provazy bez křížení, jde je rozmotat. Jinak budou tvořit posloupnost, ve které se pouze střídají jedničky a dvojky, a kterou již rozmotat nelze.

Velmi se nabízí možnost implementovat program pomocí zásobníku. Každé nové překřížení porovnáme s vrcholem zásobníku a pokud se liší, přidáme nové křížení. Pokud jsou stejné, vrchol odebereme. Takto se určitě dostaneme ke každé dvojici, kterou bychom odebrat mohli. Už teď máme algoritmus lineární s počtem křížení a toto stačilo na zisk 6 bodů.

Lépe než lineárně to samozřejmě nepůjde, každé křížení ovlivňuje výsledek. Pomůžeme si ale s pamětí a to vynecháním zásobníku. Můžeme si to dovolit, protože se v něm po sobě jdoucí prvky vždy liší – jde vždy o střídající se jedničky a dvojky. Stačí si tedy pamatovat, čím zásobník končí a jak je hluboký.

Optimálním řešením je tedy výše popsaný algoritmus běžící v čase $\mathcal{O}(N)$ a využívající $\mathcal{O}(1)$ paměti. Za takové řešení jsme už dávali plný počet bodů.

Program (C): `http://ksp.mff.cuni.cz/viz/26-2-1.c`

Ondra Hlavatý & Jirka Setnička

26-2-2 Barevný trojúhelník

V řešení se inspirujeme myšlenkou, kterou nám zaslal řešitel Matej Lieskovský. Předpokládejme pro spor, že v síti neexistuje souvislá oblast spojující všechny tři strany. Vezměme horní vrchol trojúhelníka a uvažme největší jednobarevnou souvislou oblast, ve které leží.

Tato oblast, řekněme bez újmy na obecnosti bílá, se dotýká minimálně dvou stěn, kterých se dotýká vrchol. Z předpokladu se však nedotýká třetí. Je tedy zespoda zcela ohraničena souvislým pásem černých šestiúhelníků, který navíc spojuje ty samé dvě stěny, které spojuje zvolená bílá oblast. To opět z předpokladu znamená, že tato černá oblast nezasahuje do třetí strany.

Nahlédněme, že když celou bílou oblast u horního vrcholu přebarvíme na černo, nevznikne tím souvislá oblast spojující tři strany – tato oblast spojuje stejné dvě strany jako její černá hranice, takže spojení se třetí stranou nevznikne. Tím jsme ovšem zvýšili aspoň o 1 počet černých šestiúhelníků a dostali jsme opět obrazec bez souvislé oblasti spojující všechny 3 strany.

Tuto operaci můžeme opakovat kolikrát chceme, ovšem po určitém počtu kroků musíme dojít do stavu, kdy už bude černý celý trojúhelník. To je ovšem obrazec, ve kterém už zřejmě existuje souvislá jednobarevná oblast spojující všechny tři strany, což je hledaný spor. Tím je důkaz hotov.

Alternativní interpretace stejné myšlenky by fungovala následovně: Pomocí pozorování výše snadno ukážeme, že v daném obarvení existuje souvislá oblast spojující všechny tři strany **právě tehdy, když** existuje taková oblast v obarvení, kde přebarvíme onu souvislou oblast při některém vrcholu.

KSP

Když tedy budeme dostatečně dlouho takto přebarvovat, opět dostaneme zcela černý trojúhelník, který už takovou všespojující oblast obsahuje. Z tohoto stavu ovšem můžeme do stavu, kde jsme začali, natáhnout řetěz výše popsaných ekvivalencí, takže existuje-li ve finálním stavu oblast spojující všechny tři strany, existuje i v původním obrazci.

Mark Karpilovskij

řešení

26-2-3 Plánování cesty

Úloha o plánování Jacobovy cesty nebyla ničím jiným, než hledáním nejkratší cesty v grafu. Přeformulujme si úlohu nejprve z řeči čtvercových políček do čisté řeči grafů.

Definujme orientovaný ohodnocený graf G. Vrcholy grafu budou políčka zadané oblasti. Hrany mezi dvěma políčky vedou právě tehdy, když spolu obě políčka sousedí stranou a obě jsou průchozí. Ohodnocení hrany z políčka A do políčka B bude odpovídat času potřebnému na zdolání políčka B.

Průchod do šířky

Graf G má RS vrcholů a $\mathcal{O}(RS)$ hran, kde $R \times S$ jsou rozměry oblasti. Řešením původní úlohy je délka nejkratší cesty v G ze startovního políčka do cílového políčka.

Prvním způsobem, jak úlohu řešit, je prohledat graf do šířky. Prohledávání do šířky (BFS) je popsáno v naší grafové kuchařce[41] a zde jej nebudeme opakovat.

Nesmíme ovšem zapomenout, že prohledávání do šířky funguje pouze v případech, kdy ohodnocení všech hran je jednotkové. Nejprve musíme ještě provést jednu úpravu grafu. Využijeme celočíselnosti ohodnocení hran a každou hranu nahradíme posloupností jednotkových hran. Konkrétně hranu s ohodnocením t nahradíme cestičkou složenou z t jednotkových hran. Takové operaci se také říká *podrozdělení*.

Pokud označíme jako T nejvyšší z ohodnocení všech hran, můžeme počet hran i vrcholů nového grafu omezit výrazem $\mathcal{O}(TRS)$. Nejkratší cesta v podrozděleném grafu odpovídá nejkratší cestě v původním grafu.

[41] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

Na tomto grafu už použijeme prohledávání do šířky a získáme řešení s časovou složitostí $\mathcal{O}(TRS)$.

Dodejme ještě, že si můžeme vystačit s prostorem velikosti $\mathcal{O}(RS)$. Nově přidané vrcholy a hrany si totiž nemusíme nutně pamatovat. Stačí pracovat s původním G a akorát při vkládání vrcholu do fronty přidělit vrcholu jakési „zpoždění“, se kterým má být zpracován. Zpoždění nejprve nastavíme na ohodnocení hrany. Po vyjmutí vrcholu z fronty zpoždění snížíme o jedna. Je-li zpoždění stále větší než nula, vrchol nezpracujeme, ale znovu jej vložíme do fronty.

Dijkstrův algoritmus

S pomocí průchodu do šířky bylo možno zdárně vyřešit prvních pět vstupů, v dalších pěti testovacích vstupech již byla hodnota T příliš vysoká a bylo nutno využít sofistikovanější algoritmus.

Mezi takové algoritmy se například řadí Dijkstrův algoritmus, který je vlastně rozvinutím předchozí myšlenky o zpožděných vrcholech. Dijkstrův algoritmus je důkladně vyložen v další z našich kuchařek.[42] Kuchařka shodou okolností vychází letos zároveň se zadáním třetí série a bylo by zbytečné zde její obsah opakovat.

Použijeme-li při implementaci Dijkstrova algoritmu binární haldu, dosáhneme časové složitosti $\mathcal{O}((N+M)\log N)$, kde N je počet vrcholů a M počet hran grafu. V našem případě získáváme řešení s časovou složitostí $\mathcal{O}(RS\log RS)$. Paměťová složitost řešení bude $\mathcal{O}(RS)$.

Můžete si povšimnout, že použití k-regulární či Fibonacciho haldy by nám v našem konkrétním problému nepomohlo. Graf je totiž velmi řídký (má pouze lineární počet hran vzhledem k počtu vrcholů).

Program (C) – průchod do šířky:
`http://ksp.mff.cuni.cz/viz/26-2-3-bfs.c`

Program (C) – Dijkstrův algoritmus:
`http://ksp.mff.cuni.cz/viz/26-2-3-dijkstra.c`

Lukáš Folwarczný

26-2-4 Stavba věže

Lehčí varianta

Jakmile víme, že všech K kostek má stejnou váhu, můžeme si je setřídit podle jejich nosnosti. Poté je budeme postupně procházet od největší k nejmenší a budeme kontrolovat, že nosnost $\ell_i \geq K - i - 1$ (kde i je index v poli, které je číslováno od nuly).

[42] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`

Proč to můžeme udělat právě takto? Protože potřebujeme, aby každá kostka unesla všechny nad sebou. Ty váží právě $K - i - 1$ (pokud mají všechny jednotkovou váhu) a pokud bude tato nerovnost splněna pro každou kostku, tak takovou věž určitě postavit můžeme.

Pokud by naopak tato nerovnost na nějakém místě splněna nebyla, tak bychom aktuální kostku mohli prohodit jenom za nějakou kostku výše (protože kdybychom ji prohodili níž, musela by unést ještě víc). Jenže všechny kostky nad aktuální mají nosnost menší nebo rovnou aktuální, takže v takovém případě neexistuje způsob, jak by věž šla postavit.

Časová složitost takového řešení je $\mathcal{O}(K \log K)$, protože právě tak dlouho budeme třídit. Kontrola, že lze věž postavit, už proběhne jen v čase $\mathcal{O}(K)$. Taková řešení jsme ohodnotili slíbenými 3 body, tedy maximem za lehčí variantu.

Ale lehčí variantu lze vyřešit i v čase $\mathcal{O}(K)$. Uděláme to takto:

1. Vytvoříme pole o velikosti K (opět číslujeme od nuly).
2. Projdeme kostky a u těch, které mají $\ell_i \geq K$, nastavíme jejich nosnost na $K - 1$ (protože větší nosnost nepotřebujeme, stačí nám, aby unesly maximálně $K - 1$ kostek).
3. Postupně projdeme všechny kostky a do pole velikosti K si zapíšeme, kolik kostek má kterou nosnost (tedy budeme indexovat pole přímo nosností).
4. Poté si vybereme jednu kostku s nosností $K - 1$. Pokud taková neexistuje, je jasné že věž postavit nelze a zahlásíme neúspěch. Pokud existuje, tak ji smažeme a spustíme náš algoritmus znovu pro K o jedna menší.

Ale počkat, to bude přece trvat až $\mathcal{O}(K^2)$! Zkusíme tedy vzít jenom hlavní myšlenku a vylepšíme ji. Rozmysleme si, jak se nám pole po smazaní jedné kostky ve 4. bodě změní.

Kostky, které měly předtím nosnost $\ell_i < K - 1$, to nijak neovlivní, ty zůstanou na stejném místě. Jediné, co se změní, jsou kostky s nosností $\ell_i \geq K - 1$, tyto (až na tu jednu smazanou) dostanou nosnost $\ell_i = K - 2$.

Z toho nám vyplývá, jak naprogramovat řešení. První tři kroky uděláme přesně tak, jak je napsáno výše, ale 4. krok uděláme trochu chytřeji – uvědomíme si, že vlastně vůbec není nutné celé pole počítat pokaždé znova.

Stačí nám jednu kostku o nosnosti $K - 1$ odečíst a všechny kostky o stejné nosnosti, které nám zbyly, přičíst ke kostkám s nosností $K - 2$. Takto upravený 4. krok budeme opakovat, dokud se nedostaneme na $K = 0$. Pokud jsme cestou nenarazili na žádné potíže, tak lze celá věž postavit.

Těžší varianta

V případě, že mají kostky rozdílné váhy, nám už nebude stačit obyčejné třídění podle nosnosti. Jako protipříklad použijeme dvě kostky: $w_1 = 1$, $\ell_1 = 3$ a $w_2 = 4$, $\ell_2 = 2$. Podle nosností bychom chtěli nejdříve umístit první, která

už druhou neunese. Prohlásili bychom tedy, že věž nelze postavit. Druhá kostka však první unese, ale my bychom tuto možnost ani nevyzkoušeli.

Jak tedy úlohu vyřešit? Mnozí z vás správný postup vymysleli. Obtížnější však bylo dokázat, že řešení opravdu funguje. Ukážeme si kromě algoritmu a důkazu i to, jak můžeme na takové řešení přijít postupně.

Věž stavíme odspoda v jednotlivých krocích. Na začátku máme věž nulové výšky a *hromádku* všech kostek. V každém kroku z hromádky vybereme ty kostky, které unesou všechny zbývající kostky na hromádce. Vybrané kostky můžeme přidat v libovolném pořadí do věže.

Pokud se nám podaří tímto způsobem věž postavit ze všech kostek, tak máme vyhráno, protože jsme nikdy nepoužili takovou kostku, která by neunesla vše nad sebou. Pokud věž nesestavíme, tak jsme nalezli hromádku, ze které ani jedna kostka neunese všechny zbývající. Z této hromádky však nelze nikdy postavit věž – žádná kostka nemůže být základnou věže. Přidáním dalších kostek si celou konstrukci můžeme zatížit, nikdy ji však neodlehčíme. Věž tedy nejde postavit ani ze všech kostek.

Okamžitě tak dostáváme algoritmus, který úlohu řeší v čase $\mathcal{O}(K^3)$. Provedeme totiž nejvýše K kroků, kde v každém z nich spočítáme pro každou kostku součet hmotností všech zbývajících.

Nyní si můžeme všimnout, že při sčítání hmotností ostatních kostek sčítáme dokola téměř ta samá čísla. Lepší tedy bude si předem spočítat součet hmotností všech kostek na hromádce, označme jej třeba S. Do věže potom můžeme přidat kostky, pro které platí $S - w_i \leq \ell_i$, tedy ekvivalentně $S \leq \ell_i + w_i$.

Když se nyní podíváme na vývoj S po jednotlivých krocích, zjistíme, že se číslo S pouze snižuje. Nic nám tedy nebrání kostky vybírat v pořadí od největší po nejmenší podle součtu $w_i + \ell_i$.

Algoritmus díky tomu můžeme velmi zjednodušit a urychlit. Kostky nejprve setřídíme podle součtu hmotnosti a nosnosti. A následně v lineárním čase ověříme, zda se kostky unesou. K tomu stačí, když půjdeme od špičky věže dolů a vždy porovnáme nosnost aktuální kostky se součtem hmotností kostek nad ní. Tento součet si v průběhu snadno spočítáme z předchozího pouhým přičtením hmotnosti další kostky. Celý algoritmus tak doběhne v čase $\mathcal{O}(K \log K)$ a spotřebuje při tom $\mathcal{O}(K)$ paměti.

Program (C) – lehčí varianta:
`http://ksp.mff.cuni.cz/viz/26-2-4-lehci.c`

Program (C++) – těžší varianta:
`http://ksp.mff.cuni.cz/viz/26-2-4-tezsi.cpp`

Vojta Sejkora & Jenda Hadrava

26-2-5 Vyvažování

Vyhledávací stromy lze vyvažovat mnoha různými algoritmy, které všechny pracují v lineárním čase, ovšem liší se množstvím potřebné paměti. Začneme tím nejobyčejnějším, který potřebuje lineární pracovní prostor, a postupně se propracujeme až ke konstantní paměti.

Rozebrat a složit: lineární prostor

KSP

Přetvářet nevyvážený strom na vyvážený pomocí lokálních úprav vypadá složitě. Co kdybychom ho prostě rozebrali a pak znovu poskládali?

Rozebírání proběhne rekurzivním průchodem stromu: vždy projdeme nejprve levý podstrom, pak kořen a nakonec pravý podstrom. Takto jednotlivé vrcholy navštívíme ve vzestupném pořadí. Stačí si je tedy průběžně ukládat do pole.

Ze setříděného pole posléze vyrobíme dokonale vyvážený strom: kořenem bude prvek, který v poli leží uprostřed (tedy medián). Hodnoty ležící v poli před ním patří do levého podstromu, hodnoty za ním do pravého. Oba podstromy sestrojíme rekurzivně a zavěsíme pod kořen. Do rekurze přitom stačí předávat počáteční a koncový index úseku pole, ze kterého zrovna stavíme strom.

řešení

Jak rozebrání starého stromu, tak postavení nového nás stojí $\mathcal{O}(n)$, kde n je počet vrcholů stromu. Na co všechno spotřebujeme paměť? Předně si musíme uložit lineárně velké pole hodnot. Nesmíme také zapomenout na zásobník od rekurze: na něm bude najednou tolik položek, kolik činí hloubka stromu. Ta může při rozebírání dosáhnout až n, během skládání pak pouze $\mathcal{O}(\log n)$. Prostoru tedy celkem zabereme $\mathcal{O}(n)$.

Rozebrání stromu na seznam

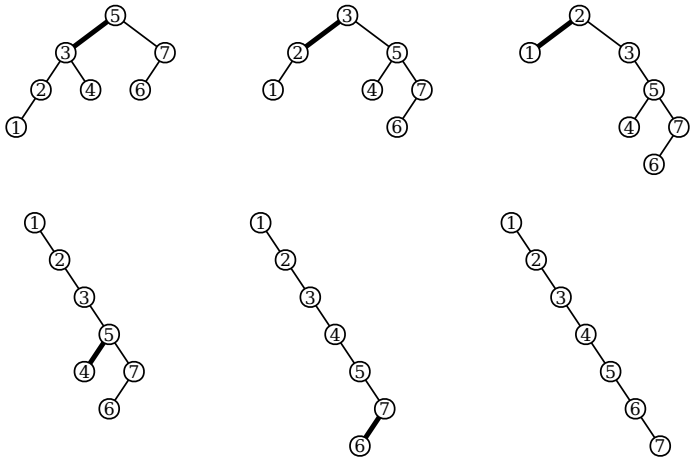
Pojďme se zbavit zbytečné kopie hodnot. Místo kopírování vrcholy zadaného stromu popřepojujeme, aby tvořily *liánu,* tedy strom, v němž nejsou žádní leví synové. Ukazatele na pravé syny nám tedy tvoří obyčejný spojový seznam, navíc setříděný podle hodnot.

K převodu stromu na liánu se budou hodit rotace (viz kuchařka).[43] Ty umožňují měnit tvar stromu, a přitom stále zachovávají uspořádání vrcholů (strom je tedy stále vyhledávací).

Budeme postupovat takto: dokud má kořen levého syna, provádíme v kořeni rotaci doprava. Když už levého syna nemá, sestoupíme do jeho pravého syna a tam algoritmus opakujeme.

Ukažme si, jak to vypadá pro jeden strom se 7 vrcholy (tučně je vždy vyznačena hrana, kterou se chystáme rotovat):

[43] `http://ksp.mff.cuni.cz/viz/kucharky/vyhledavaci-stromy`



Proč to funguje? Stačí si všimnout, že mezi kořenem a aktuálním vrcholem leží nějaká už sestrojená liána. Rotace ji nepokazí a po konečně mnoha rotacích (každá zmenšuje velikost levého podstromu) učiníme jeden krok doprava, který liánu prodlouží. Algoritmus se tedy musí zastavit a v tu chvíli je celý strom liánou.

Kolik celkem strávíme času? Kroků doprava je lineárně. Abychom ukázali, že rotací také, stačí sledovat, jak se vyvíjí délka *pravé cesty.* To je cesta vedoucí z kořene doprava, dokud to jde. Každá rotace ji prodlouží o 1, ovšem délka cesty nikdy nepřekročí n, takže všech rotací je nejvýše n.

Logaritmické řešení

Nyní upravíme funkci na převod seznamu na strom, aby si vystačila se seznamem namísto pole.

Problematické místo je hledání prostředního prvku: v poli byl přístupný v konstantním čase, v seznamu bychom ho hledali lineárně dlouho, čímž bychom si pokazili celkovou časovou složitost.

Místo toho rekurzivní funkci navrhneme tak, aby dostala jako parametry seznam S a počet prvků k. Funkce odpojí prvních k prvků seznamu, vytvoří z nich strom T a vrátí jak tento strom, tak zbytek seznamu S''.

V pseudokódu bychom ji zapsali třeba takto:
Strom(S, k):

1. Je-li $k = 0$, vrátíme prázdný strom a seznam S.
2. $\ell \leftarrow \lfloor (k-1)/2 \rfloor$
3. $(L, S') \leftarrow$ Strom(S, ℓ)
4. $x \leftarrow$ odpojíme první prvek seznamu S'

5. $(P, S'') \leftarrow \textsc{Strom}(S', k - \ell - 1)$
6. Vytvoříme strom T: kořen bude mít hodnotu x, jeho levým podstromem bude L a pravým P.
7. *Výstup:* Dvojice (T, S'').

Čas je opět lineární, paměti nám stačí $\mathcal{O}(\log n)$ na zásobník.

Konstantní prostor pro úplné stromy

Ani logaritmický prostor ovšem není nutný. Ukážeme řešení v konstantním prostoru, zatím ovšem pouze pro *úplné stromy.* Tak budeme říkat stromům, v nichž mají všechny vnitřní vrcholy právě 2 syny a všechny listy leží na téže hladině. Úplný strom hloubky h má tedy

$$2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$$

vrcholů. (Jiná možnost, jak úplné stromy definovat, je ještě zesílit definici dokonalé vyváženosti a požadovat, aby levý a pravý podstrom byly pokaždé přesně stejně velké. Proto se jim také někdy říká *perfektní stromy.*)

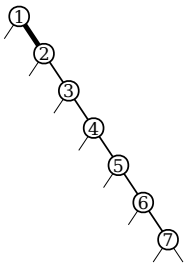
Dostaneme tedy seznam délky mocnina dvojky minus 1 a máme z něj vyrobit úplný strom.

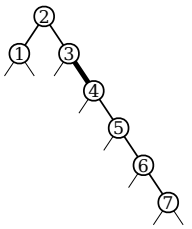
Algoritmus bude jednoduchý, ale nečekaný. Jeho průběh sledujme na obrázcích níže (tučně je opět zvýrazněna hrana, kterou rotujeme; vlasové čáry pod vrcholy zatím ignorujte).

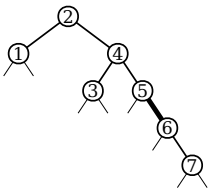
Půjdeme z kořene doprava a v každém kroku provedeme jednu rotaci doleva. Tím nám z cesty vznikne „hřeben“ – pravá cesta poloviční délky, z jejichž vrcholů vedou levé odbočky do listů.

Poté průchod z kořene doprava zopakujeme. Pravá cesta se opět dvakrát zkrátí a na levých odbočkách budou už viset „třešničky“ – úplné stromy hloubky 1 se třemi vrcholy. Další průchod vytvoří úplné stromy hloubky 2 se sedmi vrcholy a tak dále.

Nakonec celou pravou cestu rozebereme a zbude nám jeden úplný strom.







Času jsme spotřebovali lineárně: hran na pravé cestě je na počátku $n-1$ a každá rotace jednu odstraní. Prostor potřebujeme pouze na konstantně mnoho pracovních proměnných.

Konstantní prostor pro všechny stromy

Dobrá, ale co když počet vrcholů není tak hezké číslo, aby šlo sestrojit úplný strom? Tehdy najdeme nejbližší menší číslo m tvaru 2^i-1, z tolika prvků seznamu sestrojíme úplný strom a zbylých $r = n - m$ vrcholů pověsíme pod listy úplného stromu.

Zařídí se to snadno: pokud k některým vrcholům počáteční liány přivěsíme levé syny a spustíme algoritmus pro úplné stromy, tak se tyto „přívěsky“ objeví právě pod listy úplného stromu. (To znázorňují ony vlasové čáry v obrázku, které jsme napoprvé přehlíželi.)

Náš obecnější algoritmus tedy projde liánu a celkem r-krát provede rotaci doleva, aby se z části vrcholů staly přívěsky a zbyla pravá cesta délky m, na kterou půjde spustit původní algoritmus.

Oněch r rotací se samozřejmě budeme snažit rozmístit co nejrovnoměrněji. Uděláme to takto: pořídíme si proměnnou, která bude na počátku nulová, v každém kroku k ní přičteme r/m a kdykoliv překročí 1, tak zrotujeme a odečteme 1.

Dokonce se můžeme zbavit dělení: místo původní proměnné si budeme pamatovat její m-násobek. Začneme tedy na nule, pokaždé přičteme r a pokud překročíme m, zrotujeme a odečteme m. (Mimochodem, přesně tento postup se používá při aproximování úseček pomocí pixelů na obrazovce.)

Zbývá dokázat, že skutečně vyjde dokonale vyvážený strom. Pro každý vrchol má platit, že rozdíl velikostí levého a pravého podstromu činí nejvýše 1. Jelikož v úplném stromu jsou oba podstromy stejně velké, stačí ukázat, že se počty přívěsků pod nimi liší nejvýše o 1.

Všimneme si, že vrcholy obou podstromů tvoří v liáně souvislé úseky, navíc stejně dlouhé. Ukážeme, že pro každé dva úseky stejné délky platí, že se počty přívěsků pod nimi liší nejvýše o 1.

Počítejme, kolik náš algoritmus mohl vygenerovat přívěsků pro úsek délky u. Nechť pomocná proměnná řídící přivěšování má na začátku úseku hodnotu h (víme, že $0 \le h < m$). Pak vytvoříme celkem $\lfloor (h + ur)/m \rfloor$ přívěsků. Tento výraz je nejmenší pro $h = 0$ a největší pro $h = m - 1$, přičemž obě krajní hodnoty se mohou lišit nejvýše o 1. (Neboť pro každé x platí $\lfloor x+1 \rfloor = \lfloor x \rfloor + 1$.)

Výsledný strom je tedy dokonale vyvážený a na jeho vytvoření nám postačil lineární čas a konstantní paměť.

Program (C) – řešení pomocí rotací:
`http://ksp.mff.cuni.cz/viz/26-2-5-rotace.c`

Uvedený algoritmus pochází z článku *Tree Rebalancing in Optimal Time and Space* od Quentina F. Stouta a Bette L. Warrenové, na který nás upozornili řešitelé. Aneta Šťastná navíc navrhla jednodušší rozmisťování přívěsků, kterým jsme se také inspirovali.

Alternativní řešení

Když jsme úlohu zadávali, měli jsme vymyšlený úplně jiný způsob řešení. Jeho rozbor se všemi detaily je trochu pracnější, základní myšlenka ovšem také stojí za zmínku.

Upravíme logaritmické řešení, aby nepotřebovalo tolik paměti na zásobník. Budeme předpokládat, že každý vrchol si kromě ukazatelů na syny pamatuje i ukazatel na otce (časem se ukáže, že to není potřeba).

Představme si nejprve, jak libovolný strom projít ve vzestupném pořadí hodnot bez použití rekurze. Začneme v kořeni a řídíme se následujícími pravidly:

- Pokud lze jít doleva, jdeme doleva.
- Pokud už nelze jít doleva, jdeme nahoru. Pokud jsme přišli zleva, vypíšeme aktuální hodnotu a pokračujeme doprava. Pokud jsme přišli zprava, pokračujeme nahoru.

Při konstrukci stromu si budeme počínat obdobně: vytvářený strom budeme takto „obcházet“ a místo abychom vrcholy vypisovali, tak je budeme zakládat.

To se snadno řekne, ale při implementaci nás čeká několik překážek.

Předně potřebujeme udržovat plánovanou velikost podstromu (proměnná k v logaritmickém řešení). Při kroku dolů ji prostě dělíme dvěma, leč při kroku nahoru nestačí násobit dvěma, neboť původní hodnota mohla být lichá. To vyřešíme tak, že si v každé úrovni rekurze zapamatujeme zbytek po dělení dvěma. To je jeden bit na každé z $\mathcal{O}(\log n)$ úrovní, takže se všechny dají poskládat do jediného čísla velkého řádově n.

Další potíž je, že občas potřebujeme přejít přes vrchol, který jsme ještě nevytvořili. To není těžké, ale při programování to řádně zamotá hlavu. Postačí, když budeme udržovat nejbližší vyšší vrchol, který skutečně existuje, a aktuální hloubku.

Co víc, k neexistujícímu vrcholu také občas potřebujeme něco připojit. Tak to připojíme k onomu nejbližšímu vyššímu existujícímu a navíc si zapamatujeme, ze které strany připojujeme (to je opět bit na úroveň).

Konečně si potřebujeme pro každou úroveň pamatovat, jestli aktuální vrchol na této úrovni ještě není vytvořen (tzn. zrovna jsme zalezlí kdesi v jeho levém podstromu), nebo už existuje (levý podstrom hotov, teď jsme někde v pravém).

Tímto způsobem také dosáhneme lineárního času a konstantní paměti.

Program (C) – alternativní řešení:
`http://ksp.mff.cuni.cz/viz/26-2-5-pruchod.c`

Dva pointery místo tří

Na závěr jedna pozoruhodná perlička: způsob uložení binárního stromu se dvěma pointery na vrchol, který umožňuje v konstantním čase zjistit levého syna, pravého syna i otce libovolného vrcholu.

První pointer (říkejme mu třeba P) bude ukazovat na levého syna. Druhý (řečený Q) bude v levém synovi ukazovat na jeho pravého sourozence, ale v pravém synovi na otce.

Levého syna vrcholu tedy zjistíme pomocí P. Pravého pomocí Q z levého syna. Pokud chceme znát otce, zkusíme přejít pomocí Q a podíváme se, zda jsme se dostali do vrcholu s menší hodnotou. Pokud ano, je to hledaný otec. Pokud ne, dostali jsme se do pravého sourozence, takže přejdeme ještě jednou po Q.

Pokud by vrchol měl jen jediného syna, bude na něj ukazovat P a jeho Q se odkáže zpět na otce. Porovnáním hodnot zjistíme, zda je to levý či pravý syn.

Martin „Medvěd“ Mareš & Dominik Macháček

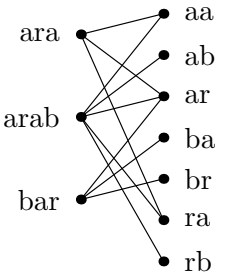
26-2-6 Zkratky míst

Všimněme si, že slova s různým počátečním písmenem nikdy nemůžou mít stejnou zkratku. Zkratka je totiž určená prvním písmenem slova a pak nějakým výběrem dalších dvou písmen z něj.

Můžeme tedy úlohu řešit samostatně pro každé počáteční písmeno a úloha se nám tak pro jednu „hromádku“ se stejným počátečním písmenem zjednodušuje na nalezení unikátních dvoupísmenných zkratek sestávajících se ze zbylých písmen slova (mimo prvního) ve správném pořadí.

Pokud jste již někdy slyšeli o problému hledání maximálního párování v bipartitním grafu, určitě vám jej tato úloha trochu připomíná. Pokud ne, doporučujeme vaší pozornosti příslušný článek v naší zbrusu nové programátorské encyklopedii.[44]

Jednu partitu (říkejme jí levá) tvoří slova, druhou (pravou) všechny možné zkratky. Hrany vedou z každého slova do všech zkratek, které z něj lze utvořit. Pak párování v tomto grafu odpovídá nějakému přiřazení zkratek slovům. Nyní stačí prostě najít párování maximální, tedy takové, které co nejvíce slovům přiřadí zkratku. Pro množinu slov `{ara, arab, bar}` je graf ukázán na obrázku vpravo.



[44] `http://ksp.mff.cuni.cz/encyklopedie/parovani.html`

Mohlo by vás ještě trochu překvapit, že si na úlohu bereme takto obecné kladivo. Naše zkratkové grafy přece mají velice speciální tvar, fakt, kterého obecný párovací algoritmus nijak nevyužije. To se občas hodí, když se náš problém podobá nějakému známému, na chvíli se tvářit, že jsou stejné. Nebojte, za chvíli se k alespoň některým zvláštnostem zkratkového grafu zase vrátíme.

Označme si N počet vstupních slov, S součet jejich délek a $|\Sigma|$ velikost abecedy. Než se pustíme do rozboru složitosti, ještě přidáme do algoritmu malý zlepšovák: při načítání vstupu upravíme slova tak, že v každém zachováme pouze první a poslední výskyt libovolného písmene. Snadno si rozmyslíte, že tím nijak nezměníme množinu zkratek, které lze ze slova vytvořit. V každém takto upraveném slově se libovolné písmeno abecedy vyskytuje nejvýše dvakrát, je tedy dlouhé $\mathcal{O}(|\Sigma|)$, a tudíž $S = \mathcal{O}(N \cdot |\Sigma|)$.

Jak bude náš graf velký? Možných zkratek existuje $|\Sigma|^2$, počet vrcholů tedy bude $N + |\Sigma|^2$. V nejhorším případě, kdy každé slovo obsahuje všechna písmena abecedy a půjde z něj vytvořit řádově $|\Sigma|^2$ zkratek, bude graf obsahovat $N|\Sigma|^2$ hran.

Pokud je abeceda malá, můžeme $|\Sigma|$ prohlásit za konstantu a dostaneme příjemný lineárně velký graf. Graf sestavíme v čase $\mathcal{O}(N \cdot |\Sigma|^2)$ (díky tomu, že každé slovo je nyní dlouhé $\mathcal{O}(|\Sigma|)$), tedy pro malou abecedu $\mathcal{O}(N)$.

Budeme-li, jako v odkazovaném článku, párovat postupným hledáním zlepšujících cest, které běží v čase $\mathcal{O}(|M| \cdot |E|)$ (kde $|E|$ je počet hran grafu a $|M|$ velikost výsledného párování), potrvá to $\mathcal{O}(N^2)$. Paměti spotřebujeme $\mathcal{O}(N)$ na uložení grafu.

Kdož jste zvyklí převádět párování na toky v sítích, vězte, že nejde o nic jiného než jinou formulaci Ford-Fulkersonova algoritmu pro párovací sítě. My zde používáme tuto verzi, protože další úpravy a optimalizace se na ní budou popisovat snáz než v tokové formulaci.

Program (Python): `http://ksp.mff.cuni.cz/viz/26-2-6-male-abc.py`

Toto řešení je dostačující jak pro většinu běžných použití (anglická abeceda, ASCII), tak pro získání plného počtu bodů. Prostou výměnou párovacího algoritmu za Hopcroft-Karpův[45] zlepšíme čas na $\mathcal{O}(|E|\sqrt{|M|}) = \mathcal{O}(N\sqrt{N})$.

Zvláště zvědaví mohou pokračovat ve čtení a dozvědět se, co dělat v případě, že máme abecedu opravdu velkou.

Velké abecedy

Pokud bychom předchozí postup chtěli zkusit např. s Unicode, náš graf by měl řádově 2^{32} vrcholů, do většiny z kterých by žádná hrana nevedla. Nabízí se vytvářet vrcholy pouze pro zkratky, které mohou z nějakého slova vzniknout. Kolik jich bude? Ze slova délky L lze vytvořit až řádově L^2 zkratek (pokud jsou

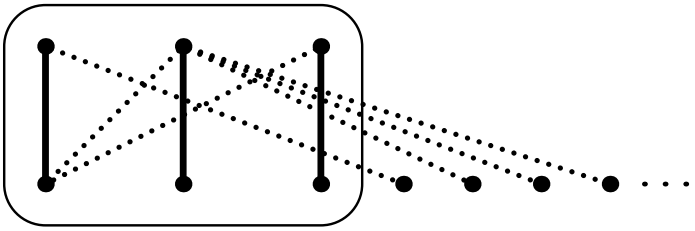
[45] `http://ksp.mff.cuni.cz/encyklopedie/hopcroft-karp.html`

všechna jeho písmena různá). Tedy náš graf může obsahovat až $\mathcal{O}(S^2)$ vrcholů a $\mathcal{O}(NS^2)$ hran.

To je pořád docela dost, jak pro časovou náročnost ($\mathcal{O}(N^2S^2)$), tak pro paměťovou náročnost ($\mathcal{O}(NS^2)$) našeho algoritmu. Obojí zkusíme zachránit drobnými úpravami párovacího algoritmu.

Pokud jej neznáte, teď je vhodná chvíle to napravit. Pokud ano, pro jistotu zopakujeme drobné shrnutí a trochu terminologie: *(ne)párovací hrana* je hrana grafu (ne)patřící do aktuálního párování, *volný vrchol* je takový, který není spárován (všechny hrany s ním incidentní jsou nepárovací), *střídavá cesta* je cesta v grafu, na které se střídají párovací a nepárovací hrany, a nakonec *zlepšující cesta*[46] (též *volná střídavá cesta*) je střídavá cesta začínající a končící volným vrcholem (a tedy nutně i nepárovací hranou). Algoritmus začne s prázdným párováním a v každém kroku najde zlepšující cestu, změní po celé její délce párovací hrany na nepárovací a naopak, čímž zachová korektnost párování a zvětší jej o jedničku. Pokud zlepšující cesta neexistuje, párování je maximální.

Klíčovým pro nás bude jedno pozorování: po celou dobu běhu algoritmu je většina grafu „nezajímavá“, totiž tvořená volnými zkratkami. „Zajímavých“ vrcholů je jen $\mathcal{O}(N)$ (N slov a nejvýše N spárovaných zkratek). Tedy v zajímavé části grafu je nejvýše $\mathcal{O}(N^2)$ hran. Graf si lze představit takto:



Nejprve napravíme čas. Všimneme si, že každá zlepšující cesta má lichou délku, a tudíž spojuje vrcholy z opačných partit. Stačí nám tedy hledat zlepšující cesty pouze z vrcholů v levé partitě. Dále si uvědomíme, že skoro celé hledání zlepšující cesty probíhá v zajímavé části grafu. Jakmile se dostaneme do nezajímavé části, narazili jsme na volný vrchol, a tedy konec zlepšující cesty. Při libovolném hledání zlepšující cesty tedy navštívíme libovolné množství vrcholů ze zajímavé části grafu, ale nejvýše jeden z nezajímavé. Takové hledání tedy bude trvat $\mathcal{O}(N^2)$. A použijeme-li odhad složitosti $\mathcal{O}(\text{čas na nalezení zlepšující cesty} \cdot |M|)$, dostáváme čas $\mathcal{O}(N^3)$.

[46] Příznivci tokového párování: rozmyslete si, že takováto definice zlepšující cesty přesně odpovídá tokové zlepšující cestě v párovací síti – jen s useknutými hranami vedoucími ke zdroji a stoku (neb v našem grafu žádný zdroj a stok nemáme).

Nyní co s pamětí? Pomoci by nám mohl obvyklý trik na úlohy, které řešíme sestavením vhodného grafu: nebudeme jej v paměti vytvářet celý najednou. Místo toho, kdykoli se bude párovací algoritmus chtít podívat na nějakou část našeho grafu, tak mu ji „na požádání“ sestavíme. Co musíme umět s naším „grafem“ provádět, aby párovací algoritmus fungoval?

1. Pamatovat si, která hrana je aktuálně v párování a která ne, umět do párování hranu přidat/odebrat.
2. Umět vyjmenovat sousedy daného vrcholu.
3. Umět o vrcholu poznat, zda je volný.

KSP

Stačí nám si mimo samotného grafu pamatovat nějaké jiné datové struktury, které budou umět na takovéto dotazy rychle odpovídat. Pak prostě párovací algoritmus upravíme tak, aby kdekoli původně přistupoval přímo k paměťové reprezentaci grafu, místo toho použil tyto naše struktury.

Najít sousedy daného slova je jednoduché: prostě vyzkoušíme všechny možnosti volby prvního písmene a pro každou z nich všechny možnosti volby druhého písmene, dostávajíce všechny možné zkratky v čase $\mathcal{O}(1)$ na každou. Ovšem obráceně (najít sousedy zkratky, tedy všechna slova, ze kterých ji lze utvořit) to vůbec není jednoduché.

řešení

Naštěstí si všimneme, že to vůbec nepotřebujeme. Hledáme pouze volné střídavé cesty, a pokud začneme zleva, půjdeme na takové cestě doprava vždy po nepárovací hraně a doleva vždy po párovací. Stačí nám tedy umět hledat:

- Pro vrcholy z levé (slovní) partity sousedy po nepárovacích hranách.
- Pro vrcholy z pravé (zkratkové) partity souseda po párovací hraně, pokud existuje.

A to už zvládneme snadno. Naše reprezentace bude vypadat následovně:

- Vrcholy z levé partity (slova) si očíslujeme, vrcholy z pravé budeme prostě označovat dvěma písmeny.
- Párování si budeme uchovávat jako dvojici slovníků (hešovacích tabulek[47] či vyhledávacích stromů)[48] mapujících slova na aktuálně přiřazené zkratky a naopak. To nám umožní rychle hledat párovací hrany „z obou stran“.
- Volný vrchol poznáme prostě tak, že v příslušném slovníku není.
- Sousedy slova vyjmenujeme triviálně, jak bylo popsáno výše, z již předem redukovaných slov (zachován jen první a poslední výskyt libovolného písmene). Některé zkratky takto můžeme vygenerovat vícekrát – ale všimněme si, že každou nejvýš čtyřikrát (např. `ab` ve slově `aabb`), navíc díky značkování vrcholů nás každé toto opakování stojí jen konstantní čas. Opakování se dá

[47] `http://ksp.mff.cuni.cz/viz/kucharky/hesovani`

[48] Viz kuchařku druhé série.

zbavit úplně, pokud si k redukovanému slovu zapamatujeme ještě něco navíc (snadné cvičení).

- Spárovaného souseda dané zkratky nalezneme prostým přečtením příslušné hodnoty ze slovníku.

Prostřednictvím těchto informací již dokážeme „simulovat“ náš graf s logaritmickým či průměrně konstantním zpomalením. Vzhledem k tomu, že žádné párování není větší než N, spotřebujeme $\mathcal{O}(N)$ paměti na pomocné slovníky, ale nesmíme zapomenout na $\mathcal{O}(S)$ pro vstup, tedy celkem $\mathcal{O}(S)$. Výsledný algoritmus poběží v čase $\mathcal{O}(S + N^3 \log N)$.

Hopcroft-Karp a velké abecedy

I pro velké abecedy bychom rádi použili Hopcroft-Karpův algoritmus. Ovšem jen pokud na něm dokážeme provést optimalizace obdobné těm výše – v původní podobě by byl pomalejší než předchozí algoritmus.

Hopcroft-Karp se skládá z *fází*: v každé najde v inkluzi maximální množinu disjunktních nejkratších zlepšujících cest a všechny použije ke zvětšení párování. Provádí to tak, že si nejdřív pomocí BFS rozdělí vrcholy grafu do vrstev podle délky nejkratší střídavé cesty vedoucí do nich z nějakého volného vrcholu levé partity a pak už jen pomocí DFS vysbírává nejkratší cesty a značkuje vrcholy, aby žádný nepoužil dvakrát. Obě prohledávání začínají ve volných vrcholech levé partity.

Kromě grafu samotného, na který si vystačíme se strukturami popsanými dříve, si navíc potřebujeme pamatovat ještě:

- BFS ohodnocení (rozdělení do vrstev)
- DFS značky (označující již navštívené vrcholy, abychom se do nich nevraceli)

Obojí budeme uchovávat ve slovnících a s každou částí se vypořádáme trochu jinak.

BFS začínáme v zajímavé části grafu. Jakmile poprvé navštívíme nezajímavý vrchol, prohledávání ukončíme. Tak určitě doběhne v $\mathcal{O}(N^2)$, jen některým vrcholům v ℓ-té (kde ℓ je délka nejkratší zlepšující cesty) vrstvě bude chybět ohodnocení (díky čemuž si též vystačíme s $\mathcal{O}(N)$ paměti). To ale vůbec nevadí. Prostě jen při DFS budeme ochotni navštěvovat neohodnocené sousedy vrcholů v $(\ell - 1)$-ní vrstvě a chovat se k nim, jako by ležely v ℓ-té vrstvě.

Nyní k DFS: Každá navštívená volná zkratka znamená novou nalezenou zlepšující cestu. Těch ovšem najdeme za jednu fázi nejvýše N, navštívíme tedy jen $\mathcal{O}(N)$ nezajímavých vrcholů a hran. To opět znamená, že si vystačíme s $\mathcal{O}(N^2)$ času na DFS část jedné fáze a $\mathcal{O}(N)$ paměti pro značky.

Hopcroft-Karpův algoritmus běží v čase $\mathcal{O}(T_f \sqrt{|M|})$, kde T_f je čas strávený jednou fází a $|M|$ je velikost výsledného párování. V předchozích odstavcích jsme ukázali, že $T_f = \mathcal{O}(N^2 \log N)$ (použijeme-li jako slovník vyvážené vyhledávací stromy), již dávno víme, že $|M| \leq N$ a ještě potřebujeme $\mathcal{O}(S)$ na načtení vstupu.

Výsledná složitost tedy bude $\mathcal{O}(S + N^2\sqrt{N}\log N)$ a stále si vystačíme s $\mathcal{O}(S)$ paměti. V praxi by nejspíš bylo výhodnější reprezentovat slovníky jako hešovací tabulky.

Program (Python): `http://ksp.mff.cuni.cz/viz/26-2-6-velke-abc.py`

Filip Štědronský

26-2-7 Čištění kmene

KSP

řešení

Nejdříve se podíváme na to, jakým způsobem může vypadat výsledný pohyb humanoidů. Nahlédneme, že existuje optimální řešení splňující tyto vlastnosti:

- Každý humanoid vyčistí nějaký souvislý interval.
- Každý humanoid změní směr pouze jednou. Tedy napřed půjde jen doprava a pak jen doleva, nebo naopak. Tedy dojde na jeden kraj svého intervalu, otočí se a zamíří ke druhému kraji, kde se zastaví.
- Z toho je hned vidět, že se nevyplatí humanoidy cíleně prohazovat. Pokud by se dva humanoidi měli prohodit, tak si můžeme situaci představit tak, že se od sebe odrazí a každý pokračuje plánovanou cestou toho druhého.

Zkusme nyní zodpovědět otázku, zda je možné kmen vyčistit, pokud budeme mít k dispozici k kroků. Postupovat budeme následovně: Pokud má nejlevější humanoid nějakou nečistotu nalevo od sebe, tak ji určitě musí vyčistit on. Tedy začátek jeho intervalu bude l kroků nalevo od něj – tam kde daná nečistota leží. Pravý konec jeho intervalu bude r kroků napravo od něj, přičemž chceme, aby r bylo co největší.

Humanoid má dvě možnosti:

- Buď půjde nejprve l kroků doleva, tam se otočí a vydá se $l+r$ kroků doprava.
- Nebo půjde nejdříve r kroků doprava, obrátí se a pak se vydá $l + r$ kroků doleva.

Celkově má udělat k kroků, takže bude platit buď $2l+r = k$, nebo $l+2r = k$. Z těchto dvou rovnic vybereme tu, jejíž řešení má větší r. Pokud ani jedna rovnice nemá řešení s $r \geq 0$, tak nejlevější nečistotu za k kroků vyčistit nemůžeme, tedy pro tuto hodnotu k řešení neexistuje.

Pokud alespoň jedna z rovnic řešení pro r má, tak smažeme všechny nečistoty do l kroků nalevo od humanoida a do r kroků napravo od humanoida a stejným způsobem pokračujeme výpočtem u dalšího humanoida zleva. Pokud po zpracování posledního humanoida budeme mít vyčištěné všechny nečistoty, tak jsme našli řešení, které funguje pro k kroků.

Nyní si stačí všimnout, že pokud úloha má řešení pro k kroků, tak určitě má řešení i pro $k + 1$ kroků. Naopak pokud úloha nemá řešení pro k kroků, tak

určitě nemá řešení ani pro $k-1$ kroků. Optimální k tedy můžeme nalézt binárním vyhledáváním.

Maximální počet kroků, který má smysl uvažovat, je dvakrát rozdíl nejmenší a největší souřadnice na vstupu, protože určitě nepůjdeme víc jak tam a zpátky. Označme tuto hodnotu X.

Pak binární vyhledávání udělá maximálně $\mathcal{O}(\log X)$ kroků; každý zabere čas $\mathcal{O}(H+M)$, kde H je počet humanoidů a M je počet nečistot. Poznamenejme ještě, že humanoidy i místa si na začátku musíme setřídit. Celková časová složitost tedy bude $\mathcal{O}((H+M)\log X)$.

Paměťová složitost je $\mathcal{O}(M+H)$. Můžete nahlédnout do vzorového programu.

Program (C++): `http://ksp.mff.cuni.cz/viz/26-2-7.cpp`

Karel Tesař

26-2-8 Továrna na přepisování

Druhou série sice neřešilo tak moc lidí, jako sérii první, ale i tak jsem byl potěšen tím, kolik vašich řešení přišlo.

Úkol 1

Na řešení první úlohy bylo možné jít několika směry. Ukážeme si zde přepisovací program, který postupně umazává stejná čísla a zvedá první z nich. Lehce nahlédneme, že nahrazením dvou stejných čísel vedle jedním (stejným) číslem nic nepokazíme. Neklesající posloupnost zůstane neklesající a klesající také nepřestane klesat.

Pak nám už stačí jen zvedat číslo na první pozici postupně až do devítky. Po každém zvýšení nám tak potenciálně vznikne nová dvojice stejných znaků na začátku, které srazíme na jeden. Pokud bude posloupnost neklesající, tak nám tento postup v průběhu vymaže všechna čísla až na poslední. Pokud se tak stane, zahlásíme úspěch přepsáním na `ANO`.

Pokud je však posloupnost někde klesající, tak první číslo zvýšíme až na devítku a ještě nám za ní něco zbude. V takovém případě se jen zbavíme zbytku vstupu a na výstupu zanecháme `NE`.

```
00 → 0
 ⋮
99 → 9
^0 → 1
 ⋮
^8 → 9
^9$ → ANO
^9 → X
X0 → X
 ⋮
X9 → X
 X → NE
```

Takový přepisovací program je uveden v pravém sloupci. V každém kroku (až na konstantně mnoho zvýšení prvního čísla) jedno číslo umaže, tedy běží v lineárním čase k velikosti vstupu.

Alternativním postupem může být naopak detekovat chyby. Neboli měli bychom pravidla pro jakoukoliv dvojici sousedních čísel ve špatném uspořádání

(takových je 45) a tuto dvojici bychom přepsali na nějaký chybový znak. Potom bychom vymazali všechna čísla a pokud by nám na konci zůstal nějaký chybový znak, vypsali bychom `NE`. Tento postup má také lineární složitost, ale potřebuje o něco více přepisovacích pravidel.

Úkol 2

Budeme triviálně realizovat binární sčítačku. Budeme si ji držet vlevo od hvězdiček tak, aby hvězdičky přiléhaly k nejnižšímu bitu čísla. To nám umožní za každou hvězdičku k tomuto bitu přičíst +1.

KSP

Jak ale zajistit přenosy do vyšších řádů? Jednoduše, v momentě, kdy bude potřeba provést přenos, zapíšeme namísto přenosu nějaký symbol (z logiky sčítání se nabízí použít třeba symbol 2). A poté ho pomocí dalších pravidel posuneme do vyššího řádu (tedy z $1011 + 1$ se stane 1012 a po přenosu nejprve 1020 a pak 1100).

Realizace s pomocí pravidel (a s ošetřením úvodní inicializace počítadla a prázdného vstupu) vypadá následovně:

řešení

`^$` → `0`
`^*` → `0*`
`0*` → `1`
`1*` → `2`
`^2` → `10`
`02` → `10`
`12` → `20`

Kolik kroků provedeme? Odstranění hvězdičky provedeme právě tolikrát, kolikrát byla na vstupu. Mohlo by se však zdát, že nám složitost pokazí přenosy. Stačí si ovšem uvědomit, že přenos z posledního bitu provedeme v každém druhém kroku, přenos z předposledního bitu jen v každém čtvrtém kroku a tak dále. Když posčítáme všechny přenosy, vyjde nám, že jich provedeme maximálně $2N$, tedy složitost je stále lineární k délce vstupu. Pro podrobnější důkaz amortizace časové složitosti binární sčítačky nahlédněte na konec řešení minulého dílu seriálu.

Skládání programů

Pro řešení třetího úkolu si nejdříve ukážeme, jak za sebe složit libovolné dva přepisovací programy. Mějme dvě sady přepisovacích pravidel, z kterých chceme vyrobit třetí sadu tak, že nám bude vracet stejný výsledek, jako kdybychom výstup prvního přepisovacího programu použili jako vstup pro druhý.

Co se stane, pokud oba programy jen napíšeme pod sebe? Jelikož aplikace pravidel probíhá postupně, tak dokud běží první přepisovací program, nemůže se provést žádné pravidlo z druhého přepisovacího programu. Problém ale nastane ve chvíli, kdy se výpočet přehoupne do druhého programu. Jak zajistit, aby se nepoužilo žádné pravidlo z prvního programu?

Kdyby programy používaly úplně jiné sady znaků, bylo by to jednoduché (metaznaky začátku a konce řetězce můžeme také odlišit a to třeba tak, že k nim vždy přilepíme ještě nějaký další speciální znak). Tak si to pojďme zařídit.

Všechna pravidla prvního programu přeložíme tak, aby používaly nějaké jiné znaky (třeba ty samé, ale s čárkou). Poté jen potřebujeme přidat překlad znaků před vstupem do druhého programu, který je přepíše zpět na původní znaky.

Druhou nutnou věcí je přepis vstupní abecedy na čárkovanou verzi před vstupem do prvního programu (ale tak, aby se přepis nemohl opakovat po doběhnutí druhého programu). K tomu nám stačí libovolný jeden znak na vstupu, který se nevyskytne v průběhu výpočtu druhého programu, ani na jeho výstupu. Pomocí něj provedeme úvodní přepis.

V praxi budeme chtít mít všechna překladová pravidla na začátku přepisovacího programu. Spouštět je budeme jedním přepisovacím pravidlem na správném místě programu, které nám přidá do řetězce speciální přepisovací znak (rozmyslete si, proč to děláme takto a proč nemůžeme všechna pravidla přesunout na místo spouštěcího pravidla).

Výsledné spojení programů pak vypadá takto:

- Pravidla pro převod abecedy $A \to A'$ využívající nějaký speciální znak ze vstupu
- Pravidla pro převod abecedy $A' \to A$ využívající znak α
- První program (s převedenou abecedou)
- Pravidlo přidávající α (zajistí překlad abecedy)
- Druhý program

Umíme tedy spojit libovolné dva programy do jednoho, pokud se na vstupu prvního vyskytuje alespoň jeden znak, který v druhém programu použitý není.

Úkol 3

Poznatek o spojování programů použijeme pro řešení třetí úlohy. Idea bude taková, že si vstup nafoukneme o zkopírované verze obou čísel. První číslo opíšeme normálně, druhé číslo pozadu (aby se nejnižší bity obou čísel dostaly k sobě).

Taky předpokládejme to, že čísla nejsou prefixována nulami. Pokud by byla, můžeme přebývající nuly snadno odmazat. Stačí si zavést dvě pomocná pravidla pro levé a pro pravé číslo.

Tedy ze vstupu ve tvaru „`1010#1100`“ vyrobíme dočasnou verzi přepisem „`1010_1010=0011_1100`“. S touto verzí pak provedeme porovnání (postupným odmazáváním nejmenších bitů) a zjistíme, jestli je větší pravé nebo levé číslo. Druhé pak vymažeme a jsme hotovi.

Nejprve se zabývejme **rozkopírováním**. Ukážeme pravidla pro kopírování pravé části pozpátku. Druhá sada pravidel pro kopírování levé části bude podobná.

Hlavní idea je, že si vyrobíme na koncích jakési zarážky a postupně budeme znak po znaku pomocí *vozíku* (znaku `v`) převážet zkopírované znaky na správné místo. Následující sada pravidel nám přepis odstartuje:

$$\texttt{\#} \to \texttt{y\_=\_Y}$$
$$\texttt{Y0} \to \texttt{0Y}$$
$$\texttt{Y1} \to \texttt{1Y}$$
$$\texttt{Y\$} \to \texttt{X}$$

Tím nám vznikne řetězec (zajímejme se teď jen o pravou stranu od `=`) ve tvaru `_Y...X`. Teď můžeme začít převážet znaky. Vždy nabereme jedno číslo u koncové značky, posuneme koncovou značku o jedna doleva a pomocí *vozíku* převezeme

číslo těsně za `_`. Ve chvíli, kdy nemáme žádný vozík, si ho pořídíme a ve chvíli, kdy pravý ukazatel dojde až k `_`, ukazatel zahodíme – viz program vpravo.

```
0v0 → v00
1v0 → v01
0v1 → v10
1v1 → v11
_v0 → 0_
_v1 → 1_
 1X → v1X1
 0X → v0X0
 _X → _
```

Tím jsme si vytvořili přepisovací program, který nám předzpracuje vstup. Nyní si vytvořme program, který nám zajistí samotné **porovnání čísel**.

Budeme postupně odebírat nejméně významné bity z obou čísel a porovnávat je. Aktuální stav porovnání si budeme držet mezi oběma čísly – jako `=`/`>`/`<` podle toho, jestli jsou čísla zatím stejná, je větší levé, nebo je větší pravé.

Je jasné, že pokud budou na obou stranách stejná čísla, stav (ať bude jakýkoliv) se nám nezmění. Jinak se nahne na tu stranu, kde je aktuálně větší číslo (významnější bit převáží nad těmi méně významnými).

Pro stejně dlouhá čísla tedy máme pravidla níže vlevo.

```
1=1 → =
0=0 → =
1=0 → >
0=1 → <
1>1 → >
0>0 → >
1>0 → >
0>1 → <
1<1 → <
0<0 → <
1<0 → >
0<1 → <
```

```
1>_ → >_
0>_ → >_
1<_ → >_
0<_ → >_
0=_ → >_
1=_ → >_
```

```
_>_ → Q
 Q1 → Q
 Q0 → Q
 Q$ →
```

Samostatně vyřešíme případ, že je jedno z čísel kratší jak druhé – pak je to delší určitě větší (ukážeme pravidla pro `>`, pro `<` jsou analogická) – viz pravidla výše uprostřed.

Uprostřed nám zůstal jeden znak udávající, které z čísel je větší. Nyní stačí to menší vymazat (opět ukážeme jen pro `>`) – viz pravidla výše vpravo.

Spojením kopírovacího a porovnávacího programu (spojení můžeme provést, protože vstup obsahuje znak `#`, který druhý program nepoužívá) vznikne program řešící celou úlohu. Jelikož při přesouvání provedeme pro každý znak lineárně mnoho kroků vzhledem k délce vstupu a přesunů je lineárně mnoho, je výsledná složitost $\mathcal{O}(N^2)$, kde N je počet znaků na vstupu.

Úkol 4

Jelikož jsme měli Turingovy stroje v minulé sérii zadané tak, že vracely pouze výsledek ANO nebo NE (stavem, ve kterém skončily), tak se na tento výstup omezíme i u přepisovacích pravidel – na výstupu tedy zanecháme buď řetězec `ANO` nebo řetězec `NE`.

V případě, že by se řetězce ANO nebo NE vyskytovaly na vstupu nebo někde v průběhu výpočtu, odlišíme ty koncové tím, že pro ně použijeme speciální symboly.

Když jsme si odbyli technické detaily, pojďme se podívat na hlavní myšlenku převodu Turingova stroje na přepisovací pravidla. Následující krok Turingova stroje je vždy určený pozicí hlavy, stavem, ve kterém se stroj nachází, a aktuálním obsahem pásky. Tuto informaci budeme potřebovat nějak kódovat v řetězci.

Pozici hlavy tedy budeme reprezentovat speciálním znakem někde v řetězci. Přesněji více speciálními znaky – pro každý stav stroje jeden znak. Zaveďme si pozici hlavy tak, že tento speciální znak bude stát před znakem, na který by ukazovala hlava v Turingově stroji.

Každé pravidlo Turingova stroje má tyto části: vstupní stav, vstupní znak, výstupní stav, výstupní znak a výstupní posunutí. To ale můžeme velmi lehce přepsat do řeči přepisovacích pravidel. Například pravidlo pro stroj říkající „Pokud jsi ve stavu S a na pásce je x, přejdi do stavu T, zapiš na pásku y a posuň se doprava“ bychom mohli napsat jako: `Sx` $\to$ `yT`.

Drobným problémem je posouvání doleva (vzhledem k tomu, že náš znak pro hlavu stojí před přepisovaným znakem). Pro posun hlavy doleva si musíme vyrobit jedno pravidlo pro každý znak abecedy. Když bude `#` zastupovat libovolný znak abecedy, vypadá pak stejné pravidlo, jen s posunem hlavy doleva, takto: `#Sx` $\to$ `T#y`.

Tím máme hlavní část převodu hotovou. Stačí vyřešit pouze tři drobné technické detaily:

- Turingův stroj počítá s tím, že má nekonečnou pásku plnou mezer, kdežto náš řetězec je omezený. To však vyřešíme jednoduše tím, že pokud se hlava dostane na začátek nebo na konec řetězce, tak přidáme mezeru. Pro všechny stavy `S` budeme tedy mít následující pravidla (tato pravidla musíme v programu umístit nad všechna ostatní):
- Druhým detailem je to, že musíme řetězec na konci výpočtu smazat a zanechat zde jen ANO nebo NE. To však lehce zařídíme nějakými mazacími pravidly.

  `^S` $\to$ `␣S`
  `S$` $\to$ `S␣`
- Poslední drobnost je, jak celý výpočet odstartovat, tedy jak na začátku umístit do řetězce symbol hlavy? Kdybychom měli jen pravidlo `^` $\to$ `S`, tak by nám (minimálně po skončení výstupu) program začal běžet znovu.

  Zabráníme tomu tak, že si zajistíme, že tato náhrada proběhne jen a pouze na začátku programu. Nejprve potřebujeme, aby se ani `ANO` ani `NE` nevyskytovaly na vstupu, ale to jsme si už ošetřili výše. Zkonstruujeme si tedy následující pravidla pro všechny znaky vstupní abecedy (kde `#` zastupuje libovolný znak abecedy na vstupu, `S` je počáteční stav a `@` je nějaký speciální

znak, který není jinde použitý):

`^#` → `@S#`
`@ANO` → `ANO`
`@NE` → `NE`

První pravidlo nám zajistí to, že k přidání počáteční pozice hlavy už nikdy znovu nedojde (protože neexistuje přidávací pravidlo s `^@` na levé straně), a druhé a třetí pravidlo nám odstraní tento speciální znak z výstupu. Také je nutné upravit pravidlo na přidávání mezer na začátek řetězce, aby zachovávalo pozici `@`, ale to je už maličkost. Převod je tedy tímto hotov.

KSP

Úkol 4 – dodělávky

Zbývá zamyslet se nad složitostí. Na každou aplikaci pravidla Turingova stroje provedeme právě jednu aplikaci přepisovacího pravidla, takže by se mohlo zdát, že složitost obou bude stejná. Přepisovací programy ovšem musí na konci výpočtu smazat celý řetězec, takže pokaždé běží alespoň lineárně s délkou vstupu (kdežto Turingův stroj může skončit třeba po první aplikaci pravidla přechodem do koncového stavu).

Můžeme tedy prohlásit, že přepisovací program má stejnou složitost jako Turingův stroj, vždy ale nejméně lineární s velikostí vstupu.

řešení

Pokud bychom měli převod demonstrovat na příkladu s vyváženou posloupností z prvního dílu seriálu, tvořil by hlavní část programu pouze mechanický přepis pravidel Turingova stroje na pravidla přepisovacího programu. To jistě každý zvládne sám.

Mimo nich budeme potřebovat pravidla, která nám na začátku vloží do řetězce hlavu, a pravidla, která nám na konci vymažou zbylé znaky.

A to je milí přátelé z druhé série vše. Přeji vám hodně zdaru i v dalších dílech seriálu při potýkání se s dalšími zajímavými výpočetními modely.

Jirka Setnička

26-3-1 Výklad z drahokamů

Protože věštění, byť z drahokamů, je komplikovaná věc, začneme jednodušší úlohou. Budeme pro začátek hledat co největší žlutý čtverec.

Podobné úlohy se vyskytují docela často a zkušenému řešiteli už něco našeptává „dynamika“.

Každý (i největší) jednobarevný čtverec má nějaký pravý dolní roh. My si tedy pro každou pozici spočítáme největší čtverec, který od něj vede doleva nahoru a potom jen vybereme nejlepší možnost.

Pokud je aktuální pozice červená, je zřejmě výsledek 0, neboť je to největší žlutý čtverec, který se zde nachází. Naopak, pokud je žlutá, podíváme se na pozici o jedna doleva, o jedna nahoru a o jedna šikmo doleva nahoru. Z nich vybereme minimum jejich čtverců a to zvětšíme o 1 – na čem se „zarazí“ čtverec odpovídající tomuto minimu, na tom se zarazí i náš čtverec v aktuální pozici. Naopak, tím jak se čtverce sousedních políček překrývají, tak nám dají k dispozici odpovídající žlutou plochu, kterou jen aktuální pozicí rozšíříme. Zkuste si to nakreslit.

Samozřejmě, maximální čtverce musíme počítat v takovém pořadí, abychom všechny tři sousedy, do kterých nahlížíme, měli již spočítané. Například po řádcích zleva doprava, stejně jako se čte.

A jako malý technický trik, abychom nemuseli řešit vykukování z pole na levém a horním okraji, připravíme si řádek a sloupeček nul jako jakýsi rámeček.

Nyní tedy, jak na naši těžší úlohu? Všimneme si, že každý šachovnicový čtverec je podčtvercem nějaké velké šachovnice takových rozměrů, jaké má celá mřížka. A tyto velké šachovnice existují právě dvě – jedna, která má vlevo nahoře žlutý drahokam, a k ní inverzní, která má vlevo nahoře červený. Celou mřížku si tedy převedeme a budeme pokládat žluté drahokamy tam, kde barva na dané pozici odpovídá první šachovnici, a červené, kde druhé. Tím jsme úlohu převedli na nalezení největšího jednobarevného čtverce – což je buď žlutý, nebo červený. Žlutý už najít umíme a nalezení červeného bude ponecháno na čtenáři.

Jak paměťová, tak časová složitost jsou lineární s velikostí vstupu. Ke každému políčku vstupu si pamatujeme konstantně mnoho informací (jeho převedenou barvu a jeho maximální čtverec doleva nahoru). Obdobně, při převodu políčka uděláme konstantně mnoho práce, a při počítání minima také koukáme jen na tři okolní políčka.

Program (Python): `http://ksp.mff.cuni.cz/viz/26-3-1.py`

Michal „vorner“ Vaner

26-3-2 Střih látky

Označme si vrcholy mnohoúhelníka V_0 až V_{n-1} a střihovou polopřímku p (zadanou počátkem P a vektorem u). Dovolme si z úsporných důvodů zanedbat „ošklivé“ případy, kdy se polopřímka mnohoúhelníka dotýká v jednom bodě či splývá s některou jeho stranou. Nebude těžké ošetřit je dodatečně přidáním několika podmínek na vhodná místa.

KSP

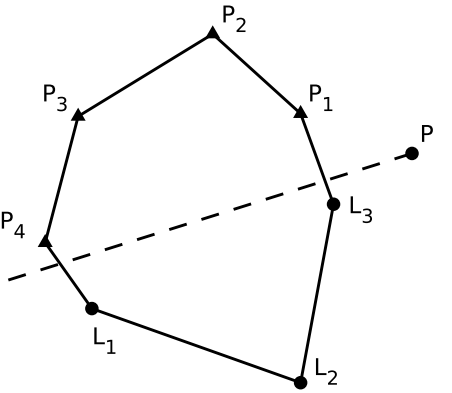
Víceméně z definice konvexity plyne, že p bude mít s mnohoúhelníkem nejvýše dva průsečíky.

Všimněme si, že doopravdy vlastně chceme zjistit, se kterými stranami mnohoúhelníka se protíná. Dopočítat konkrétní souřadnice průsečíků už je pak jen drobné cvičení ze středoškolské geometrie. Pokud si s takovými úlohami nerozumíte, nastala správná chvíle přečíst si geometrickou kuchařku.[49]

V průběhu řešení budeme často chtít vědět, jestli bod od nějaké (polo)přímky leží napravo, nebo nalevo (při pohledu po směru polopřímky). Prozatím nám uvěřte, že to snadno zjistíme v konstantním čase.

řešení

Všimněme si, že nějaká hrana mnohoúhelníka kříží p právě tehdy, když jeden její konec je vlevo od p a druhý vpravo. Dále si všimněme, že vrcholy nalevo a napravo od p tvoří v mnohoúhelníku souvislý úsek (díky konvexitě). My tedy vlastně v posloupnosti vrcholů nehledáme nic než hranici mezi levými a pravými vrcholy (díky cykličnosti existují dvě). To svádí k tomu použít něco jako binární vyhledávání.



Zde nám ovšem komplikuje život fakt, že posloupnost je cyklická a záleží na tom, jaký vrchol vezmeme za počáteční. Uvažme např. posloupnosti `PLLPPPP` a `PPPPLLP`. Pouhým pohledem na prostřední prvek (v obou případech `P`) nepoznáme, ve které polovině máme pokračovat v hledání.

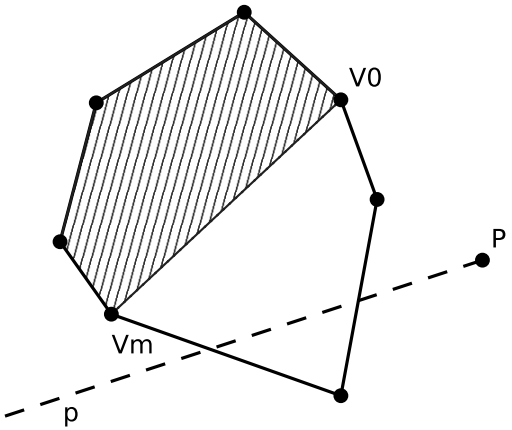
[49] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`

Pokud bychom znali alespoň jeden vrchol nalevo od přímky a jeden napravo, je to jednoduché: v těchto dvou vrcholech posloupnost rozstřihneme. Tím vzniknou dvě posloupnosti tvaru `LLLPPP` či `PPPLLL`, v obou z kterýchž najdeme hranici prachobyčejným binárním vyhledáváním.

Jak takové dva protilehlé vrcholy sehnat? Uvažme body V_0 a $V_m := V_{n/2}$. Mohou nastat dva případy:

a) V_0 a V_m leží na opačných stranách p. Vyhráno.

b) V_0 a V_m leží na stejné straně p. Pak p neprotíná úsečku V_0V_m. Ta však rozděluje mnohoúhelník na dvě poloviny – p tedy prochází jen jednou z nich. Tu druhou můžeme zahodit a pokračovat v hledání rekurzivně s mnohoúhelníkem o polovičním počtu bodů.



Algoritmus se tedy skládá ze dvou fází. V první hledáme dva body na opačných stranách p. Pokud takové dva body najdeme, přejdeme do druhé fáze. Jinak se zastavíme, až nám zbude trojúhelník, pro který už úlohu vyřešíme triviálně testem každé ze tří hran. V druhé fázi binárně hledáme hrany protínající se s p postupem popsaným výše.

V obou fázích nás každý krok stojí konstantní čas a zmenšíme jím počet zpracovávaných bodů na polovinu. Tím dosáhneme celkové složitosti $\mathcal{O}(\log n)$.

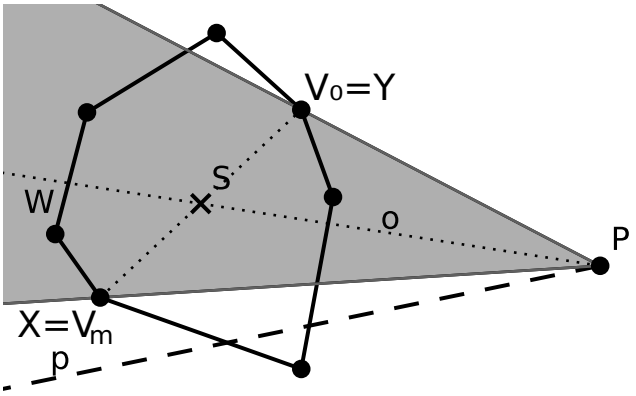
Nalezení správné poloviny bodů

Až dosud je algoritmus docela přímočarý a vymyslitelný. Zatajili jsme ovšem jeden na první pohled drobný detail: jak z polohy p poznáme, kterou polovinu bodů ($V_1, \ldots, V_{m-1}$, nebo $V_{m+1}, \ldots, V_{n-1}$) zahodit?

To překvapivě není vůbec zřejmé. Existuje spousta způsobů, jak to „umlátit“, např. počítáním nějakých vzdáleností a průsečíků. To jsou ale výpočty kom-

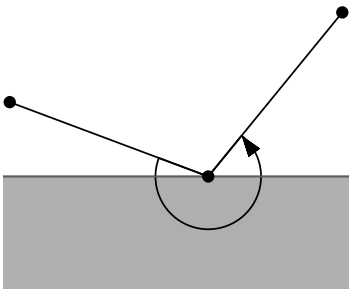
plikované a nepřesné, zkusíme si proto ukázat hezčí, byť možná myšlenkově trochu náročnější řešení.

Představme si úhel V_0PV_m jako jakýsi „stín“. Víme, že p neleží ve stínu (jinak by se protínala s V_0V_m), tedy kdykoli se nějaký objekt nachází celý ve stínu, nemá průsečík s p. Označme si S střed V_0V_m a o polopřímku PS („pseudoosu“ stínu). Ta nám rozdělí rovinu na dvě poloroviny.



Označme si X ten z bodů V_0, V_m, který leží ve stejné polorovině jako p a Y ten opačný.

Podívejme se nyní na sousedy X. Alespoň jeden z nich leží ve stínu, neb kdyby oba ležely na světle, měl by mnohoúhelník vnitřní úhel větší než 180° – viz obrázek níže.



Označme si takového souseda W. Protože $X, W, \ldots, Y$ tvoří konvexní mnohoúhelník, musí jeho obvod „zatáčet“ pořád na stejnou stranu – na obrázku výše doprava (směrem k Y). Tedy pokud je W ve stínu, budou tam i všechny následující vrcholy (přinejmenším ty ve stejné polorovině). Tedy polovina našeho mnohoúhelníku tvořená body $X, W, \ldots, Y$ se nemůže protnout s p: část se jí nachází ve stínu a druhá část v opačné polorovině vůči o než p. Tyto body tedy můžeme (kromě krajních X, Y) zahodit.

Pokud jsou oba sousedé ve stínu, p nemůže mít s mnohoúhelníkem žádný průnik.

Tudíž potřebujeme umět zjistit, (1) ve které polorovině se nachází p, (2) jestli je daný soused X ve stínu.

To jsou ale jen další instance naší operace „poloha bodu vůči polopřímce“. Zvolíme si libovolný bod na p (třeba $P + u$) a podíváme se, na které straně je od o. Pak X je ten z V_0, V_m, který leží na stejné straně. A nějaký soused X je ve stínu, právě když leží od PX na opačné straně, než X od o.

Zbývá nám nějak zařídit onu mýtickou operaci zjištění polohy bodu vůči polopřímce: mějme nějaký bod B a polopřímku q s počátkem Q a směrovým vektorem w. Nyní se nám bude hodit vektorový součin.[50] Z pravidla pravé ruky si snadno rozmyslíte, že $w \times (B - Q)$ je kladný právě tehdy, když B je nalevo od q, záporný, když napravo, a nulový, když leží na q.

Další (byť méně elegantní) způsob je reprezentovat si přímku rovnicí $y = kx + q$. Detaily necháme čtenáři k rozmyšlení.

Program (C): `http://ksp.mff.cuni.cz/viz/26-3-2.c`

Filip Štědronský

26-3-3 Grafová

Požadovaný důkaz je ve své podstatě celkem jednoduchý, k dokázání použijeme *matematickou indukci*.

Nechť je dán graf s N vrcholy, M hranami a minimálním stupněm k. Důkaz provedeme konstrukcí, ze které nám rovnou vyplyne lineární algoritmus. Induktivně sestrojíme vhodnou posloupnost vrcholů:

1. V prvním kroku zvolíme libovolný vrchol v_1.
2. V i-tém kroku máme již cestu z $i-1$ vrcholů, chceme přidat i-tý. Vybereme tedy libovolného souseda v_{i-1}, který ještě nebyl vybrán, a zvolíme jej jako v_i. Pokud takový už neexistuje, skončíme.

Takto jsme dostali v zadaném grafu cestu $v_1 v_2 \ldots v_r$ pro nějaké $r \leq N$.

Nyní nahlédneme, že všichni sousedé vrcholu v_r musí ležet na sestrojené cestě, neboť jinak bychom mohli cestu prodloužit a provést další krok indukce. Vrchol v_r je tedy spojen hranou s aspoň k vrcholy na sestrojené cestě a tudíž určitě existuje vrchol v_s, který je sousedem v_r a pro který platí $s \leq r - k$. Nyní už můžeme jásat, neboť vrcholy $v_s v_{s+1} \ldots v_r$ nám tvoří kružnici délky alespoň $k + 1$.

Postup užitý v důkaze můžeme implementovat přímo jedním průchodem grafu do hloubky, což nám dává časovou i paměťovou složitost $\mathcal{O}(N + M)$.

Program (C++): `http://ksp.mff.cuni.cz/viz/26-3-3.cpp`

Mark Karpilovskij

[50] `http://en.wikipedia.org/wiki/Cross_product`

26-3-4 Kladení pastí

Snažíme se rozmístit pasti v pralese. Přitom musíme dodržet pravidlo, že každá pěšinka spojující dvě křižovatky sousedí alespoň s jednou pastí. V řeči teorie grafů křižovatku nazveme *vrchol* a pěšinku *hrana*. Rozmístění pastí splňující výše popsanou podmínku se obvykle nazývá *vrcholové pokrytí*. Na stavbu pasti v každém vrcholu musíme vynaložit úsilí U_i (zaplatit cenu). Hledáme proto takzvané *minimální vážené vrcholové pokrytí*.

KSP

Podívejme se nejprve na řešení lehčí varianty, ve které je graf tvořen jedinou cestou. Postup potom dokážeme zobecnit i pro těžší verzi.

Lehčí varianta

Pro každou z křižovatek máme dvě možnosti. Buď na ni past umístíme, nebo neumístíme. Pro N vrcholů je tedy všech možných rozmístění pastí 2^N. Některá z těchto rozmístění samozřejmě nejsou vrcholová pokrytí. Každopádně ani tak není v našich silách vyzkoušet všechny možnosti.

řešení

Co kdyby nás pro začátek nezajímalo samotné rozložení pastí, ale pouze celková minimální cena? Zkusme ji počítat postupně pro jednotlivé počáteční úseky cesty délky i. (Délku budeme měřit třeba v počtu vrcholů.)

Zvlášť si zapamatujeme minimální cenu p_i tohoto počátečního úseku délky i za podmínky, že v posledním (i-tém) vrcholu nalíčíme past. Naopak n_i bude značit cenu začátečního úseku délky i v případě, že v i-tém vrcholu past není. Celkovou minimální cenu tohoto úseku cesty snadno spočteme jako $\min(p_i, n_i)$.

Pro první vrchol určíme minimální ceny jednoduše: $p_1 = U_1$ a $n_1 = 0$. Pro ostatní využijeme toho, co jsme spočítali dříve. Ceny tedy budeme počítat postupně pro všechna i od 1 do N. Pokud je v i-tém vrcholu past, tak minimální cenu p_i spočteme z celkové minimální ceny předchozího úseku jako $p_i = \min(p_{i-1}, n_{i-1}) + U_i$. V případě, že past do vrcholu i neumístíme, tak jsme ji museli umístit do předchozího vrcholu $i-1$. Platí tedy, že $n_i = p_{i-1}$.

Tímto postupem snadno v lineárním čase $\mathcal{O}(N)$ spočteme minimální cenu potřebnou k rozmístění pastí. Původně jsme však chtěli minimální vrcholové pokrytí i najít. Zrekonstruujeme jej v opačném pořadí od N do 1.

Pokud se rozhodujeme, zda použít i-tý vrchol, pak jej vybereme tehdy, když se nám to vyplatí. Tedy když platí $p_i \leq n_i$. Pokud však vrchol nevybereme, nutně musíme vybrat vrchol $i-1$. V tom případě se znovu rozhodujeme až u vrcholu $i-2$.

Těžší varianta

Jak postupovat v případě, že cestičky a křižovatky tvoří strom? Minimální ceny chceme opět počítat postupně z předchozích mezivýsledků. Představme si celý strom jako zakořeněný (všechny vrcholy kromě kořene mají jednoznačně určeného otce). Minimální ceny p_i a n_i pak budeme počítat pro všechny podstromy, tj. pro vrchol i se všemi jeho potomky.

Potřebné pořadí pro postupné počítání cen získáme tak, že projdeme celý strom pomocí algoritmu prohledávání do hloubky.[51] Při opuštění vrcholu spočteme ohodnocení p_i a n_i jeho podstromu následovně:

- Pokud je vrchol i list, nastavíme minimální cenu s použitím daného vrcholu $p_i = U_i$.
- Cena bez umístění pasti do listu potom bude $n_i = 0$.
- Pro ostatní vrcholy určíme minimální cenu s použitím pasti ve vrcholu i jako součet ohodnocení daného vrcholu a celkové minimální ceny podstromů všech jeho synů:
$$p_i = U_i + \sum_{j \in Syn(i)} min(p_j, n_j),$$
kde $Syn(i)$ značí množinu synů vrcholu i.
- Pokud do vrcholu i past neumístíme (dle předchozího bodu), musíme pasti nalíčit ve všech jeho synech:
$$n_i = \sum_{j \in Syn(i)} p_j.$$

Při opouštění posledního vrcholu tak spočítáme minimální cenu rozmístění pasti v celém stromě.

Stejně jako v lehčí variantě musíme ještě určit, ve kterých vrcholech máme pasti nalíčit, abychom dodrželi spočítanou minimální cenu. Projdeme tedy znovu náš strom. Na křižovatku i past umístíme, pokud opět platí $p_i \leq n_i$. Když však past do vrcholu neumístíme, musíme ji přidat do všech jeho synů.

Tím máme i pro těžší variantu celkem slušnou časovou a paměťovou složitost $\mathcal{O}(N)$. Strom pouze dvakrát projdeme. Přidané výpočty sice závisí na počtu synů daného vrcholu, dohromady je však synů stejně jako vrcholů.

Pro (NP-)úplnost ještě dodejme, že pro obecný graf není v současnosti známé žádné efektivní řešení.

Program (C++) – lehčí varianta:
`http://ksp.mff.cuni.cz/viz/26-3-4-lehci.cpp`

Program (C++) – těžší varianta:
`http://ksp.mff.cuni.cz/viz/26-3-4-tezsi.cpp`

Jenda Hadrava

[51] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

26-3-5 Rozvrh kovárny

Až na pár originálních řešení, často fungujících a optimálních, zvolili všichni hladový přístup zezadu, čili hladový algoritmus, který si popíšeme. Stručně řečeno: seřadíme si požadavky dle termínu dokončení, bereme je od nejvyšší hodiny dokončení po nejmenší a dáváme do rozvrhu do nejvyšší volné hodiny tak, že vždy bereme nezařazený požadavek s nejvyšší prioritou, který lze v tu hodinu vyrábět.

KSP

Jak vybírat požadavek s nejvyšší prioritou? Řešení napovídal symbol kuchařky u zadání odkazující se na kuchařku o haldách.[52] Na požadavky použijeme haldu, konkrétně maximovou, tedy řazenou od největšího prvku po nejmenší. Náš algoritmus tedy bude takový hladový.

Trochu podrobněji: Nejprve si seřadíme požadavky dle termínu dokončení a zavedeme si proměnnou *hodina*, což bude poslední možná hodina, do níž můžeme rozvrhnout požadavek, a kterou zinicializujeme na maximální hodinu dokončení nějakého požadavku mínus jedna. Pak bereme požadavky sestupně dle hodiny dokončení.

řešení

V každé iteraci se nejprve podíváme, jestli mezi požadavky nejsou nějaké s termínem $hodina + 1$ (plus jedna kvůli tomu, že s výrobou nástroje musíme začít hodinu před termínem). Všechny takové přidáme do haldy. Pak vybereme z haldy požadavek s nejvyšší prioritou, rozvrhneme vybraný požadavek na hodinu *hodina* a tuto proměnnou snížíme o jedna. Skončíme, když se *hodina* dostane pod nulu.

Počet iterací je roven největší hodině dokončení H mezi požadavky. H však není polynomiální v N, ani v délce vstupu, náš algoritmus tedy není ani polynomiální. Nejčastější chybou v řešeních právě bylo, že skončila s algoritmem, který závisel na H.

Oprava na polynomiální algoritmus je však nasnadě: pokud je halda prázdná, rovnou posuneme proměnnou *hodina* na nejvyšší hodinu dokončení nezpracovaného požadavku mínus jedna. Díky tomu v každé iteraci odebereme jeden prvek z haldy a přidáme ho do rozvrhu. Alternativně je možné na začátku inicializovat proměnnou *hodina* na N a všimnout si, že si tím nic nepokazíme, protože stejně nebudeme vyrábět více jak N hodin.

Seřazení dle hodin dokončení jistě zvládneme v $\mathcal{O}(N \log N)$, kde N je počet požadavků. Z kuchařky víme, že haldové operace jako přidávání, nalezení největšího prvku a jeho smazání trvají logaritmus z velikosti haldy, která bude v nejhorším případě obsahovat všech N požadavků.

V každé iteraci vybereme z haldy maximum (za $\mathcal{O}(\log N)$) a smažeme ho, můžeme ale do haldy přidat hodně požadavků. Každý požadavek však dáme do haldy jen jednou, což celkově dává $\mathcal{O}(N \log N)$ času na přidávání. Časovou

[52] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`

složitost jsme tedy určili na $\mathcal{O}(N \log N)$. Co se týče paměti, vystačíme si jistě s prostorem $\mathcal{O}(N)$.

Důkaz správnosti

Zbývá už jen dokázat, že náš algoritmus dává optimální rozvrh, tedy že nelze udělat rozvrh s větším součtem priorit. Budeme postupovat sporem, tedy předpokládat, že pro nějaký seznam požadavků existuje lepší rozvrh než ten, jež našel náš algoritmus. Postupně dojdeme k nějaké zjevné nepravdě, což znamená, že lepší rozvrh neexistuje.

Rozvrh R_1 vytvořený naším algoritmem a lepší rozvrh R_2 se musí někde lišit, přičemž nás zajímají jen rozdíly v prioritách požadavků pro konkrétní hodiny. Jelikož R_2 má vyšší součet priorit, musí existovat hodina H, ve které má lepší rozvrh požadavek P s vyšší prioritou, než má požadavek zpracovávaný v hodině H v našem rozvrhu.

Provedeme výměnu a upravíme lepší rozvrh R_2 na podobný rozvrh se stejným součtem priorit. Požadavek P musíme vyrábět v našem rozvrhu R_1 v hodině H', kde $H' > H$, jinak bychom ho v hodině H vytáhli z haldy. Nyní zaměníme v rozvrhu R_2 požadavky v hodinách H a H' a všimneme si, že jsme si nepokazili rozvrh: požadavek z hodiny H' vyrábíme dříve a požadavek z hodiny H lze v našem rozvrhu vyrábět v hodině H', takže to musí jít i v rozvrhu R_2.

Jelikož upravený rozvrh R_2 má vyšší součet priorit než R_1, tak i po výměně musí existovat hodina, v níž se v R_2 vyrábí požadavek s vyšší prioritou než v R_1. Mohli bychom tedy takovéhle výměny provádět donekonečna, což však nejde, neboť po každé výměně vzroste alespoň o jedna počet hodin, kde se shodují rozvrhy R_1 a R_2, konkrétně o hodinu H'. Čili máme kýžený spor.

Tím jsme úspěšně vykovali algoritmus. Užijte si zbytek zimy ... tedy, chtěl jsem říct jara.

Program (C++): `http://ksp.mff.cuni.cz/viz/26-3-5.cpp`

Pavel „Paulie“ Veselý

26-3-6 Trezor

Všichni, kdo se pokusili o tuto úlohu, se nezastavili u lehčí verze a rovnou se pustili do verze plné. Přesto se na chviličku u lehké verze zastavme, uvidíme, že se dá řešit pěkným trikem.

V lehčí verzi nám jednotlivé vnitřní klíče dají dohromady číslo ve dvojkovém zápise – sérii jedniček a nul – a my chceme všechny nuly změnit na jedničky. To určitě můžeme udělat tak, že použijeme právě klíče s exponenty odpovídající pozicím nul v tomto čísle. Nejde to však lépe, s menším počtem klíčů, než kolik je v čísle nul?

Nejde. Žádným způsobem totiž přidáním jednoho klíče nezměníme více než jednu nulu na jedničku. I když by došlo k „přetečení“ nějakého řádu, tak se

nám na tomto místě objeví nula a zbytek čísla se chová stejně, jako kdybychom jedničku přičítali k o jedna většímu řádu. Zadání úlohy se tedy dá přeformulovat přímo na spočítání počtu nul ve dvojkovém zápisu součtu vnitřních klíčů.

Protože se nám v lehčí variantě stejné klíče neopakují, tak rovnou dostaneme číslo ve dvojkové soustavě, kde počet jedniček odpovídá počtu vnitřních klíčů. Stačí tedy v lineárním čase a konstantní paměti při čtení vstupu najít největší exponent vnitřního klíče a odečíst od něj celkový počet vnitřních klíčů, čímž rovnou dostaneme potřebný počet vnějších klíčů.

Při řešení těžší varianty ale již bude potřeba hodnoty klíčů nějak sčítat. Nemusíme operovat přímo s hodnotami, bohatě nám stačí sčítat exponenty. Označme si pomocí N počet exponentů (vnitřních klíčů) na vstupu a jako M maximální exponent, který se na vstupu může vyskytnout. Podle velikosti M můžeme úlohu řešit dvěma různými způsoby:

Pro malé M, pokud $M < N \log N$, je nejvýhodnější použít *přihrádkové třídění*. Maximální exponent, ke kterému se můžeme v průběhu výpočtu dostat, je $M + \log N$ (kdyby všech N vnitřních klíčů bylo maximálního exponentu), takže si vyrobíme takto velké pole přihrádek a pak v lineárním čase spočítáme počty jednotlivých exponentů na vstupu.

Pak nám stačí toto pole projít odzadu, počítat počet nul, a kdykoliv se nám v nějaké přihrádce vyskytne číslo větší jak jedna, tak ho necháme „přetéct“ – v aktuální přihrádce necháme zbytek po dělení dvěma (tedy buď 0 nebo 1) a do další přihrádky přičteme (celočíselnou) polovinu hodnoty z aktuální přihrádky. Tím úlohu vyřešíme v čase $\mathcal{O}(N + M)$ a s pamětí $\mathcal{O}(M + \log N)$.

Program (Python) – přihrádky:
`http://ksp.mff.cuni.cz/viz/26-3-6-prihradky.py`

Pro velká M (klidně řádově větší jak N) by se nám však již pole přihrádek do paměti vůbec nemuselo vejít, nemluvě o tom, že by v něm mohly být velké „díry“, ve kterých bychom trávili zbytečně moc času. Pak je výhodnější zvolit jiný postup.

Všech N vnitřních klíčů si můžeme na začátku programu v čase $\mathcal{O}(N \log N)$ seřadit, seřazené naskládat do spojového seznamu, a pak ho opět jako v předcházejícím případě projít od nejmenšího. Zde však bude drobný rozdíl v tom, že pokud ve spojovém seznamu ještě položka pro další exponent není, tak ji založíme. V tomto případě se dostáváme na časovou složitost $\mathcal{O}(N \log N)$ (nejdéle trvá úvodní setřídění) a paměťovou $\mathcal{O}(N)$.

Poznámka: V ukázkovém programu je použit navíc trik, díky kterému se bez spojáku nakonec obejdeme úplně. Stačí si při průchodu jen pamatovat, kolik stejných exponentů jsme v seřazeném poli už potkali. Kdo chcete, nahlédněte.

Program (C) – velké exponenty:
`http://ksp.mff.cuni.cz/viz/26-3-6-velke-exp.c`

Jirka Setnička

26-3-7 Sopečné pokrytí

Naším úkolem je pro mapu $R \times S$ najít posunutí mřížky s velikostí buňky $A \times B$ takové, že počet buněk obsahujících alespoň jeden sopečný kráter je minimální. K takovému výpočtu se nám bude hodit umět rychle zjišťovat, jestli nějaký konkrétní obdélník o velikosti $A \times B$ obsahuje sopečný kráter.

My si ukážeme, jak v čase $\mathcal{O}(RS)$ vyrobit strukturu, pomocí které v čase $O(1)$ dokážeme zjistit počet kráterů v libovolném podobdélníku, speciálně tedy i v podobdélnících $A \times B$.

Jedná se o takzvané *dvourozměrné prefixové součty*. Mohli jste je potkat například v kuchařce o základních algoritmech.[53] Pokud nepotkali, nevadí, myšlenku zde zopakujeme:

Pro každé políčko si předpočítáme, kolik kráterů obsahuje horní levý obdélník, který má pravý dolní roh právě v tomto políčku. Tedy políčko na pozici $[x, y]$ bude mít hodnotu počtu kráterů v obdélníku $([1, 1], [x, y])$. Tento výpočet si můžete sami rozmyslet, nebo se na něj podívat ve zdrojovém kódu – není to nic těžkého.

Dvourozměrné pole dvourozměrných prefixových součtů si označme P, pak počet kráterů v obdélníku $([a, b], [c, d])$ získáme jako:

$$K[a, b, c, d] = P[c, d] - P[c, b - 1] - P[a - 1, d] + P[a - 1, b - 1]$$

Když již máme vybudovanou pomocnou datovou strukturu, zkusíme každé možné posunutí větších buněk. Každé políčko se právě jednou stane nějakým pravým dolním rohem buňky (pro právě jedno posunutí).

V praktické realizaci tak pro každé políčko v mapě $R \times S$ zjistíme, jestli obdélník $A \times B$ mající pravý dolní roh v tomto políčku obsahuje kráter a pokud ano, tak k příslušnému posunutí mřížky přičteme jedničku. Tím jsme vlastně hotovi, už se jen stačí podívat, při jakém z $A \times B$ posunutí jsme potřebovali nejméně buněk mřížky.

Časová i paměťová složitost algoritmu je $\mathcal{O}(RS)$. Můžete se také podívat do vzorového kódu psaného v jazyce C++.

Program (C++): `http://ksp.mff.cuni.cz/viz/26-3-7.cpp`

Karel Tesař

[53] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy`

26-3-8 Zdivočelá počítadla

Úkol 1 – aritmetické instrukce

Začneme zkratkou `ADD` x, y, z pro uložení součtu $x + y$ do registru z. Nejprve zkopírujeme x do z a y do pomocného registru t. Poté budeme postupně dekrementovat t a inkrementovat z.

(Všimněte si, že naše zkratka nepotřebuje, aby registry x, y a z byly navzájem různé. O to se budeme snažit i u ostatních aritmetických operací. Jen musíme dodefinovat `MOV` x, x, aby neprovedl nic.)

```
        MOV x,z
        MOV y,t
        JZ  t,Y
X:      INC z
        DEC t
        JNZ t,X
Y:
```

S odčítáním `SUB` x, y, z si poradíme podobně. Nezapomeneme na případ, kdy $x < y$, na což máme podle zadání odpovědět nulou.

```
        MOV x,z
        JZ  y,Y
        MOV y,t
X:      DEC z
        DEC t
        JNZ t,X
Y:
```

Násobení `MUL` x, y, z pak definujeme jako opakované sčítání (s je další pomocný registr):

```
        CLR z
        JZ  y,Y
        MOV y,s
X:      ADD x,z,z
        DEC s
        JNZ s,X
Y:
```

Ještě se nám v následujících úkolech bude hodit dělení `DIV` x, y, p, q, které do p uloží celou část podílu x/y a do q zbytek po tomto dělení. Implementujeme jako opakované odčítání, jen pokaždé odečteme $y - 1$ a před odečtením zbývající jedničky zkontrolujeme, zda se dělenec už nevynuloval.

```
        MOV x,t
        CLR p
        MOV y,s
        DEC s       ; s = y-1
X:      MOV t,q
        SUB t,s,t
        JZ  t,Y
        DEC t
        INC p
        JMP X
Y:
```

Úkol 2 – tradiční závorky

Závorky na vstupu dostaneme zakódované do jednoho velkého čísla. To potřebujeme rozebrat na desítkové číslice, což se daleko snáz dělá od konce než od začátku: vydělením deseti odstraníme poslední číslici, zbytek nám řekne, jaká číslice to byla.

Na samotnou kontrolu uzávorkování použijeme jako obvykle počítadlo, ale jelikož vstup zpracováváme zprava doleva, bude tentokrát udávat, kolik závorek bylo uzavřeno, ale ještě ne otevřeno. Za každou) ho tedy zvýšíme o 1 a za každou (snížíme. Nesmí přitom nikdy klesnout pod nulu a na konci vstupu musí vyjít nulové.

S našimi aritmetickými instrukcemi je implementace hračka. V registru x očekáváme vstup, do y zapisujeme výstup, z nám bude počítat závorky, v d bude uložena konstanta 10 a r bude obsahovat právě odebranou číslici.

```
      CLR y        ; zatím špatně
      CLR z        ; počítadlo závorek
      CLR d
      INC d  (10x) ; d=10
NEXT: JZ  x,END
      DIV x,d,x,r  ; další závorka?
      DEC r
      JZ  r,OPEN
      INC z        ; zavírací
      JMP NEXT
OPEN: JZ  z,DONE   ; otvírací
      DEC z
      JMP NEXT
END:  JNZ z,DONE   ; závěrečný test
      INC y        ; správně
DONE:
```

Úkol 3 – simulace Turingova stroje

Nejprve využijeme trik z 1. série, abychom daný Turingův stroj převedli na jednopáskový.

Nyní navrhneme, jak do registrů zakódovat konfiguraci stroje. Abecedu stroje očíslujeme od 0 do nějakého $A-1$, přičemž 0 bude znamenat mezeru. Pásku rozdělíme v místě hlavy. Levou část uložíme do registru ℓ jako číslo v soustavě o základu A, číslice nejnižšího řádu bude odpovídat znaku těsně před hlavou (zde se hodí, že 0 je mezera, takže vlevo přirozeně vznikne nekonečně mnoho mezer). Znaky vpravo od hlavy zakódujeme obdobně do registru r, nejnižší řád bude odpovídat znaku bezprostředně za hlavou.

Zbývá nějak reprezentovat znak, na kterém právě stojí hlava, a aktuální stav řídicí jednotky stroje. To zakódujeme do pozice v počítadlovém programu,

kde se zrovna nacházíme. Program bude tvořen *velikost abecedy* × *počet stavů* podobnými bloky, změna stavu nebo aktuálního znaku bude pouhý skok do jiného bloku.

Stačí tedy umět posouvat hlavu. Popišme posun doprava: Aktuální znak se přesune do levé části pásky, takže registr ℓ vynásobíme velikostí abecedy a přičteme aktuální znak. Naopak registr ℓ vydělíme velikostí abecedy a zbytek nám řekne, jaký znak se stal aktuálním. Posun doleva vyřešíme obdobně.

Úkol 4 – redukce počtu registrů

KSP

Kdyby nám stačilo dokázat, že stačí nějaký pevný počet registrů, můžeme k tomu použít předchozí úkol. Ze zadání víme, že každý počítadlový program lze přeložit na ekvivalentní Turingův stroj, v předchozím úkolu jsme se naučili převést ho zpět. Kombinací obou převodů získáme počítadlový program s pevným počtem registrů. Kdybychom promysleli detaily (zejména počty pracovních registrů v aritmetických instrukcích), dostali bychom se na něco jako 5 registrů. Ukážeme, že stačí méně.

řešení

Mějme program, který používá registry $r_1, \ldots, r_k$. Jejich stav dovedeme zakódovat do jediného čísla

$$r = p_1^{r_1} \cdot \ldots \cdot p_k^{r_k},$$

kde $p_1, \ldots, p_k$ jsou navzájem různá prvočísla. Z jednoznačnosti prvočíselného rozkladu plyne, že zakódovaný stav lze dekódovat jediným možným způsobem.

Instrukci `INC` r_i přeložíme na násobení registru r prvočíslem p_i. Ačkoliv bychom mohli použít zkratku `MUL`, raději si násobení naprogramujeme sami využívajíce toho, že p_i je konstanta. Bude nám stačit jediný pracovní registr t.

```
      CLR t
X:    INC t   (p_i-krát)
      DEC r
      JNZ r,X
Y:    INC r       ; přelijeme zpět do r
      DEC t
      JNZ t,Y
```

Podobně `DEC` r_i přeložíme na dělení registru r číslem p_i, ovšem musíme si dát pozor, abychom v případě, kdy r není dělitelné, vše vrátili do původního stavu.

```
      CLR t
      ; Opakovaně odčítáme p_i.
DIV:  JZ  r,OK
      DEC r
      JZ  r,R1
      DEC r
      JZ  r,R2
```

```
      (... dalších p_i-3 dvojic ...)
      DEC r
      INC t
      JMP DIV

      ; Nebylo to dělitelné,
      ; pozice v programu udává zbytek.
      (... p_i-3 inkrementů jako níže ...)
R2:   INC r
R1:   INC r
      ; Nakonec k r přičteme t * p_i.
T:    JZ  t,DONE
      INC r   (p_i-krát)
      DEC t
      JMP T

      ; Povedlo se, přelijeme zpět do r,
      ; které už je touto dobou nulové.
OK:   INC r
      DEC t
      JNZ t,OK
DONE:
```


Podmíněný skok `JNZ` vyřešíme obdobně: pokusíme se o `DEC`, pokud se povede, zvrátíme jeho účinek dalším `INC` a skočíme. Pakliže se nepovede, jen uvedeme registr r do původního stavu a pokračujeme v programu.

Vypadá to tedy, že každý program dokážeme upravit, aby mu postačily pouhé dva registry r a t. Jenže ouha: ještě musíme umět zakódovat vstup do našeho „exponenciálního“ kódování, a na konci programu zase dekódovat výstup. K tomu bohužel potřebujeme další registr.

Kódování vstupu bude probíhat tak, že na počátku položíme $r = 1$ a pak budeme dekrementovat vstupní registr a přitom inkrementovat jeho zakódovaný obraz v r. Podobně dekódování bude dekrementovat zakódovaný obraz výstupního registru a inkrementovat skutečný výstupní registr. Na obojí nám postačí tři registry.

Dodejme ještě, že je známo, že se dvěma registry takové kódování provést nelze. Důvod je prostý: nelze spočítat žádnou funkci, která roste exponenciálně. Dvojregistrový stroj tedy nemůže být univerzální. (Důkaz viz Rich Schroeppel: “A Two-counter Machine Cannot Calculate 2^N”, Massachusetts Institute of Technology, Artificial Intelligence Memo #257.)

Úkol 5 – minimální instrukční sada

Někteří řešitelé dokázali vymyslet jednoinstrukční sadu, ale pokaždé nějakým ošklivým trikem. Třeba instrukcí, jejíž součástí je konstanta, která instrukci

řekne, jakou operaci má provést. Zde předvedeme také mírně podlé, ale snad o ždibec elegantnější řešení.

Naše instrukce se bude jmenovat `IDJNZ` x, y, p (increment, decrement and jump if not zero) a bude fungovat takto: Nejprve otestuje registr y na nulu. Pak inkrementuje registr x, načež dekrementuje registr y (pokud by vzniklo záporné číslo, zapíše nulu). Nakonec skočí na adresu p, pokud na začátku byl registr y nenulový. V opačném případě nikam neskáče.

`INC` x zapíšeme jako `IDJNZ` x, t, p, kde t je nějaký pracovní registr a p adresa těsně za instrukcí.

`DEC` x přeložíme analogicky na `IDJNZ` t, x, p.

`JNZ` x, p upravíme na `IDJNZ` x, x, p.

Martin „Medvěd“ Mareš

26-4-1 Kamínkový solitér

Mnoho řešitelů odeslalo správná řešení, to nás moc těší.

Lehčí varianta

Lehčí varianta byla, jak mnozí poznali, Euklidův algoritmus na zjišťování největšího společného dělitele (nsd). Viz kuchařku o teorii čísel.[54]

Správnou odpovědí tedy bylo, že nám na každé hromádce zbude $\mathrm{nsd}(a, b)$, kde a a b jsou počty kamínků na jedné a druhé hromádce.

Někteří řešitelé si povolili odčítat i stejně velké hromádky od sebe, to pak vede k tomu, že celkový počet zbylých kamínků bude $\mathrm{nsd}(a, b)$.

Těžší varianta

Jak se nám změní úloha pro více hromádek?

S hromádkami, které mají 0 kamínků, se vypořádáme tak, že je prostě zahodíme, ty nám nijak nemohou ovlivnit průběh hry ani celkový výsledek (protože odečtením této hromádky od jiné se stav hry nezmění).

Nyní musíme dokázat, že $\mathrm{nsd}(a, b, c) = \mathrm{nsd}(a, \mathrm{nsd}(b, c))$.

Víme, že $a > 0, b > 0, c > 0$ (jinak bychom je neuvažovali mezi hromádky). Dále platí, že $a = g \cdot a'$, $b = g \cdot b'$, $c = g \cdot c'$. Navíc platí, že $\mathrm{nsd}(b, c) = g \cdot z'$, protože $\mathrm{nsd}(b, c)$ získáme odečítáním hromádek b a c od sebe. Vždy tu vyšší odčítáme od té nižší, a tím zůstane g zachováno. To platí proto, že $b - c = (g \cdot b') - (g \cdot c') = g \cdot (b' - c')$. Pak platí, že pokud $g = \mathrm{nsd}(a, b, c)$, pak i $\mathrm{nsd}(a, g \cdot z') = g$.

Tedy správná odpověď byla, že na každé hromádce zbude $\mathrm{nsd}(h_i)$, případně že zbude právě $\mathrm{nsd}(h_i)$ (pokud si řešitel povolil odčítat od sebe i stejně velké hromádky). Dokázané pozorování $\mathrm{nsd}(a, b, c) = \mathrm{nsd}(a, \mathrm{nsd}(b, c))$ využijeme pro náš algoritmus, který bude umět v $\mathcal{O}(1)$ poradit další tah hry. Algoritmus bude vypadat takto:

Nejprve zahodíme hromádky, které mají hodnotu 0. Nyní vezmeme 1. a 2. hromádku, odčítáme vždy od větší menší, dokud nemají stejnou hodnotu (která je, jak už víme, nsd původních hodnot těch dvou hromádek). Poté to samé provedeme s 2. a 3. (zde dostáváme nsd původních hodnot všech předchozích hromádek a té aktuální), pak se 3. a 4., …, $(n-1)$-tou a n-tou. Tím jsme získali na n-té a $(n-1)$-té hromádce nsd všech hromádek. Nyní potřebujeme tento dělitel dostat k předchozím. Tedy provedeme odčítání hromádek zpětně (nejprve s hromádkou $n-1$ a $n-2$, pak s $n-2$ a $n-3$, …, 1 a 2). Když skončíme, máme na všech nenulových hromádkách $\mathrm{nsd}(h_i)$ a 0 na nulových.

Vojta Sejkora

[54] `http://ksp.mff.cuni.cz/viz/kucharky/teorie-cisel`

26-4-2 Výroba amuletu

Úloha se značně podobá známému problému hledání *nejdelší společné podposloupnosti*, popsanému např. na Wikipedii,[55] včetně velice přímočarého řešení. Zmiňujeme se o něm i v naší kuchařce o dynamickém programování,[56] ale algoritmus popsaný tam by se hůře upravoval pro potřeby naší úlohy.

KSP

Předpokládejme nejdříve pro jednoduchost, že ceny všech korálků jsou stejné, $c_R = c_G = c_B = 1$. Upravený amulet bude obsahovat tři druhy korálků: *nové* (které jsme vyrobili pro potřebu hesla), *recyklované* (které jsme použili z původního amuletu jako součást hesla) a *zbytečné* (které zbyly z původního amuletu, ale nejsou použity k vytvoření hesla).

My hledáme řešení s co nejméně novými korálky. Ale ježto nových a recyklovaných dohromady je vždy stejně (jako délka hesla), můžeme zrovna tak hledat řešení obsahující nejvíce recyklovaných korálků. Vzhledem k tomu, že recyklované korálky tvoří (z definice) společnou podposloupnost amuletu a hesla, můžeme použít algoritmus z odkazovaného článku pro nalezení jejich nejdelší společné podposloupnosti (NSP), a tak zjistit, které korálky chceme v optimálním řešení recyklovat.

řešení

Ukažme si to na příkladu pro amulet `RRRGGGBBB` a heslo `RGBRGB`:

```
Původní amulet           RRRG  GGBBB
Heslo                    R  GBR GB
Nejdelší spol. podposl.  R  G   GB
Vyrobeno                     BR
Nový amulet              RRRGBRGGBBB
Typ korálku              rzzrnnzrrzz
```

Z výstupu algoritmu kromě samotné podoby NSP (`RGGB`) snadno zjistíme, i na jakých pozicích v amuletu a hesle se tyto recyklované korálky budou nacházet. Jedním průchodem NSP si tedy můžeme ke každému recyklovanému korálku uložit jeho pozici v původním amuletu, a z toho už snadno dopočítáme, kam se vkládají korálky nové.

Tady je trochu problém s interpretací zadání, neboť přidáváním nových korálků se nám původní posouvají a není příliš jasné, co vlastně znamená „i-tý korálek“. V příkladu výše nejprve vypíšeme „vlož modrý korálek za čtvrtý“ a pak červený za … čtvrtý, nebo pátý? Nejjednodušším způsobem, jak se problému elegantně zcela vyhnout, je vypisovat přidané korálky odzadu. Tak nám žádný přidaný korálek nemůže ovlivnit číslování míst, na která budeme přidávat

[55] `http://en.wikipedia.org/wiki/Longest_common_subsequence#Solution_for_two_sequences`

[56] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

později. V našem případě by to tedy znamenalo nejdříve vložit červený korálek za čtvrtý a pak modrý za čtvrtý (čímž se červený odsune o jednu pozici doprava).

Složitosti algoritmu dominuje hledání NSP, které zvládneme v čase a paměti $\mathcal{O}(|A| \cdot |H|)$, kde $|A|$ je délka původního amuletu a $|H|$ je délka hesla. Zbývá se vypořádat s tím, že výrobní ceny korálků nemusí být jednotkové. Ale není těžké si rozmyslet, že algoritmus NSP můžeme jednoduše upravit tak, aby místo nejdelší hledal *nejdražší* společnou podposloupnost. Prostě všude místo délek posloupností počítáme jejich ceny a při rozšiřování posloupnosti místo jedničky přičítáme cenu přidávaného korálku. Složitost tím nijak neovlivníme. Pro přesnou podobu upraveného algoritmu nahlédněte do programu.

Program (C): `http://ksp.mff.cuni.cz/viz/26-4-2-nsp.c`

Filip Štědronský

26-4-3 Obnovené spojení

V této úloze se nám bohužel vloudila chyba do zadání úlohy, kde mělo být uvedeno, že jeskyně, které mezi sebou propojujeme, si očíslujeme od 1 do n. Proto se řada z vás spokojila s triviálním řešením pomocí třídění nebo pomocí hešování. Chyba je ovšem na naší straně, takže jsem taková řešení jen mírně bodově odlišil od těch, která si chybějící předpoklad domyslela a byla zcela správně.

Optimálním řešením je dvojité užití přihrádkového třídění. Nejprve si každou dvojici jeskyň uspořádáme vzestupně. Nyní roztřídíme dvojice do přihrádek podle hodnoty druhé jeskyně. V praxi to můžeme reprezentovat třeba polem spojových seznamů délky n, kde ke každé hodnotě druhé jeskyně budeme mít spojový seznam hodnot první jeskyně. K druhému třídění využijeme výsledek prvního. Budeme postupně procházet každý spojový seznam od nejmenší hodnoty druhé jeskyně k největší a zatřiďovat do jiných přihrádek tentokrát podle první jeskyně. Všimněme si, že pokud k jedné první jeskyni existují dvě druhé, tímto průchodem jako první narazíme na tu s menším číslem. Z toho vyplývá, že čísla ve výsledných přihrádkách budou setříděná od nejmenšího k největšímu.

Tím jsme dostali všechny dvojice setříděné primárně podle první a sekundárně podle druhé jeskyně. Takto setříděná data už snadno projdeme a vyhneme se při vypisování duplikátům například tak, že si pamatujeme, co jsme naposledy vypsali.

Zacházeli jsme s $\mathcal{O}(n)$ přihrádkami a s $\mathcal{O}(m)$ jeskyněmi, přitom jsme na každou jeskyni spotřebovali konstantní čas při jejím zapracování do některého seznamu, výsledná časová i paměťová složitost tohoto řešení je tedy $\mathcal{O}(n + m)$.

Program (C++): `http://ksp.mff.cuni.cz/viz/26-4-3.cpp`

Mark Karpilovskij

26-4-4 Skládání mapy

Častou chybou u úlohy bylo to, že jste předpokládali celočíselnost souřadnic a pokoušeli jste se obdélníky s mapou napasovat do nějaké tabulky, o celočíselnosti jsme však nic neslíbili. Ta by se sice dala zachránit nějakou kompresí souřadnic (seřadili bychom si neceločíselné souřadnice za sebe a očíslovali), ale větší problém byl v tom, že taková tabulka by mohla být obrovská – představte si dva malé kusy mapy vzdálené od sebe milion políček. Inicializace takové tabulky by nám zabrala neúměrně mnoho času, a proto bylo nutné vydat se jinudy.

KSP

Nejdříve se zkusíme zamyslet nad tím, jak by se úloha řešila jen v jednom rozměru, tedy kdybychom namísto obdélníků měli jen úsečky na přímce. Takový případ je přesně stvořený pro zjednodušený intervalový strom..[57]

Intervalový strom nám ve zkratce umožňuje provádět dotazy nad intervaly, tedy s kterými intervaly se protíná hledaný interval. My ale hledáme jen jeden bod a navíc nás zajímá jen počet obdélníků mapy, se kterými se protneme, proto nám stačí použít jednodušší verzi intervalového stromu.

řešení

Z původního intervalového stromu si vypůjčíme jen vyhledávání podle konců pokrytých intervalů ve vnitřních vrcholech, a protože naše dotazy půjdou vždy až do samotných listů intervalového stromu, stačí nám mít počty pokrytých intervalů jen v těchto listech. Později u dvourozměrné varianty budeme muset do vnitřních vrcholů přidat nějaké další hodnoty, ale u jednorozměrné verze bez nutnosti měnit intervaly za běhu si vystačíme s touto lehčí verzí.

Takový strom si lehce postavíme v čase $\mathcal{O}(N \log N)$ (kde N je počet úseček) tak, že si nejdříve setřídíme souřadnice začátků a konců úseček, budeme je postupně procházet (za začátek přičteme jedničku k aktuálnímu počtu pokrytých úseček, za konec zase odečteme) a ukládat do listů úplného binárního stromu. Při obcházení stromu navíc ještě do vnitřních vrcholů uložíme konce intervalů (pro vyhledávání) a jsme hotovi. Vyhledání pak lehce zvládneme v čase $\mathcal{O}(\log N)$ na dotaz.

Dva rozměry

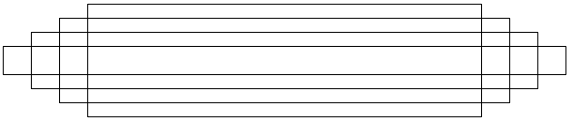
Jak postup výše zobecnit pro dva rozměry? Ano, použijeme dvourozměrné intervalové stromy. Nejdřív si vyrobíme intervalový strom pro jeden rozměr (jako kdybychom kusy mapy promítli jako úsečky třeba na osu x). Tím si rovinu rozdělíme na jednotlivé (v tomto případě svislé) pásy, kde máme jen spodní a vrchní okraje obdélníků mapy. Ještě však potřebujeme vyhledat správnou oblast uvnitř těchto pásů.

Proto si v každém listu prvního intervalového stromu ubytujeme další intervalový strom, který bude mít na starost pouze tento pás (každý pás si totiž můžeme zase představit jako úsečky v ose y odpovídající jednotlivým kusům mapy). Vyhledání je pak dvoustupňové: nejdříve v prvním stromě najdeme správný

[57] Viz kuchařku čtvrté série.

pás a v tomto pásu pomocí odpovídajícího stromu najdeme přesné místo, kde je umístěn bod, na který se ptáme. Takový dotaz zvládneme v $\mathcal{O}(\log N)$.

S dvourozměrnými intervalovými stromy se ale musí opatrně – spousta zdrojů sice uvádí, že se dají vybudovat v čase $\mathcal{O}(N \log^2 N)$, ale to jen, pokud se na ně uplatní nějaký trik. Ti z vás, kteří použili ve svých řešeních pojem 2D-intervalového stromu jako všemocný blackbox bez toho, aby se nad ním a jeho použitím a problémy zamysleli, nějaký ten bod bohužel ztratili.



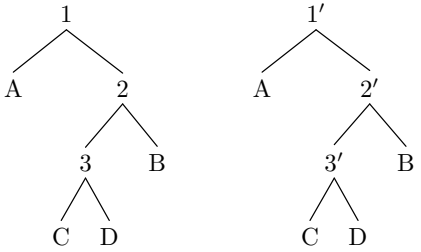
Pokud by totiž obdélníky vypadaly třeba jako na obrázku výše, budeme mít řádově N intervalů v prvním (x-ovém) stromě a každý z y-ových stromů bude také držet řádově N intervalů. Pokud si budeme každý z menších stromů ukládat samostatně, dostaneme se na paměť $\mathcal{O}(N^2)$, a k jejich konstrukci tedy i na stejnou časovou složitost.

Persistentní datová struktura

Největším problémem konstrukce našeho 2D-intervalového stromu bylo to, že stromy v druhé úrovni vypadaly pořád skoro stejně, ale pokaždé jsme je museli konstruovat celé znovu. Nedá se nějak využít již dříve zkonstruovaných stromů?

Dá, a takovému triku se říká *persistence*. V našich stromech vedou všechny hrany jen dolů (nepotřebujeme zpětné hrany do otce), a tak klidně můžeme udělat to, že si zkonstruujeme nový strom tak, že se bude odkazovat na části již postavených stromů. Stačí se držet základního pravidla persistence: Staré hodnoty nikdy neměním, když potřebuji něco změnit, vyrobím si od toho kopii.

Kdybychom například chtěli ve stromu níže změnit hodnotu ve vrcholu s číslem 3, stačí nám vyrobit si kopii tohoto vrcholu, kopii jeho otce (a tak dále, až ke kopii jeho kořenu) a ty odkázat na již existující podstromy. Když se nyní vydám z kořene $1'$, budu mít k dispozici nový strom, ale když se vydám z kořene 1, uvidím strom v původním stavu. Máme tedy dva stromy velikosti N, ale výrobou nového jsme museli strávit pouze čas $\mathcal{O}(\log N)$ (délku cesty z vrcholu do kořene).



V našem případě persistenci použijeme k vyrábění jednotlivých intervalových stromů v pásech. Mezi jednotlivými pásy dochází jen k tomu, že nějaký obdélník zmizí, nebo se jiný objeví, tedy potřebujeme přičíst nebo odečíst nějakou hodnotu od nějakého intervalu hodnot. Počkat, co když ale budeme muset modifikovat řádově N vrcholů oproti předchozímu stromu, nepokazí se nám časová složitost?

Kdybychom to dělali jako dosud (se zjednodušenými intervalovými stromy), tak by se pokazila. Teď už budeme potřebovat plné intervalové stromy (tak, jak jsou popsané v kuchařce), které nám umožňují i rychlou aktualizaci hodnot. Pokud chceme modifikovat nějaký celý interval, můžeme to rozložit na modifikaci $\mathcal{O}(\log N)$ vrcholů intervalového stromu (do vnitřních vrcholů si přičteme nějaké plus a mínus jedničky, přes které pak při dotazu projdeme a vezmeme je v potaz).

Modifikace každého vrcholu v persistentním intervalovém stromu nám trvá $\mathcal{O}(\log N)$, takže nám (kromě prvního) výroba každého z $\mathcal{O}(N)$ intervalových stromů trvá $\mathcal{O}(\log^2 N)$ a zabere maximálně stejně paměti. Vyhledávání v nich je pak stejné jako v minulém případě.

Dohromady tedy potřebujeme na výrobu celé struktury čas $\mathcal{O}(N \log^2 N)$ a na Q dotazů čas $\mathcal{O}(Q \log N)$. Paměti nám stačí $\mathcal{O}(N \log^2 N)$.

Program (C): `http://ksp.mff.cuni.cz/viz/26-4-4.c`

Jirka Setnička

Poznámka: Algoritmus můžeme ještě trochu vylepšit, pokud si všimneme, že při přidávání či odebírání obdélníka sice upravujeme až logaritmický počet vrcholů, ale všechny leží v blízkém okolí dvou cest z kořene do listu. Pokud persistenci naučíme zaznamenat všechny tyto změny najednou, zkopírujeme celkově jen $\mathcal{O}(\log N)$ vrcholů stromu, takže výroby struktury klesne na $\mathcal{O}(N \log N)$.

Dalšími triky by šlo snížit paměťové nároky na $\mathcal{O}(N \log N)$, ale to už se do tohoto textu nevejde. Kdybyste chtěli, příležitostně se zeptejte :)

Martin „Medvěd“ Mareš

26-4-5 Místo pro tábor

Úlohu budeme řešit jen pro těžší variantu. Řešení lehčí varianty je úplně stejné, akorát jen v jednom rozměru. Zadáním úlohy je najít obdélník velký $r \times s$, kde potřebujeme převézt co nejméně hlíny na jeho vyrovnání. Tedy takový, v kterém na políčkách pod průměrnou výškou obdélníka chybí co nejméně hlíny. Pro další část řešení zadefinujeme $n = rs$ a $N = RS$.

Pokud v daném obdélníku máme průměrnou výšku V a právě k políček s podprůměrnou výškou $p_1, \ldots, p_k$, musíme převézt právě

$$k \cdot V - \sum_{j=1}^{k} p_j$$

hlíny. Chtěli bychom tedy pro všechny možné obdélníky $r \times s$ tyto hodnoty spočítat.

Nejdříve spočítáme průměrné výšky. Pomocí dvourozměrných prefixových součtů snadno zvládneme spočítat součty výšek v obdélnících v čase $\mathcal{O}(RS)$ a tyto hodnoty vydělíme n. Průměrné výšky si setřídíme a označíme jako $X_1, X_2, \ldots, X_{(R-r+1)(S-s+1)}$. Zároveň zvolíme $X_0 = 0$.

Pro výpočet hodnot $\sum_{j=1}^{k} p_j$ se nám budou hodit dva dvourozměrné součtové intervalové stromy S a T.[58] Do stromu T budeme iterativně přidávat hodnoty výšek v původní mapě a do stromu S budeme vkládat jedničky na místa, kde se přidané prvky vyskytují.

V i-té iteraci do stromů vložíme všechny hodnoty z intervalu $\langle X_{i-1}, X_i)$. Pokud si navíc pro každé X_i budeme pamatovat, ke kterému obdélníku $r \times s$ patří, rovnou pro něj zvládneme spočítat hodnoty k a $\sum_{j=1}^{k} p_j$. Zde je pseudokód celé jedné iterace:

1. Do intervalových stromů přidáme všechna políčka, jejichž hodnota je v intervalu $\langle X_{i-1}, X_i)$.
2. Ze stromu S získáme hodnotu k pro obdélník patřící k X_i.
3. Ze stromu T získáme hodnotu $\sum_{j=1}^{k} p_j$ pro obdélník patřící k X_i.

A to je celé. Do každého intervalového stromu vložíme každou z N hodnot maximálně jednou, tedy dostáváme časovou složitost $\mathcal{O}(RS \cdot \log R \cdot \log S)$. Hodnoty výšek a průměrů máme setříděné a zpracováváme je ve vzestupném pořadí.

Posíláme gratulaci Michalu Punčochářovi, který tuto úlohu jako jediný vyřešil optimálně a zcela správně.

Karel Tesař

26-4-6 Sněhová bitva

Nejjednodušší řešení by bylo nepředpočítávat si vůbec nic. Při dotazu, zda na polopřímce $\overrightarrow{AB}$ je ještě nějaký další bod, jednoduše spočítáme směr z bodu A do všech ostatních a porovnáme se směrem do B. Jak spočítat a reprezentovat směr rozebereme níže, prozatím předpokládejme, že jde směr spočítat i porovnat v konstantním čase. Takové řešení má dotaz za $\mathcal{O}(n)$ a potřebuje $\mathcal{O}(n)$ paměti

[58] O intervalových stromech se můžete dočíst v naší kuchařce.

na uložení vstupu. Ale za takto jednoduchou úlohu by asi organizátoři 11 bodů nenabízeli.

Tak co bychom si mohli předpočítat? Co třeba odpověď na každý možný dotaz? Protože odpověď na takový dotaz je jen ANO/NE a počet dvojic je $\mathcal{O}(n^2)$, bude nám stačit dvourozměrné pole. Ale předpočítat si ho pomocí výše zmíněného jednoduchého řešení by trvalo $\mathcal{O}(n^3)$. To zkusíme zrychlit.

Zkusme si spočítat naráz všechny odpovědi pro jednoho mířícího bojovníka A a všechny možné cíle C. Tedy odpovědi pro všechny polopřímky $\overrightarrow{AX}$. Uděláme to tak, že spočítáme směry pro všechny cíle a tyto směry setřídíme. Tím se nám stejné směry „sesypou“ k sobě. Nyní můžeme směry ještě jednou projít a najít úseky stejných směrů a jim odpovídající cíle označit, že leží na společné polopřímce, a tedy že pro ně bude odpověď ANO. Toto nám zabere $\mathcal{O}(n \cdot \log n)$ pro jednoho mířícího bojovníka, tedy $\mathcal{O}(n^2 \cdot \log n)$ celkem. Spotřebujeme $\mathcal{O}(n^2)$ paměti a odpovídat můžeme v konstantním čase.

Nyní, co s tím směrem? Nejpřirozenější reprezentace by asi byla počítat úhel. To s sebou ale nese jisté technické problémy (jako například trochu pomalejší výpočet trigonometrických funkcí či chlupatá iracionální čísla, co se kamarádí s π i pro rozumná vstupní data). Budeme tedy počítat poměr rozdílu v ose X a Y. To má stále jisté problémy. Jednak tento poměr bude pro protilehlé směry vycházet stejně – směr doleva dolů a směr doprava nahoru bychom nerozlišili. Tedy, směr bude dvousložková hodnota. První složka bude říkat, jestli je to směr

nahoru, nebo dolů (resp. doleva, či doprava v případě vodorovných směrů, ať se vyhneme kladné a záporné nule). Druhým problémem je dělení nulou pro svislé směry. Ale v takovém případě si budeme nějakým způsobem reprezentovat nekonečno (např. čísla s plovoucí čárkou opravdu mají speciální hodnotu pro nekonečno).

Nakonec jedna poznámka o tom, že to může jít lépe. Pokud bychom použili hešování místo třídění pro shluknutí stejných směrů dohromady, dostali bychom se na očekávanou složitost $\mathcal{O}(n)$ (bohužel jen očekávanou, ne nejhorší případ) pro jednoho střelce, namísto $O(n \cdot \log n)$. To lze udělat proto, že nám vlastně vůbec nezáleží na pořadí, v jakém směry dostáváme, jen aby byly ty stejné pohromadě.

To má ale háček. Čísla s plovoucí čárkou mají v počítači tendenci akumulovat při operacích chybu. Proto je třeba je porovnávat s malou tolerancí, nikoliv na přesnou rovnost, a to s hešovací tabulkou nejde. Pokud bychom ale měli celočíselný (nebo alespoň racionální) vstup, můžeme směr ukládat jako zlomky a poté nám již hešovací tabulka pomůže. Ale převést zlomek na základní tvar nezvládneme v konstantním čase. Všimneme si však, že když jsou čísla na vstupu omezená hodnotou L, dva různé zlomky se budou lišit alespoň o $1/L^2$. Vynásobíme tedy všechny směry L^2 a budeme dělit celočíselně.

Program (Python): `http://ksp.mff.cuni.cz/viz/26-4-6.py`

Michal „vorner“ Vaner

26-4-7 Královští špioni

Rozplést síť špionů většinou nebývá snadné. V této úloze jsme ji ale měli již pěkně naservírovanou.

Nejjednodušším, avšak pomalým řešením je odsimulovat putování zpráv postupně od každého špiona. K tomu nám stačí si vstup načíst do pole A. Pak už jen v jedné proměnné udržujeme počet kroků a ve druhé pozici zprávy, tj. číslo špiona, u kterého se zrovna nachází. Jedno přeposlání zprávy provedeme snadno přiřazením `pozice = A[pozice]`.

Jak poznáme, kdy se má předávání zpráv zastavit? Jednoduše si budeme v dalším poli pamatovat, kteří špioni tuto zprávu viděli. Uvedené řešení má časovou složitost $\mathcal{O}(N^2)$, kde N je počet špionů. Za takové CodEx uděloval tři body.

Problém s pomalostí spočívá v tom, že procházíme v síti špionů stále dokola ta stejná místa, přestože výsledek je vždy stejný. Při každém průchodu můžeme rovnou uložit počty kroků pro všechny špiony, přes které jsme zprávu posílali.

Všimněme si, že každá zpráva končí své putování nějakým *cyklem* – částí, ve které si špioni posílají informace v kruhu. Zpráva vyslaná libovolným špionem na cyklu bude předaná přesně tolikrát, kolik je v daném cyklu špionů. V úseku před cyklem roste doba putování postupně o jedna za každého špiona.

Asymptoticky jsme si zatím nepomohli. Nic nám totiž nezaručuje, že špiony budeme procházet ve správném pořadí od toho nejvzdálenějšího. Musíme začít využívat to, co jsme již spočítali. Když při simulaci dojdeme ke špionovi, pro nějž máme výsledek spočítaný, nebudeme pokračovat dál, protože bychom se stejně nic nového nedozvěděli. K počtu kroků jen přičteme výsledek pro aktuálního špiona.

Tím už pro každého špiona provedeme jenom konstantně mnoho operací. Celková časová složitost tak je $\mathcal{O}(N)$. Pro úplnost ještě dodejme, že s použitý trikem se můžeme setkat častěji. Dokonce si vysloužil vlastní název *memoizace*.

Program (C): `http://ksp.mff.cuni.cz/viz/26-4-7.c`

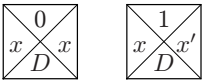
Jenda Hadrava

26-4-8 Dlaždičky

Úkol 1 – Počet jedniček dělitelný třemi

K řešení prvního úkolu nám bude stačit poměrně jednoduchá myšlenka. Budeme si zleva doprava propagovat dosavadní počet jedniček modulo 3 (úplně stejný trik bychom mohli použít pro libovolnou jinou konstantu).

Dlaždice si pořídíme dvojího typu, jedny budou mít nahoře nulu, druhé jedničku. Ty s nulou předávají stejnou hodnotu, tedy mají levou i pravou stranu obarvenou stejně. Naopak ty s jedničkou předávaný počet zvyšují, takže mají vlevo x a vpravo $(x+1)$ mod 3. Dole všechny dlaždice obarvíme barvou D. Naše dlaždice tedy vypadají takto:



pro $x' = (x+1) \bmod 3$

Dolnímu okraji přidělíme barvu D, levému i pravému okraji pak barvu 0.

Pro vstup, který má počet jedniček dělitelný třemi, určitě umíme dláždění vytvořit (přesně podle myšlenky řešení jen počítáme jedničky ve vstupu modulo 3).

Dláždění buď vůbec nelze vytvořit, nebo je určené jednoznačně. Pro daný vstup máme právě jednu možnost, jakou dlaždici přiložit přímo k levému okraji, pak právě jednu možnost, jakou dlaždicí navázat na tuto první atd. Dláždění tak vždy počítá jedničky modulo 3, tedy pokud dláždění existuje, vstup skutečně podmínku splňoval.

Všimněte si, že máme levý i pravý okraj zdi obarvený stejnou barvou. Tím jsme si ušetřili řešení okrajových případů. Chceme-li ovšem takový trik provést, musíme si velmi dobře rozmyslet, jestli si tím nepokazíme správnost. Obecně tím totiž tvrdíme, že zřetězíme-li několik vstupů splňujících naši podmínku, výsledek ji bude splňovat také.

Úkol 2 – Snižování výšky dláždění o konstantu

Představme si, že máme dláždění výšky t, a podívejme se na libovolný sloupec tohoto dláždění. Obarvení horizontálních stran dlaždic, vyjma horní stěny nejvyšší a spodní stěny nejnižší dlaždice, nemá pro zbytek dláždění žádný význam, určuje pouze návaznosti v tomto sloupci.

Naopak obarvení vertikálních stran důležité je, a to po celé výšce sloupce, určuje totiž návaznosti v rámci jednotlivých řádků. Toto obarvení tedy chceme zachovat.

Abychom snížili výšku dláždění, nahradíme každý sloupec jedinou dlaždicí. Očíslujme si dlaždice ve sloupci od 0 do $t-1$ (dlaždice v nultém řádku přiléhají ke vstupu), obarvení každé z nich pak (ℓ_i, h_i, p_i, d_i). Nová dlaždice pak bude nahoře obarvena barvou h_0, dole barvou d_{t-1}. Pro boční stěny zavedeme nové barvy, které označíme jako uspořádanou t-tici $(\ell_0, \ldots, \ell_{t-1})$, resp. $(p_0, \ldots, p_{t-1})$.

Pro úplnost dodejme, že levý okraj přebarvíme z barvy ℓ na $(\ell, \ldots, \ell)$, podobně pravý.

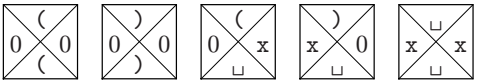
Sloupce k sobě přiléhají stále stejně, ovšem výšku dláždění jsme z t zmenšili na 1. Velikost množiny D, tedy množiny všech použitelných dlaždic, nepovažujeme za důležitou, přesto stojí za zmínku, že jsme ji tímto trikem exponenciálně zvětšili.

Úkol 3 – Dlaždičkové závorkování

Kdybychom chtěli správnost závorkování ověřovat přímočaře, mohli bychom postupně umazávat dvojice `()`, které určitě správně jsou. Místo opravdového smazání takových dvojic a následného posouvání zbytku řetězce se hodí „smazané“ závorky jen něčím nahradit a vstup nechávat na původním místě.

Potřebujeme tedy vždy nahradit dvě závorky `()`, mezi kterými buď není vůbec nic, nebo jsou pouze smazané znaky. Ostatní závorky musíme poslat o řádek níž, na konci chceme mít smazaný celý vstup. Toho můžeme dosáhnout například

s následující sadou dlaždiček:



Levý a pravý okraj obarvíme barvou 0, dolní barvou ␣.

V nejhorším případě budeme ovšem potřebovat $n/2$ kroků na smazání celého vstupu, program tedy pracuje v lineární časové složitosti. To přece musí jít lépe!

KSP

Závorkování rychleji

Abychom dosáhli lepší než lineární složitosti, vypůjčíme si trik z řešení prvního dílu seriálu. Budeme počítat úroveň zanoření, neboli kolik jsme zatím potkali neuzavřených závorek. A protože máme jen konečnou množinu barev, budeme si tuto úroveň ukládat jako číslo ve dvojkovém zápisu.

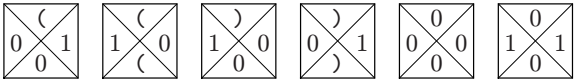
řešení

Úroveň zanoření může být u správného závorkování nejvýše $n/2$, na reprezentaci takového čísla ve dvojkovém zápisu potřebujeme $\log n$ bitů. My si dvojkové zápisy budeme ještě doplňovat tak, aby vždy začínaly nulou, ale tím jsme přidali jeden řádek. Náš program tak bude pracovat v logaritmické časové složitosti.

Vždy, když potkáme otvírací závorku, úroveň vnoření o 1 zvýšíme, u zavíracích závorek ji zase o 1 snížíme. Zápis úrovně nám bude vznikat směrem dolů, jinak řečeno, při hotovém dláždění ho můžeme číst na pravých stranách dlaždic v daném sloupci. Navíc si nepořídíme dlaždice, které by uměly reprezentovat odečítání do záporných čísel.

Horizontálně si budeme předávat hodnoty jednotlivých bitů čísla, vertikálně přenos mezi těmito bity. Ještě si dovolíme jeden podlý trik, a to ten, že vertikální přenosy budeme kódovat opět závorkami. Díky tomu totiž nemusíme nijak odlišovat přikládání dlaždic ke vstupu a k předchozímu řádku dláždění.

Pro úplnost dodejme, že všechny tři okraje zdi obarvíme 0, a pak se konečně pojďme podívat na používané dlaždice:



Úkol 4 – Minimální složitost závorkování

Zkusme si pořídit různé posloupnosti závorek délky n. Řekněme, že budeme posloupnosti budovat z dvojic závorek, že nejprve vybudujeme první polovinu této posloupnosti a že v první polovině smíme použít pouze dvojice `()` a `((`.

Takto jistě můžeme vytvořit $n/4$ posloupností délky $n/2$ takových, že žádné dvě nemají stejný počet otevřených závorek. Nyní každou z nich doplňme pomocí dvojic `()` a `))` na správně uzávorkovanou posloupnost délky n.

Pojďme se teď podívat na dláždění, která pro jednotlivé posloupnosti vzniknou. Na chvilku předpokládejme, že každé z nich má výšku právě V.

Pro $n/4$ prvních polovin máme $n/4$ druhých polovin, přičemž první polovině vždy odpovídá právě jedna ze druhých polovin. Na prostředním sloupci dláždění tedy musíme být schopní odlišit alespoň těchto $n/4$ různých možností, jinými slovy musí existovat $n/4$ možných obarvení prostředního sloupce.

Pokud označíme celkový počet barev v našem programu jako B, můžeme možná obarvení prostředního sloupce omezit také výrazem B^V. Tím dostáváme nerovnost $B^V \geq n/4$, úpravami získáme $V \geq \log_B n - \log_B 4$. To ovšem můžeme asymptoticky zapsat jako $V \in \Omega(\log n)$.

Rozhodnout správné závorkování tedy skutečně neumíme lépe než logaritmicky.

Pro úplnost ještě okomentujme náš drzý předpoklad, že dláždění pro každou z našich posloupností mělo výšku právě V. To samozřejmě nemusí být pravda, některá klidně mohla být jednořádková. Každé dláždění, které má výšku maximálně V ovšem umíme „nafouknout“ tak, aby mělo výšku právě V. Můžete si jako cvičení rozmyslet, jak toho dosáhnout.

Úkol 5 – Ekvivalence s Turingovými stroji

Začněme s asi snazším převodem, a to z dlaždičkových programů na Turingovy stroje. Necháme Turingův stroj simulovat stavbu dláždění, a to konkrétně po řádcích. Obarvení vodorovných hran budeme považovat za znaky na pásce vždy před provedením kroku a po něm, obarvení svislých hran pak za stavy.

Pro každou dlaždici (ℓ, h, p, d) zavedeme pravidlo typu $(\ell, h) \to (p, d, \to)$. Tedy pokud se ve stavu ℓ na pásce nachází znak h, stroj zapíše znak d a přejde do stavu p, hlava se posune doprava.

Tímto umíme vybudovat jeden řádek dláždění. Řádků ale může být víc, potřebujeme se tedy umět vracet zpět. V případě, že aktuální stav odpovídá barvě pravého okraje zdi, řekněme P, a na vstupu se nachází mezera, přejde stroj do speciálního návratového stavu. V něm znaky vždy zapisuje zpět a posouvá se doleva, dokud opět nenarazí na mezeru.

Návrat nastartuje pravidlo $(P, \sqcup) \to (N, \sqcup, \leftarrow)$, během vracení potřebujeme pravidla typu $(N, x) \to (N, x, \leftarrow)$ pro všechna možná obarvení x a k ukončení návratu přidáme ještě pravidlo $(N, \sqcup) \to (L, \sqcup, \to)$.

Ještě musíme poznat, že je celé dláždění dokončené. Mohli bychom při návratu odlišovat dva stavy: jeden, kdy jsme dosud ze vstupu četli pouze znak odpovídající obarvení dolního okraje, a druhý, kdy se alespoň jeden znak lišil. My si ale představíme, že ve speciálním stavu vybudujeme ještě jeden řádek navíc.

Po skončení návratu umožníme přechod do speciálního stavu, $(N, \sqcup) \to (*, \sqcup, \to)$. V tomto stavu přijímáme pouze správný znak (tedy správné obarvení),

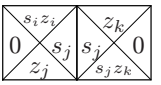
a pouze pokud dojdeme až na konec vstupu, ukončíme výpočet. Zavedeme proto ještě dvě pravidla, $(*, D) \to (*, D, \to)$ a $(*, \sqcup) \to$ ANO.

Už víme, že dlaždičkové programy nejsou silnější než Turingovy stroje. Pojďme teď převést Turingův stroj na dlaždičkový program.

Tentokrát bude jeden řádek dláždění odpovídat jednomu kroku Turingova stroje. Musíme vědět, nad kterým znakem se právě nachází hlava a v kterém stavu se nachází celý stroj. Zároveň nám informaci o stavu stačí znát u znaku pod hlavou, ostatní znaky stejně výpočet ovlivnit nemohou.

Pro každé pravidlo typu $(s_i, z_i) \to (s_j, z_j, \bullet)$ si pořídíme dlaždici vpravo.

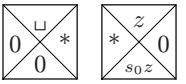
Dále si pořídíme dlaždice pro pravidla, kdy se hlava posouvá, pokud bychom u předchozího pravidla měli posun hlavy doprava, potřebujeme dvojici „slepených“ dlaždic níže vlevo.



Nesmíme zapomenout, že většinu znaků ze vstupu v jednotlivých krocích nijak neměníme, ale dlaždice pro ně existovat musí. Taková dlaždice je níže vpravo.

Koncového stavu dosáhneme při čtení nějakého konkrétního znaku, ale aby šlo dláždění dokončit, musí být správnou barvou obarvený celý poslední řádek. Pro jakékoli pravidlo typu $(s, z) \to$ ANO proto přidáme dlaždici (ANO, (s, z), ANO, ANO) a pro ostatní prvky abecedy dlaždice (ANO, z, ANO, ANO). Pouze pro okrajové mezery musí dlaždice vypadat jinak, (L, $\sqcup$, ANO, ANO), analogicky pro pravou stranu.

Zbývá si ještě rozmyslet, jak by měl vypadat začátek dláždění. Potřebujeme do vstupu dostat informaci o počátečním umístění hlavy a počátečním stavu. K tomu si zavedeme ještě jeden speciální inicializační stav. Pak nám budou stačit tyto dlaždice:



Tím je převod hotov, Turingův stroj umíme simulovat pomocí dlaždičkových programů, tedy tyto dva výpočetní modely jsou skutečně ekvivalentní.

Karolína „Karryanna“ Burešová

26-5-1 Made in China

První úloha byla trochu netradiční tím, že správné řešení má časovou i paměťovou složitost konstantní. Číslo relé totiž dostaneme tak, že spočítáme bitový XOR obou souřadnic (počítáme-li od nuly).

Pro jistotu si zopakujme, jak takový XOR funguje: Obě čísla zapíšeme pod sebe ve dvojkové soustavě a kdekoliv se pod sebou objevily dvě různé číslice, píšeme do výsledku jedničku, všude jinde nulu. Takže třeba

$$27 \text{ XOR } 17 = \texttt{11011} \text{ XOR } \texttt{10001} = \texttt{01010} = 10.$$

Zajímavější je, jak se na takovou věc přijde. Pokusíme se zde jeden z možných myšlenkových postupů nastínit.

Podívejme se na jednotlivé řádky trochu zvětšené tabulky ze zadání:

```
0 1 2 3 4 5 6 7
1 0 3 2 5 4 7 6
2 3 0 1 6 7 4 5
3 2 1 0 7 6 5 4
4 5 6 7 0 1 2 3
5 4 7 6 1 0 3 2
6 7 4 5 2 3 0 1
7 6 5 4 3 2 1 0
```


Nultý řádek odpovídá uspořádaným číslům. Na prvním jsou sousední čísla prohozená. Druhý prohazuje celé dvojice dohromady. Třetí kombinuje prohazováni prvního a druhého řádku. Čtvrtý vyměňuje celé čtveřice čísel. A tak dál.

Číslo řádku nám tedy přímo udává, jaký je vztah mezi číslem sloupce a samotným relé. Nejsnazší je se na číslo řádku podívat ve dvojkové soustavě. Každá jednička představuje jedno prohazování po blocích velikosti odpovídající příslušné mocnině dvojky. Například pátý řádek (ve dvojkové soustavě 101) prohazuje po blocích velikosti 4 a 1. Všechna čísla tedy budou posunuta nejdříve o 4 a následně o 1.

Chceme-li určit hodnotu na pozici $[r, s]$, stačí zjistit, na jaké pozici je stejné číslo na nultém řádku. Z pořadí řádku již víme, o kolik budeme posouvat. Směr poznáme po celočíselném vydělení čísla s daným posunem. Lichá čísla posuneme doleva, sudá doprava.

Když se trochu zamyslíme nad uvedeným dělením, zjistíme, že se jen díváme na hodnotu jednoho bitu v čísle s. Protože posouváme o mocniny dvojky, tak tím vždy měníme hodnotu jenom jediného bitu. Každému bitu z čísla s změníme hodnotu právě tehdy, když je daný bit jedničkový i v čísle r. Dostali jsme tak přímo slibovaný bitový XOR.

Jenda Hadrava

Kdyby vám uvedený postup přišel nedostatečně formální a chtěli jste důkladný důkaz, máte ho mít. Budeme postupovat indukcí a v i-tém kroku indukce dokážeme, že tabulka velikosti $2^i \times 2^i$ popisuje operaci XOR.

Pro $i = 0$, tedy tabulku 1×1, tvrzení evidentně platí.

Nyní krok od i k $i + 1$. Tabulku $2^{i+1} \times 2^{i+1}$ rozdělíme na čtyři podtabulky velikosti $2^i \times 2^i$. Budeme jim říkat LH, PH, LD, PD (levá horní atd.).

LH nezávisí na ostatních podtabulkách, takže podle indukčního přepokladu odpovídá XORu. Navíc si všimneme, že v každém řádku i každém sloupečku této podtabulky leží všechna čísla od 0 do $2^i - 1$. (Xorování čísel 0 až $2^i - 1$ libovolnou konstantou z téhož rozsahu musí tato čísla jenom přeházet po dvojicích.)

KSP

Nyní se zaměřme na podtabulku PH. Shora není ničím ovlivněna, zleva jsou tabulkou LH blokovaná všechna čísla od 0 do $2^i - 1$. Jsme tedy ve stejné situaci jako v LH, jenom je ke všem číslům přičteno 2^i. Analogicky v tabulce LD.

Zbývá podtabulka PD. Ta má od LD i od PH zablokovaná čísla 2^i až $2^{i+1} - 1$, ale žádná nižší, takže opět dopadne stejně jako LH.

řešení

Toto chování podtabulek přitom přesně odpovídá XORu: LH a PD mají nejvyšší bit obou souřadnic stejný, takže vypadají stejně jako XOR o bit kratších čísel; LD a PH ho mají různý, takže oproti XORu kratších čísel přibude ještě nejvyšší jedničkový bit, který odpovídá posunutí hodnot o 2^i.

Martin „Medvěd“ Mareš

26-5-2 Cesta pralesem

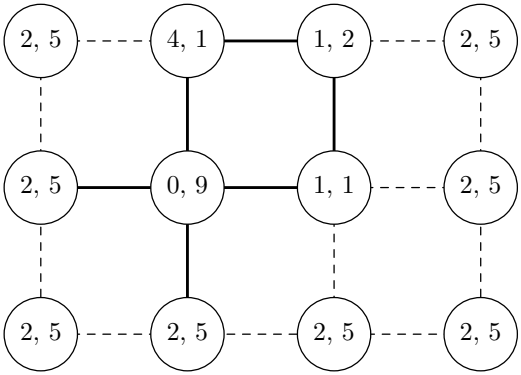
Úvodem řešení si jdu sypat popel na hlavu, neboť jsem jako autorka úlohy zapomněla ohlídat, že se v zadání objeví některé předpoklady.

Předně, koeficienty přehlédnutelnosti jsou vždy nezáporné (ale většina z vás naštěstí předpokládala, že až tak podlí organizátoři nejsou).

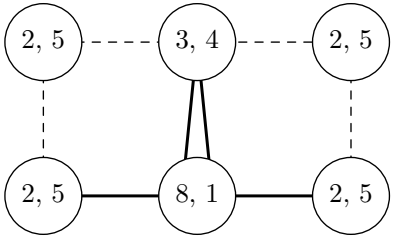
Zadruhé, v zadání mělo být asi jasněji ukázáno, že navzdory běžnému vnímání pojmu cesta v této úloze připouštíme cesty, ve kterých se opakují vrcholy, dokonce i takové, ve kterých se opakují (neorientované) hrany.

Omlouvám se všem, kterým opomenuté předpoklady způsobily komplikace při řešení.

Ukažme si ještě, že ačkoli je neopakování hran a vrcholů obvykle velice přirozené, v naší úloze se zopakování skutečně může vyplatit. Představme si situaci na následující straně (první číslo je přehlédnutelnost při průchodu rovně, druhé při odbočení).



Zatímco kdybychom odbočili rovnou, dostaneme přehlédnutelnost 9, při obejití políčka bude přehlédnutelnost pouze 4. To vysvětluje opakování vrcholů. A kdy se vyplatí zopakovat i hrany? Třeba v takovémto případě:



Přímý průchod má přehlédnutelnost 8, při „odskoku“ na jiné políčko dostaneme výslednou přehlédnutelnost 6.

Ale teď už honem na samotné řešení. Na hledání v grafech (a čtvercová mřížka je jen trochu speciální graf) se často vyplatí použít Dijkstrův algoritmus, který máme blíže popsaný v kuchařce o cestách.[59] Jenže použití tohoto algoritmu brání fakt, že ohodnocení závisí na tom, z kterého vrcholu jsme přišli.

Potřebujeme tedy vstup nejprve nějak upravit. Každý vrchol si rozčtvrtíme, jednotlivé čtvrtiny budou reprezentovat právě to, odkud jsme do daného vrcholu přišli. Všechny čtvrtiny pak pospojujeme s příslušnými čtvrtinami sousedních vrcholů, hrany ohodnotíme podle toho, zda odbočujeme, nebo procházíme rovně.

Pozor na to, že v této fázi musíme u každé čtvrtiny vytvořit také hranu odpovídající situaci, kdy se na dané křižovatce otočíme čelem vzad. Ohodnocení této hrany bude odpovídat koeficientu přehlédnutelnosti při odbočení.

[59] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`

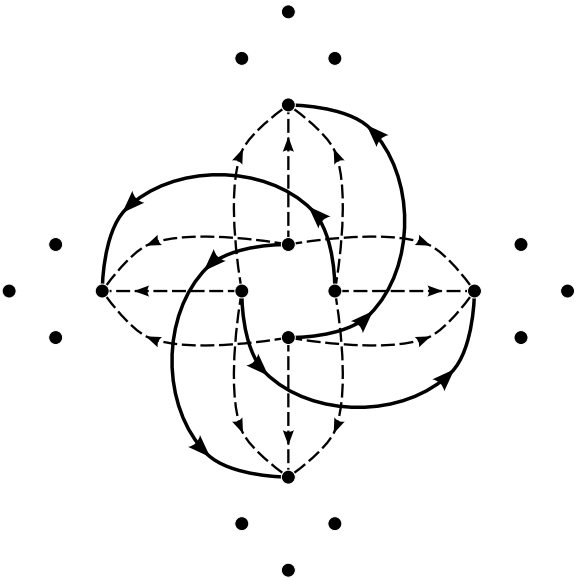
Ještě potřebujeme ošetřit počáteční a koncové políčko. Vytvoříme dva nové vrcholy. První z nich spojíme hranami ohodnocenými 0 se všemi čtvrtinami, do kterých se lze dostat z počátečního políčka, druhé spojíme se čtvrtinami políčka koncového.

V takto upraveném grafu (kde se na čtvrtiny díváme jako na plnohodnotné vrcholy) už jsou všechna ohodnocení jednoznačná, můžeme v něm tedy použít Dijkstrův algoritmus.

Na následujícím obrázku můžete sledovat, jak konstrukce grafu funguje. Nakreslili jsme hrany vedoucí ze čtvrtin jednoho vrcholu; plné hrany odpovídají chůzi rovně, čárkované zabočení.

Zbývá vyřešit složitost. Každý vrchol jsme nahradili čtyřmi, v novém grafu jich tedy máme konstanta-krát víc, podobně s hranami. Úpravu grafu bychom zvládli v lineárním čase, složitost Dijkstrova algoritmu je (při použití binární haldy) $\mathcal{O}((N + M) \log N)$, kde N označuje počet vrcholů a M počet hran.

Ve čtvercové mřížce máme hran $4N$ (a v našem upraveném grafu $16N$), celkovou složitost řešení tak můžeme odhadnout na $\mathcal{O}(N \log N)$. Paměťová složitost je lineární, $\mathcal{O}(N)$.



Program (C): `http://ksp.mff.cuni.cz/viz/26-5-2.c`

Karolína „Karryanna“ Burešová

26-5-3 Náhradní kabel

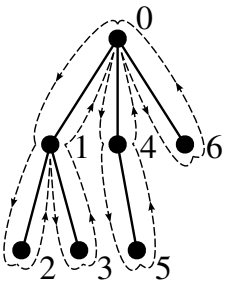
Přiznáváme, že úloha byla trochu složitější, než se mohlo podle deseti bodů zdát. Možná i proto dorazila pouze tři řešení. Trochu však mohla napovědět přiložená kuchařka o vyhledávání v textu.

Jak ale použít textové algoritmy k porovnávání stromů? Inu, budeme muset každý strom nějak popsat pomocí textového řetězce. Takovémuto popisu obvykle říkáme *kód* daného stromu. Různých kódování (tedy způsobů, jak stromům přiřazovat řetězce) můžeme vymyslet spousty. Na dvě z nich se teď podíváme.

Potřebujeme, aby v kódu bylo zachyceno pořadí synů. Toho můžeme dosáhnout tím, že projdeme strom do hloubky, v každém vrcholu vždy procházíme syny postupně od levého k pravému. Cestou si budeme zaznamenávat každý vrchol, kterým projdeme, a to i když už se do něj poněkolikáté vracíme. Pokud prohledávání zahájíme z jiného vrcholu, dostaneme kód, který je *rotací*[60] původního. To platí proto, že průchod je ekvivalentní „obejití stromu po obvodu“:

Řetězec reprezentující strom na obrázku by mohl vypadat následovně:

`0 1 2 1 3 1 0 4 5 4 0 6`.



Máme však problém. Co když nám někdo vrcholy přečísluje? V tu chvíli by se nám kódy porovnávaly dost špatně. Zkusme tedy vrcholy reprezentovat nějak jinak než jejich číslem. Můžeme použít například *stupeň*. To je číslo udávající počet hran, které z daného vrcholu vedou. Podívejme se opět na první strom:

`3 3 1 3 1 3 3 2 1 2 3 1`.

Musíme si ještě rozmyslet, že dva odlišné stromy nebudou mít nikdy stejný kód, jinými slovy, že z každého kódu dokážeme sestavit jednoznačně původní strom. To však naštěstí platí. Stačí si při vytváření stromu pamatovat, kolik hran jsme ve kterém vrcholu již použili, a tedy zda máme vytvářet nový vrchol, nebo se už vracíme. Zkuste si to nakreslit.

Pokud máme dva stromy se stejným počtem vrcholů, stačí nám každý z nich projít do hloubky a vygenerovat kód. Nyní jen ověříme, zda je jeden rotací druhého. To můžeme provést tak, že jeden kód napíšeme dvakrát za sebe a ve vzniklém řetězci se pokusíme najít ten druhý. Pokud hledání uspěje, představují oba stromy stejný kabel, jinak jsou různé. Pro samotné vyhledávání použijeme algoritmus KMP z kuchařky. Celé řešení tak pracuje v čase i prostoru $\mathcal{O}(N)$, tedy lineárně s počtem vrcholů.

Porovnání dvou kabelů pomocí KMP (C++):
`http://ksp.mff.cuni.cz/viz/26-5-3-kmp.cpp`

[60] Rotace znamená, že nějaký počet znaků přesuneme ze začátku na konec. Jednou z rotací řetězce `abcdef` je například `cdefab`.

Hledání dvojice kabelů

Teď umíme o dvou kabelech říct, zda jsou stejné. Co dělat, když máme více kabelů a cheme najít dvojici stejných? Mohli bychom použít algoritmus Aho-Corasickové k hledání kódů všech stromů najednou. Problém je v tom, že máme moc velkou abecedu – stupeň vrcholu může nabývat až $N - 1$ různých hodnot (představte si třeba kabely ve tvaru „hvězdiček“ s různým počtem ramen). A Aho-Corasicková (alespoň ve verzi z kuchařky) potřebuje abecedu konstantně velkou.

KSP

Slibovali jsme na začátku, že ukážeme více způsobů kódování stromu do řetězce. Teď je ta správná chvíle pro druhý z nich. Je vcelku jednoduchý. Strom projdeme opět stejným způsobem, ale tentokrát si budeme zaznamenávat průchod po hranách. Konkrétně při každém průchodu hranou zapíšeme, zda se pohybujeme dolů (`D`), tedy přicházíme do nového vrcholu, nebo nahoru (`N`), čili se vracíme z již prohledaného podstromu. Strom ze zadání tedy vytvoří kód

`D D N D N N D D N N D N`.

řešení

Kód nám přímo říká, jak máme strom kreslit. Nestane se nám proto, že by mu odpovídaly dva různé stromy. Je tu však jiný háček. Pokud začneme prohledávání z jiného vrcholu, dostaneme jiný kód.

Potřebovali bychom nějaký způsob volby počátečního vrcholu, který u stejných kabelů vybere stejný vrchol. Jednou z možností je najít takzvané *centrum stromu.* Získáme jej tak, že ze stromu postupně po krocích odebereme vždy všechny listy. Na konci nám zbude buď jeden jediný vrchol, nebo dva vrcholy spojené hranou. V prvním případě máme vyhráno. Ve druhém danou hranu *rozdělíme* – vytvoříme uprostřed nový umělý vrchol. Tento nový vrchol prohlásíme za centrum. Snadno si rozmyslíte, že shodné kabely (dokonce libovolné dva izomorfní stromy) opravdu mají stejné centrum.

Nejdříve si stromy rozdělíme do skupin podle počtu vrcholů. V každé skupině pak provedeme následující kroky:

1. Najdeme centrum každého stromu.
2. Každý strom projdeme do hloubky z jeho centra, a vytvoříme tak jeho kód.
3. Sestrojíme vyhledávácí automat Aho-Corasickové pro jehelníček tvořený kódy všech stromů. U každé jehly si navíc musíme poznamenat (v koncovém vrcholu), ze kterého stromu vznikla.
4. Pro každý ze stromů provedeme:
   - Napíšeme jeho kód dvakrát za sebe, a takto vzniklý řetězec použijeme jako seno, na které spustíme výše vytvořený automat.
   - Pokud automat najde výskyt nějaké jehly (kromě té, která odpovídá aktuálnímu stromu), znamená to, že aktuální kód je její rotací, tedy jsme našli dvojici stejných kabelů.

Všechny části stihneme opět v lineárním čase a zaberou jen lineárně prostoru, proto je i celý algortimus lineární s celkovým počtem vrcholů ve všech stromech dohromady.

Hledání dvojice kabelů (C++):
`http://ksp.mff.cuni.cz/viz/26-5-3-ac.cpp`

Karolína „Karryanna“ Burešová & Jenda Hadrava

26-5-4 Rozdělování jídla

Zadnání si můžeme přeformulovat také tak, že chceme vybrat nějakou množinu receptů tak, aby dohromady spotřebovaly surovin co nejvíc, ale zároveň nepřesáhly určitou mez (dostupné množství). To se náramně podobá *problému batohu* z kuchařky o dynamickém programování.[61]

Přesněji pro[62] $K = 1$ jde přímo o problém batohu, jehož řešení je popsáno v kuchařce, pro vyšší K o jakousi „vícerozměrnou“ verzi.

Tu vyřešíme analogicky: pořídíme si K-rozměrné pole, kde bude na pozici $(j_1, \ldots, j_K)$ nenulová hodnota právě tehdy, když existuje podmnožina receptů, které dohromady spotřebují j_1 první suroviny, j_2 druhé, … Pole naplňujeme stejně jako u jednorozměrné verze: postupně procházíme předměty (recepty) a zkoušíme je všemi možnými způsoby do batohu přidat.

Alternativně se dá na recepty dívat jako vektory, které klasicky vektorově sčítáme a snažíme se je naskládat do batohu o kapacitě $(\vec{m})$. Naše vícerozměrné pole pak můžeme chápat jako pole indexované vektory.

Kuchařkové řešení

Při psaní tohoto řešení jsem nejdřív zkusil naprogramovat verzi přesně podle kuchařky (tedy opakované procházení celým polem odzadu a postupné zjišťování, jakých všech možných zaplnění batohu lze dosáhnout). Ve vícerozměrném poli „procházení odzadu“ odpovídá průchod do šířky z posledního („pravého dolního“) políčka. Později si ukážeme, jak takový průchod dělat efektivně.

Z důvodu šetření pamětí neukládáme do pole čísla receptů jako v kuchařce, nýbrž jen jedničky a nuly, podle toho, jestli extistuje tak velká množina receptů. Tím přijdeme o možnost zjistit, z jakých receptů se množina skládá, ale ježto nás zajímá jen celkové množství spotřebovaných surovin, nevadí to.

Leč takovéto řešení úspěšně vyřešilo jen jeden vstup (konkrétně osmý).

Problém je v tom, že kvůli průchodu do šířky čteme napřeskáčku z různých řádků vícerozměrného pole, a tedy z různých míst v paměti. A zatímco při

[61] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

[62] Připomeňme značení: K je počet surovin, N počet receptů, m_i dostupné množství i-té suroviny a pro daný recept je a_i množství i-té suroviny spotřebované tímto receptem, všechna čísla celá.

teoretických úvahách pro zjednodušení předpokládáme, že všechny přístupy do pole trvají stejně (konstantně) dlouho, u opravdového počítače tomu tak není. Ukazuje se, že číst pole sekvenčně od začátku do konce (u vícerozměrného po řádcích) je výrazně rychlejší než neuspořádaně a napřeskáčku.

Pokud se trochu zajímáte o fungování počítačů, můžeme prozradit, že za to mohou přinejmenším dva jevy: neefektivní využití *keší procesoru* (z každého načteného kešového řádku obvykle použijeme jen jedno číslo) a *prefetchování*. O obojím se můžete dozvědět např. v Medvědově textu o programování s ohledem na hardware.[63]

KSP

Kuchařkové řešení „odpředu“

Proč vlastně procházíme pole odzadu? Představte si například, že máme jednorozměrnou verzi a jediný předmět o váze 2. Vlevo je počáteční stav pole.

```
i     0 1 2 3 4 5          i     0 1 2 3 4 5
P[i]  1 0 0 0 0 0.         P[i]  1 0 1 0 1 0,
```

řešení

Pokud ho budeme procházet odzadu, dostaneme správný výsledek. Ale při průchodu odpředu získáme stav pole vpravo výše.

tedy stejný předmět jsme vložili do batohu několikrát. Tomu ale můžeme snadno zabránit tím, že si budeme do políček místo jedniček zapisovat číslo kroku, ve kterém byla vytvořena (stejně jako to dělá originální kuchařkové řešení, i když z jiných důvodů), a políčka vytvořená v aktuálním kroku ignorovat. Pak už můžeme s klidem zvolit průchod hezky po řádcích z levého horního rohu.

To ovšem zvýší paměťové nároky, neb si v poli musíme pamatovat hodnoty z rozsahu $0 \ldots N$, tedy pro větší N nám nebude stačit jeden bajt na položku. A překvapilo mě, jak to bylo obtížné nepřekročit paměťové limity nastavené v CodExu. Nakonec jsem zjistil, že pro dodržení limitů se nedalo naprogramovat jedno univerzální řešení, ale muselo se rozlišit několik případů podle maximální hodnoty N, a pro každý zvolit jinak velký typ položek v matici (pro $N \leq 255$ stačí jednobajtový, pro větší dvoubajtový). Ve zdrojáku je toto pro přehlednost jen zmíněno v komentáři.

Takovéto řešení už selhalo jen na třech vstupech (vypršel časový limit).

Problém obou algoritmů je také v tom, že receptů je málo, a tudíž hlavní pole je zaplněné velmi řídce (na drtivé většině míst má nuly). A náš algoritmus stráví spoustu času vytrvalým procházením tohoto moře nul, aby mezi nimi našel tu a tam nějakou nenulovou hodnotu.

Tomuto by velice pomohlo, kdybychom si pamatovali, kam nejdál jsme se v poli zatím dostali. Ale to jsem již ani nezkoušel testovat a programovat. Za chvíli si ukážeme ještě lepší řešení – ale nejdříve slibované odbočka o průchodu do šířky.

[63] `http://mj.ucw.cz/papers/hwopt.pdf`

Průhod do šířky

Snadno si povšimneme, že při prohledávání dvourozměrné mřížky do šířky z jednoho jejího rohu projdeme v i-té fázi právě i-tou diagonálu v pořadí od tohoto rohu. Dobře je to vidět, když si do matice napíšeme vzdálenosti jednotlivých políček od pravého dolního rohu:

```
6 5 4 3
5 4 3 2
4 3 2 1
3 2 1 0
```

Takže vůbec nepotřebujeme frontu, ale průchod ve správném pořadí zajistíme vhodným dopočítáváním souřadnic na diagonálách (vizte zdroják).

Pro více rozměrů je dobrý mezikrok dívat se, jak by to dopadlo pro krychli. Tam se to dá představit tak, že postupně ukrajujeme z rohu.

Nyní bychom rádi tento postup zobecnili do více dimenzí. Podívejme se například na posloupnost souřadnic, které projdeme ve dvourozměrné tabulce výše:

```
0. vrstva: (4,4)
1. vrstva: (4,3) (3,4)
2. vrstva: (4,2) (3,3) (2,4)
...
```

Snadno si všimneme, že každou vrstvu tvoří políčka se stejným součtem souřadnic (ba dokonce všechna s takovým součtem), a tyto součty se postupně o jedničku snižují.

To není žádné velké překvapení. Každé políčko se do i-té vrstvy dostane tak, že je v nějaké souřadnici „levým sousedem“ některého políčka v $(i-1)$-ní vrstvě, tedy příslušnou souřadnici má o jedničku menší. A pokud má právě jednu souřadnici o jedničku menší, pak i součet souřadnic je pochopitelně o jedničku menší.

No a najít k-tice čísel s daným součtem už je snadné programátorské cvičení.

Řešení se spojákem

Problému řídkého pole se můžeme vyhnout tak, že si budeme udržovat spojový seznam nenulových políček. Pak stačí procházet tento seznam místo toho, abychom je v poli dlouho hledali.

Mírnou nevýhodu může představovat, že spoják zabere nějakou paměť navíc. Ale vzhledem k tomu, že matice je zaplněná řídce, moc velký nebude. Navíc nyní nám opět stačí ukládat si do matice jen jedničky a nuly,[64] pročež nám opět stačí jen jeden bajt na položku v hlavním poli (stačil by i jeden bit, ale tím nebudeme řešení zbytečně komplikovat).

Závěr

Doufám, že jste si z této úlohy odnesli to, že asymptotická složitost není vždy ekvivalentní s tím, jak rychle program poběží. A že více implementací ve stejném jazyce a se stejnou složitostí se může výrazně lišit dobou běhu.

[64] Problému vícenásobného přidávání se můžeme vyhnout třeba tak, že budeme nová políčka připojovat na začátek spojáku. Nebo si je budeme skládat do pomocného spojáku, který k hlavnímu připojíme až na konci každého kroku.

Pokud máte k něčemu z řešení (případně k implementaci) dotaz, nebojte se zeptat na fóru, rádi odpovíme.

Verze bez spojáku od konce (C):
`http://ksp.mff.cuni.cz/viz/26-5-4-konec.c`

Verze bez spojáku od začátku (C):
`http://ksp.mff.cuni.cz/viz/26-5-4-zacatek.c`

Verze se spojákem (C):
`http://ksp.mff.cuni.cz/viz/26-5-4-spojak.c`

KSP

Vojta Sejkora

26-5-5 Průjezd tunelem

řešení

V řešení budeme značit W šířku a H výšku obdélníka.

Hodně z vás v mnohoúhelníku hledalo první a poslední místo, kde je mnohoúhelník vysoký alespoň H, a pak zkontrolovalo, jestli jsou tato dvě místa vzdálená alespoň W. Takové řešení však nefunguje. Nic nám nezaručuje, že obě nalezená místa se nachází ve stejné výšce. Dobrým protipříkladem je například kosodélník.

My si ukážeme dva možné přístupy k řešení. Jeden je založen na technice zametání roviny přímkou,[65] kdy mnohoúhelník budeme zametat zároveň dvěma svislými přímkami vzdálenými W. Ve druhém řešení více využijeme vlastností konvexnosti a získáme všechna možná řešení pronikáním různě posunutých původních mnohoúhelníků. Obě řešení budou pracovat v čase $\mathcal{O}(n)$, kde n je počet bodů mnohoúhelníka.

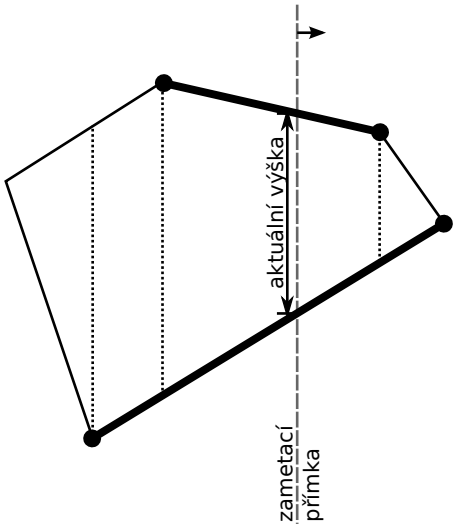
První způsob řešení si pro jednoduchost popíšeme nejdříve jen pro hledání svislé úsečky dlouhé H. Na to nám bude stačit jedna zametací přímka.

Zametání přímkou je poměrně obecná technika pro řešení širokého spektra geometrických úloh. Spočívá v tom, že si představujeme přímku pohybující se spojitě napříč geometrickou scénu, která sleduje, co se děje „pod ní“ a umí „ohlásit“, když narazí na něco pro nás zajímavého.

V našem případě zametáme svislou přímkou, která se posouvá postupně přes mnohoúhelník od jeho nejlevějšího bodu po nejpravější. Pro každou polohu zametací přímky si (myšleně) spočítáme, jak vysoký je v daném místě mnohoúhelník. Pokud zametací přímka projde místem s výškou alespoň H, ohlásí na výstup příslušnou x-ovou souřadnici (stačí jednou).

Tohle je dobrý způsob, jak si řešení představit, ale určitě jej nemůžeme takto naprogramovat, neb bychom museli počítat výšku mnohoúhelníka pro každou z nekonečně mnoha x-ových pozic, kterými přímka prochází.

[65] anglicky *plane sweeping* či *sweep line*



Všimneme si, že v každý okamžik (až na konečně mnoho výjimek) protíná zametací přímka právě dvě strany mnohoúhelníka. Navíc to, které dvě to jsou, se mění pouze, když přímka projde nějakým vrcholem mnohoúhelníka. Můžeme tedy podle x-ových souřadnic vrcholů rozdělit mnohoúhelník na svislé pásy, v každém z kterýchž protíná zametací přímka vždy nějakou pevnou dvojici stran mnohoúhelníka.

V rámci jednotlivého pásu už snadno určíme místo s výškou alespoň H, aniž bychom museli zkoušet všechny možnosti. K tomu se nám bude hodit malá odbočka do analytické geometrie.

Máme-li dané dva body $A = [x_A, y_A], B = [x_B, y_B]$, pak všechny body na úsečce AB se dají vyjádřit soustavou rovnic:

$$\begin{aligned} x &= x_A + t(x_B - x_A) \\ y &= y_A + t(y_B - y_A) \\ t &\in \langle 0, 1 \rangle \end{aligned}$$

Tedy po dosazení libovolného t z intervalu $\langle 0, 1 \rangle$ dostaneme souřadnice bodu ležícího na úsečce AB. Mějme nyní spodní úsečku AB a horní úsečku CD, chceme najít souřadnici x, kde jsou úsečky od sebe svisle vzdálené alespoň H. Tu získáme vyřešením následující soustavy (ne)rovnic:

$$\begin{aligned} x_A + t(x_B - x_A) &= x_C + s(x_D - X_C) \\ y_A + t(y_B - y_A) &\leq y_C + s(y_D - y_C) - H \end{aligned}$$

Neznámými jsou proměnné s a t. Pokud získáme řešení, kde $s, t \in \langle 0, 1 \rangle$, tak jsme našli místo, kam se nám vejde svislá úsečka délky H.

Implementace je nyní už jednoduchá. Obvod mnohoúhelníka si rozdělíme na horní a spodní polovinu – tedy části, které vedou od nejlevějšího bodu k nejpravějšímu „horem“, resp. „spodem“. Všimněte si, že pokud existuje víc nejlevějších (nejpravějších) bodů, je jedno, který si vybereme.

Nyní bychom chtěli postupně zleva doprava projít všechny pásy a pro každý z nich provést nás výpočet se správnou dvojicí úseček. To uděláme tak, že si budeme průběžně udržovat, která úsečka je aktuálně horní a spodní. Na začátku jsou to strany sousedící s nejlevějším vrcholem mnohoúhelníka. Poté postupně procházíme všechny zbylé body mnohoúhelníka v pořadí dle x-ové souřadnice a s každým vyměníme buď horní, nebo dolní úsečku za jinou, čímž přejdeme do sousedního pásu.

Jelikož úseček je celkem n a každou vyměníme maximálně jednou, tak celý algoritmus má časovou složitost $\mathcal{O}(n)$.

Nyní řešení upravíme pro hledání obdélníka. Na to nám jen jedna zametací přímka stačit nebude, ale budeme potřebovat dvě od sebe vzdálené W. Pro obě přímky si budeme udržovat horní a spodní úsečku, kterou zrovna prochází, a řešení budeme přepočítávat vždy, když některá z přímek projde vrcholem mnohoúhelníka.

A jak se bude lišit výpočet? Potřebujeme najít místo, kde vzdálenost výše postavené spodní úsečky a níže postavené horní úsečky je alespoň H. To spočítáme tak, že pravou horní a spodní úsečku posuneme o W doleva. Jednoduchými lineárními nerovnicemi zjistíme, pro jaké intervaly je která horní (resp. spodní) úsečka níže (resp. výše).

Tím se nám řešení rozpadne nejvýše na tři intervaly, protože v každé dvojici se maximálně jednou může změnit, která úsečka je horní (resp. spodní). Pak jen pro každý takový interval použijeme původní výpočet pro svislou úsečku o délce H. Časová složitost tohoto řešení je také $\mathcal{O}(n)$, protože pro obě zametací přímky uděláme maximálně n výměn úseček a mezi dvěma výměnami provádíme jen konstatně dlouhý výpočet.

Druhý způsob řešení zde pouze naznačíme. Každý konvexní útvar má tu vlastnost, že pokud v něm leží body A a B, tak pak v něm leží i celá úsečka AB. Speciálně pokud v mnohoúhelníku najdeme umístění všech rohů obdélníka, tak v něm leží i celý obdélník.

Opět se nejdříve podíváme, jak najít svislou úsečku délky H. Taková úsečka může mít svůj spodní konec ve všech bodech, které splňují, že bod o H nad nimi je také uvnitř mnohoúhelníka. Takže všechny takové body získáme tak, že mnohoúhelník posuneme o H nahoru a určíme jeho průnik s původním mnohoúhelníkem.

Průnik konvexních mnohoúhelníků je opět konvexní mnohoúhelník. Ten teď vezememe posuneme jej o W doleva a znovu nalezneme průnik. Tím dostaneme

konečný mnohoúhelník, který obsahuje právě všechny body, kde obdélník může mít svůj levý horní roh.

Jak provedeme onen průnik mnohoúhelníků? Ukážeme si to pro první průnik. Ve druhém případě nám jen stačí situaci otočit o 90 stupňů a výpočet zopakovat. Jelikož jsou oba mnohoúhelníky shodné, tak nám jen stačí najít první a poslední průsečík horní části spodního mnohoúhelníka a spodní části horního mnohoúhelníka. Pak body mezi těmito průsečíky tvoří obvod výsledného průniku.

Na hledání příslušných průsečíků opět použijeme zametání přímkou a dostaneme řešení fungující v čase $\mathcal{O}(n)$.

Karel Tesař

26-5-6 Nejvyšší stavby

„Hledání maxim ve 2D nebo dokonce jen 1D oblastech matice? To jsou jasný haldy," řekne si kdekterý KSPák. Jenže v optimálním lineárním řešení se haldy nepoužívají, jen by ho zdržovaly. Pomocí hald či binárních výhledávacích stromů lze dosáhnout časové složitosti $\mathcal{O}(RS(\log r + \log s))$, což ponecháme jako cvičení na práci s těmito strukturami. Lineární řešení však nevyžaduje kromě pár triků žádnou složitější datovou strukturu než spojový seznam.

Lehčí varianta fungovala jako nápověda, proto si pojďme nejprve vyřešit v lineárním čase 1D oblasti. Pro $R = r = 1$ máme tedy posloupnost S čísel a chceme zjistit nejvyšší číslo v každém souvislém úseku délky s. V prvním, nejlevějším úseku najdeme nejvyšší číslo snadno tím, že projdeme všechna čísla úseku. Poté budeme posouvat aktuální úsek o jedno číslo doprava.

Když posuneme úsek o jedna doprava, jedno číslo nám vypadne a jedno přibude. Pokud je nové číslo vyšší než dosavadní nejvyšší číslo úseku, bude i nejvyšším číslem nového úseku, jinak jím zůstane původní číslo. Problém nastane, když dosavadní nejvyšší číslo z posloupnosti vypadne. V tom případě by ho mělo nahradit druhé nejvyšší číslo, jež však také musíme udržovat. Kdyby však vypadlo i toto náhradní číslo, tak musíme mít ještě dalšího náhradníka, uvažte třeba klesající posloupnost.

Proto si budeme udržovat seznam kandidátů na nejvyšší číslo, reprezentovaný obousměrným spojovým seznamem. Prvním kandidátem bude samo nejvyšší číslo úseku, druhým bude nejvyšší číslo *napravo* od nejvyššího čísla úseku, třetím nejvyšší číslo napravo od druhého kandidáta, … Kandidáti budou tedy tvořit nerostoucí podposloupnost aktuálního úseku. U kandidátů si navíc budeme pamatovat jejich pozici v posloupnosti, abychom věděli, kdy vypadnou ze seznamu.

Při každém posunutí úseku o jedna doprava nám z kandidátů může vypadnout jen první, ten největší, a v tom případě se nejvyšší číslo úseku přesouvá na druhého kandidáta. Nové číslo, jež přibylo do úseku, přidáme do seznamu kandidátů. Navíc z tohoto seznamu od konce smažeme všechny kandidáty menší

nebo rovné přidanému číslu, díky čemuž bude seznam kandidátů klesající podposloupnost úseku. Poslední číslo aktuálního úseku tak vždy bude kandidátem.

Jaká je časová složitost tohoto postupu? Nejlevější úsek a seznam kandidátů inicializujeme jistě v $\mathcal{O}(s)$, nicméně i posun úseku může zabrat až $\mathcal{O}(s)$, neboť kandidátů je až s a my je při jednom posunu můžeme všechny smazat. To se však stane jen relativně málo často.

K důkazu, že tento algoritmus je lineární, nám pomůže amortizovaná složitost.[66] Každé číslo posloupnosti totiž do seznamu kandidátů jednou přidáme a maximálně jednou ho smažeme, čili celkový počet operací vyjde na $\mathcal{O}(S)$. Dodatečné paměti (bez vstupní posloupnosti) použijeme $\mathcal{O}(s)$, neboť kromě seznamu kandidátů a jejich pozic si už nepotřebujeme nic ukládat.

Nyní zobecníme řešení pro těžší, dvourozměrnou variantu. Výše představený postup pro jeden řádek budeme dělat najednou pro všechny řádky. Nejprve zpracujeme nejlevější úseky všech řádek a získáme sloupec maxim těchto úseků, na němž provedeme stejný postup jako na řádku, akorát s parametry R a r místo S a s. Tímto získáme maxima v obdélnících sousedících s levým okrajem, tedy první sloupec výsledné tabulky.

Pak posuneme úseky v každém řádku o jedna a opět dostaneme sloupec maxim, na kterém provedeme algoritmus pro jednorozměrnou variantu. Toto opakujeme, dokud nedojdeme s řádkovými úseky k pravému okraji. Algoritmus vrátí správné řešení, neboť maximum na obdélníku počítáme jako maximum z maxim na příslušných úsecích v jednotlivých řádcích.

Posouvání úseků na každém řádku zabere $\mathcal{O}(S)$, čili celkově $\mathcal{O}(RS)$. Jeden sloupec maxim z aktuálních úseků řádků zpracujeme v $\mathcal{O}(R)$, což dává opět $\mathcal{O}(RS)$, takže algoritmus je lineární s velikostí vstupu. Kromě vstupní matice čísel si stačí pamatovat kandidáty pro každý řádek a pro aktuálně zpracovávaný sloupec, tedy $\mathcal{O}(Rs + r)$ čísel.

Program (C): `http://ksp.mff.cuni.cz/viz/26-5-6.c`

Pavel „Paulie“ Veselý

Medvědí poznámka: Existuje ještě jiný, podobně elegantní způsob, jak vyřešit jednorozměrnou verzi. Posloupnost rozdělíme na bloky délky s a pro každý z nich předpočítáme prefixová minima (tedy minima od začátku bloku do každého jeho prvku) a podobně suffixová minima (od prvku do konce bloku). Pak si všimneme, že každý úsek délky s je buďto blok, nebo ho lze složit ze suffixu jednoho bloku a prefixu následujícího bloku. Stačí tedy v konstantním čase zkombinovat nejvýše dvě přepočítaná minima. Pokud bychom z tohoto algoritmu odvodili 2D řešení, bylo by stejně rychlé, ale potřebovalo by paměť na pomocnou matici.

[66] `http://ksp.mff.cuni.cz/viz/kucharky/minimalni-kostra`

26-5-7 Partie piškvorek

Řešení vybudujeme postupně: začneme absurdně zjednodušenou verzí úlohy a postupně se budeme všech omezení zbavovat. Hracímu plánu budeme říkat *mřížka,* křížkům a kolečkům *symboly* a souvislým úsekům stejných symbolů *linie.* Linii nejde ani jedním směrem rozšířit, z obou stran je tedy ohraničena mezerou, opačným symbolem, případně okrajem mřížky.

Jeden rozměr, jeden symbol, jedna operace

Prozkoumejme nejprve úlohu, v níž má mřížka jediný řádek o n políčkách, pokládáme jenom jeden druh symbolů (třeba křížky) a navíc je nikdy nemažeme.

Postačí udržovat si pro každé políčko mřížky, zda na něm nějaká linie začíná nebo končí, a pokud ano, tak kde leží její opačný konec.

Na počátku výpočtu žádné linie neexistují. Kdykoliv přidáme křížek, podíváme se, zda nějaká linie končí těsně před ním nebo začíná těsně za ním. Rozlišíme čtyři případy:

- Nenastane ani jedno: tehdy zakládáme novou linii o jednom křížku a k aktuálnímu políčku napíšeme, že na něm tato linie začíná i končí.
- Před námi končí linie, za námi žádná nezačíná: tehdy prodlužujeme linii před námi o jedno políčko. Posuneme koncovou značku a u té počáteční přepíšeme informaci o konci.
- Symetrický případ, kdy za námi linie začíná, ale před námi nekončí, ošetříme obdobně.
- Zbývá případ, kdy nastane obojí současně, čili naše políčko propojuje dvě linie. Tehdy zrušíme konec levé a začátek pravé, načež začátek levé necháme ukazovat na konec pravé a naopak.

Navíc se pokaždé podíváme, zda nově vzniklá linie není delší než dosavadní maximum. To vše zvládneme v konstantním čase, navíc ale musíme připočíst lineární čas na inicializaci pole začátků a konců.

Přidáváme mazání

Nyní strukturu vylepšíme, aby uměla křížky mazat. Opět mohou nastat čtyři případy: buďto byl smazaný křížek osamocený (tehdy prostě zrušíme celou linii), nebo leží na začátku či na konci linie (to poznáme podle značek začátků a konců a linii prostě zkrátíme), nebo leží uvnitř linie. Tehdy potřebujeme najít její začátek a konec a linii rozdělit na dvě.

První tři případy rozpoznáme pomocí značek na aktuálním políčku a jeho sousedech. Pokud ale políčko leží kdesi uvnitř dlouhé linie, potřebujeme rychle zjistit, kde leží nejbližší značka.

Pořídíme si tedy navíc vyhledávací strom, v němž si budeme pamatovat pozice všech značek začátku. Kdykoliv mažeme nějaký křížek, zeptáme se stromu, jaká je nejbližší nižší pozice začátku. To strom zvládne v logaritmickém čase a

jakmile známe polohu začátku, příslušná značka nám prozradí, kde leží konec. Navíc musíme strom aktualizovat, kdykoliv se nějaký interval změní. To naštěstí nastane $\mathcal{O}(1)$-krát za operaci, takže to celkově trvá $\mathcal{O}(\log n)$.

Ještě se nám ovšem zkomplikovalo udržování nejdelší linie. Už totiž není pravda, že by se nejdelší linie stále jen prodlužovala. Pořídíme si proto další vyhledávací strom, do kterého budeme ukládat délky všech existujících linií. Jelikož se délky linií mohou opakovat, budeme si u každé délky pamatovat počítadlo, abychom věděli, kdy ji smazat. Každá operace s intervaly opět způsobí $\mathcal{O}(1)$ změn stromu, které potrvají $\mathcal{O}(\log n)$. Na konci operace se pak stačí zeptat tohoto stromu (budeme mu říkat *souhrnný strom*) na aktuální maximum.

Souhrnný strom bychom navíc mohli snadno upravit, aby si pamatoval nejen délky linií, ale i jejich polohy. Pak bychom věděli nejen jak dlouhá je maximální linie, ale také kde přesně leží.

Jedno umístění nebo mazání křížku nám tedy trvá $\mathcal{O}(\log n)$ a navíc potřebujeme čas $\mathcal{O}(n)$ na inicializaci struktury.

Jednorozměrná verze s křížky a kolečky

Teď je na čase si přiznat, že v piškvorkách hrají nejen křížky, ale také kolečka. To nám úlohu zkomplikuje jen kosmeticky: pořídíme si dvě datové struktury, jedna si bude pamatovat linie křížků, druhá linie koleček. A necháme je pracovat se společným souhrnným stromem, takže rovnou dostaneme maximum z křížků a koleček. Časová složitost se asymptoticky nezměnila.

Dvojrozměrná verze

Opravdová piškvorkovnice je ovšem dvojrozměrná, řekněme $n \times n$. Poradíme si jednoduše: pořídíme si samostatnou jednorozměrnou strukturu pro každý řádek, každý sloupeček i každou úhlopříčku a kdykoliv umístíme nebo smažeme symbol, řekneme o tom všem čtyřem strukturám, ve kterých dané políčko leží. Opět všechny struktury necháme pracovat se společným souhrnným stromem, takže nám budou udržovat globální maximum.

Časová složitost na operaci zůstává $\mathcal{O}(\log n)$, protože pokaždé přepočítáváme $\mathcal{O}(1)$ struktur, z nichž každá pracuje v logaritmickém čase. Na začátku výpočtu inicializujeme řádově n struktur o n prvcích, což dohromady trvá $\mathcal{O}(n^2)$.

Program (Python): `http://ksp.mff.cuni.cz/viz/26-5-7.py`

Malé kouzlo na závěr

Celé řešení je ještě možné zrychlit. Stačí si všimnout, že do vyhledávacích stromů ukládáme pouze čísla od 1 do n. Můžeme proto místo klasických stromů použít některou ze speciálních celočíselných datových struktur, například van Emde-Boasovy stromy. Jejich popis najdete ve skriptíčkách Krajinou grafových algoritmů.[67] Zde prozradíme pouze to, že pro hodnoty od 1 do U jim

[67] `http://mj.ucw.cz/vyuka/ga/`

jedna operace trvá $\mathcal{O}(\log \log U)$, takže se naše piškvorková struktura zrychlí na $\mathcal{O}(\log \log n)$ na operaci plus $\mathcal{O}(n^2)$ na inicializaci.

Martin „Medvěd“ Mareš

26-5-8 Automatizovaný graf

Část z vás se automatů ve vrcholech grafu zalekla a nepouštěla se dále než za první nebo druhý úkol. Ale jsem velmi rád, že se mezi vámi našlo i dost odvážlivců, kteří se rozhodli poprat se i se zbylými úkoly.

Úkol 1

Tento úkol byl velmi podobný druhému ukázkovému příkladu – vyslat dva signály proti sobě a tam, kde se potkají, označit vrchol. Abychom ale označili vrchol ve třetině kružnice, musel by jeden ze signálů běžet dvakrát tak rychle než druhý.

Zrychlit signál neumíme (musel by přeskakovat najednou přes dvě hrany, což ale nejde, protože každý automat ve vrcholu vidí pouze do svých bezprostředních sousedů), ale umíme jeden ze signálů zpomalit na polovinu a tím zařídit stejný efekt. Zpomalení na polovinu uděláme tak, že signál v každém vrcholu na jeden takt pozdržíme a teprva poté pošleme dál.

Abychom označili vrcholy ve třetině a dvou třetinách, budeme muset na začátku vyslat čtyři signály. Na jednu stranu signál A o normální rychlosti a signál B o poloviční rychlosti, na druhou stranu naopak (A o poloviční a B o normální). A poté, v místě kde se potkají signály A a B, tam označíme výsledné vrcholy a počkáme na ustálení.

Signály nám tak obejdou kružnici jen jednou a celkově tak vykonáme $\mathcal{O}(N)$ taktů výpočtu. Program by mohl vypadat třeba takto:

```
# Proměnné:
# x - rozsah 0..1
# signalA, signalB - rozsah 0..1, vých. hod. 0
# statusA, statusB - rozsah 0..2, vých. hod. 0

if x == 1:
        # Vyšleme úvodní signál na obě
        # strany a skončíme
        signalA = signalB = 1
        stop

# Dostali jsme signál z obou stran
if S[0].signalA and S[1].signalA:
        x = 1
        stop
# A proti směru: Počkáme na odeslání 1 tah
```

```
elif S[0].signalA and statusA == 0:
        statusA = 1
# A po směru nebo již čekáme: Odešleme ihned
elif S[1].signalA or waitA:
        statusA = 2 # Abychom neodesílali znovu
        signalA = 1
# ... obdobně pro signalB a statusB
```

Úkol 2

KSP

K nalezení kostry grafu použijeme simulaci prohledávání do šířky. Spustíme prohledávání z jednoho vrcholu a budeme postupně přivěšovat vrcholy, které patří do dosud nenavštívené části grafu. Každý vrchol přivěsíme za právě jednu hranu, tedy nám určitě nikdy nevznikne cyklus a výsledný graf po zastavení bude strom.

Pokud byl původní graf souvislý, tak ke každému vrcholu existuje od počátečního vrcholu cesta. Ve chvíli, kdy zpracujeme sousední vrcholy, navěsíme i tento vrchol do vznkajícího stromu. Protože výsledný strom bude obsahovat všechny vrcholy, vznikne nám tak kostra.

řešení

Prohledávání budeme dělat tak, že každý dosud nezpracovaný vrchol bude sledovat své sousedy. Ve chvíli, kdy se nějací sousedé stanou součástí vznikajícího stromu (stanou se zpracovanými), vybere si z nich aktuální vrchol jednoho a s tím se spojí hranou (nastaví odpovídající proměnnou na svojí straně na jedničku).

Končit budeme ustálením.

```
# Proměnné:
# a - vstup, rozsah 0..1
# kostra[i] - výstup, rozsah 0..1, výchozí 0

# Už jsem v kostře, jen sleduji sousedy.
if a == 1:
  for i in range(5):
    if S[i].kostra[P[i]]:
      kostra[i] = 1

# Nejsem v kostře. Pokud je nějaký soused
# v kostře, připojím se k němu.
else:
  for i in range(5):
    if S[i].a:
      kostra[i] = 1
      a = 1
```

Prohledání grafu do šířky trvá asymptoticky lineárně dlouho a tedy i celý program pro grafomat skončí po ustálení za $\mathcal{O}(N)$ kroků (tím, že je graf 5-regulární, by šel odhad ještě upřesnit, ale zlepšíme ho tak pouze o konstantu).

Úkol 3

V zadání nám uteklo, že graf je souvislý, za což se omlouváme. Nikoho z řešitelů to však nezmátlo a všichni správně řešili verzi pro souvislý graf.

Mezi došlými řešeními se objevil nápad všechny automaty hned zastavit. To ale trvá právě jeden krok výpočtu a to rozhodně není $C \cdot N$ pro nějaké konstantní C ($C = \frac{1}{N}$, což určitě není konstanta), tudy tedy cesta nevede.

Budeme potřebovat vymyslet řešení, které za každý vrchol stráví nějaký konstantní čas, než předá práci dalšímu vrcholu. K tomu se přímo nabízí využít DFS neboli prohledávání do hloubky.

Začneme v označeném startovním vrcholu a půjdeme po dosud nenavštívených vrcholech tak dlouho, dokud to půjde. Ve chvíli, kdy narazíme na slepou uličku, tak se vrátíme a zkusíme to jinudy. Po každé hraně vzniklého DFS stromu tak právě jednou sestoupíme dolů a právě jednou se po ní vrátíme, skončíme opět ve startovním vrcholu.

Můžeme si to představit tak, že vrcholy si budou předávat nějaký token. Vrchol, který drží token, si vždy vybere nějakého ze sousedů, kterému token odešle (nastaví u sebe proměnnou, které si pak soused v dalším taktu všimne) a správný soused si token převezme. Při vracení se zpět budeme navíc vrcholy zastavovat a výpočet grafomatu tak ukončíme celkovým zastavením.

Tím, že obejdeme graf pomocí DFS, nám vznikne strom na N vrcholech. Ten má ale $N - 1$ hran, takže celkově výše popsaný postup provede $2N - 2$ kroků. Tím, že na začátku počkáme právě dva kroky, už dosáhneme přesného počtu $2N$ kroků. Program může vypadat třeba takto:

```
# Proměnné:
# start - vstup, rozsah 0..2
# navstiveno - rozsah 0..1, výchozí 0
# uzavreno - rozsah 0..1, výchozí 0
# smer - rozsah 0..K, výchozí K

# Hlavní funkce: Pokusí se mezi sousedy najít
# zatím nenavštíveného a předat mu token, pokud
# ale takového nenajde, uzavře aktuální vrchol.

def predejToken():
  for i in range(K):
    if S[i].navstiveno == 0:
      # Pošleme token tímto směrem
      smer = i
      return
  # Pokud se nám nepovede token nikam
  # poslat, uzavřeme tento vrchol
  uzavreno = 1
```

```
  stop

# 1. Na startu musíme počkat
if start == 1:
  start = 2
elif start == 2:
    start = 0
    predejToken()
# 2. Sledujeme, jestli nam nekdo neco neposlal
elif navstiveno == 0:
  for i in range(K):
    # Jestli ukazuje na nás
    if S[i].smer == P[i]:
      navstiveno = 1
      predejToken()
# 3. Pokud už jsme někam token poslali,
# sledujeme, jestli se nám nevrací
else:
  if S[smer].uzavreno:
    predejToken()
```

Úkol 4

Kdybychom měli k dispozici počítadlo, byl by úkol velmi jednoduchý. Stačilo by jen si závorky postupně posouvat doleva k prvnímu vrcholu, zde je zpracovávat a počítat počet otevřených závorek. Pokud bychom se někdy dostali pod nulu, skončili bychom s chybou, stejně jako kdyby na konci mělo počítadlo nenulovou hodnotu.

Všechnu práci budeme dělat v prvním, označeném, vrcholu. V ostatních budeme jen posouvat závorky. Vrchol, který bude potřebovat závorku, si ji vždy nakopíruje od souseda a zapíše v nějaké domluvené proměnné, že si ji vzal. Druhý vrchol se podívá na tuto proměnnou svého souseda, zjistí, že byl „okraden“, a sebere závorku sousedovi na druhé straně.

Takto lze zařídit postupné posunutí všech závorek až do prvního vrcholu, kde se zpracovávají. Jenom musíme na přesun počítat se dvěma kroky výpočtu: v prvním seberu závorku sousedovi, ve druhém teprve soused zjistí, že mu chybí, a sebere ji zase svému sousedovi. Zbývá zařídit počítadlo.

Počítadlo si ale jednoduše můžeme vyrobit jako bitové počítadlo kombinací vícero vrcholů – nejblíže prvnímu vrcholu budu mít nejnižší bity a dále bude jejich hodnota vzrůstat. Největší číslo, které budeme potřebovat, bude N – to zakódujeme do $\log N$ bitů a vrcholy nám tak bohatě stačí.

Na začátku výpočtu bude počítadlo ve všech vrcholech nastaveno na nulu. Při přičtení jedničky zvětšíme počítadlo v aktuálním vrcholu o jedna a kdyby mělo přetéct, nastavíme ho na nulu a do domluvené proměnné uložíme přenos.

Soused se podívá na naší domluvenou proměnnou a případně přenos přičte ke své hodnotě.

Odčítání bude fungovat obdobně: Pokud budu mít ve svém vrcholu jedničku, změním ji na nulu a končím, pokud ale budu mít nulu, změním ji na jedničku a směrem dál odešlu přenos s hodnotou mínus jedna. Pokud takový přenos dojde nazpět až k prvnímu vrcholu, dostal jsem se právě do záporných čísel a vyhlásím chybu.

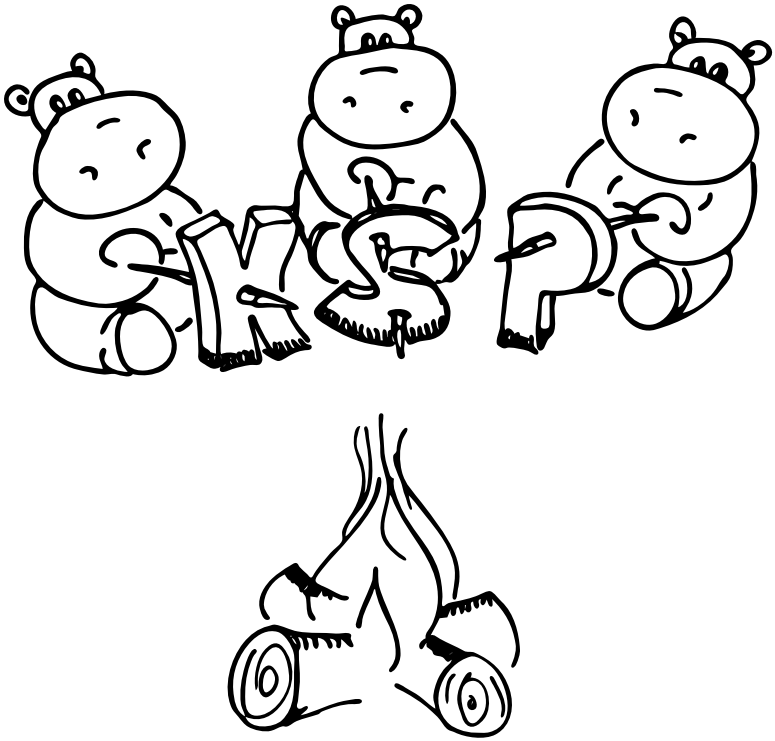
Poslední, co zbývá, je po konci výpočtu zkontrolovat, že nám v počítadlu zbyly samé nuly. To uděláme jednoduše tak, že vyšleme z prvního vrcholu signál, který když cestou narazí na jedničku, vyhlásí chybu. Pokud dojde nazpět až k prvnímu vrcholu, je uzávorkování správné a zahlásíme úspěch.


```
# Proměnné:
# prvni -vstup, rozsah 0..1
# zavorka - vstup, rozsah -1..1
# prenos_zavorky - rozsah 0..1, výchozí 0
# pocitadlo - rozsah 0..1, výchozí 0
# prenos - rozsah -1..1, výchozí 0
# vystup - rozsah 0..1, výchozí 1
# signal - rozsah 0..1, výchozí 0

levy = S[0]
pravy = S[1]

# Reset hodnot
prenos = 0
signal = 0

if prvni == 1:
  # 1. Počítadlo přeteklo do mínusu, nebo přišel
  # signál s chybou -> chyba
  if levy.prenos == -1 or levy.vystup == 0:
    vystup = 0
    stop

  # 2. Zpracování závorek, dokud jsou
  if prenos_zavorky:
    # Jenom počkáme, než soused bude mít
    # připravenou novou závorku
    prenos_zavorky = 0
  else:
    if zavorka == 1:
      prenos = pocitadlo
      pocitadlo = (pocitadlo + 1) % 2
    elif zavorka == -1:
```

```
      prenos = -1 * pocitadlo;
      pocitadlo = (pocitadlo - 1) % 2
    # 3. Závorky došly, spuštění kontroly
    else:
      signal = 1

    # Zkopírujeme si závorku zprava
    zavorka = pravy.zavorka
    prenos_zavorky = 1

else:
  # Reset hodnot
  prenos_zavorky = 0

  # 1. Pokud přišla chyba, vyhlásíme ji také
  # a končíme
  if levy.vystup == 0 or pravy.vystup == 0:
    vystup = 0
    stop

  # 2. Přenos závorek
  if levy.prenos_zavorky:
    # Pokud ještě existuje závorka napravo
    if pravy.prvni != 1 and pravy.zavorka != 0:
      zavorka = pravy.zavorka
      prenos_zavorky = 1
    else: # Již není závorka napravo
      zavotka = 0

  # 3. Počítadlo
  if levy.prenos == 1:
    prenos = pocitadlo
    pocitadlo = (pocitadlo + 1) % 2
  if levy.prenos == -1:
    prenos = -1 * pocitadlo;
    pocitadlo = (pocitadlo - 1) % 2

  # 4. Závěrečná kontrola počítadla
  if levy.signal:
    if pocitadlo =! 0:
      vystup = 0
      stop
    else:
      signal = 1
```

Přesunování závorek nám na každou z nich zabere 2 kroky, aktualizaci počítadla zvládneme dělat průběžně během posouvání závorek, a nakonec ještě jednou projdeme všechny vrcholy. Dohromady tak provedeme $\mathcal{O}(N)$ kroků výpočtu.

Jirka Setnička



Pořadí řešitelů KSP

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
0.				40	300.0
1.	Martin Raszyk	G_Karvina	4	34	278.5
2.	Jan Špaček	G_Wicht	3	33	276.3
3.	Marek Černý	G_Chrudim	3	29	267.6
4.	Michal Punčochář	GJírovcČB	4	23	215.3
5.	Michal Korbela	GJJesen	4	28	213.6
6.	Václav Rozhoň	GJirsíkaČB	3	21	201.7
7.	Jakub Svoboda	GKomHavíř	4	29	195.4
8.	Matej Lieskovský	GOmskPha	4	24	195.1
9.	Richard Hladík	GOAMarLaz	1	21	176.3
10.	Aneta Šťastná	GOmskPha	4	22	153.4
11.	Jakub Zárybnický	GTomkovaOL	3	26	141.8
12.	Václav Volhejn	GKepleraPH	1	18	124.9
13.	Jan-Sebastian Fabík	GJarošeBO	4	13	118.6
14.	Jan Knížek	G_Strakon	3	18	95.2
15.	Filip Bialas	GOpatovPHA	1	12	95.1
16.	Antonín Češík	SPSE_Pard	4	11	94.6
17.	Lucie Studená	GKepleraPH	4	15	83.8
18.	Jakub Maroušek	G_Písek	4	12	77.7
19.	Jan Pokorný	G_Bučovice	2	10	77.3
20.	Ondřej Hübsch	GArabskáPH	4	8	73.9
21.	Štěpán Hojdar	GJírovcČB	4	12	73.8
22.	Anna Steinhauserová	GDačice	4	11	71.7
23.	Dorian Řehák	GCoubTábor	3	10	67.5
24.	Anna Gajdová	GFPValMez	3	6	55.8
25.	Štěpán Trčka	GSlavičín	3	9	54.3
26.	Jonatan Matějka	SŠP_ČB	4	7	50.9
27.	Stanislav Lukeš	GPísnickáPH	1	4	37.2
28.–29.	Dalimil Hájek	GKepleraPH	3	4	34.1
	Matěj Konečný	GJírovcČB	3	4	34.1
30.	Adam Španěl	ArcibisGPH	2	4	32.0
31.	Dominik Roháček	SPŠLegioJI	4	5	27.0
32.	Jan Tománek	GPelhřimov	3	3	25.2
33.	Antonín Teichmann	GJeronýmLI	4	3	22.6
34.	Jan Pavlovský	GJiM	4	3	21.3
35.	Tomáš Marius	SŠkybernHK	1	3	20.6
36.	Marek Dobranský	GHorMichal	4	3	20.3
37.	Aneta K. Lesná	GZborovPH	1	5	16.8
38.	Michal Hloušek	GNadŠtolPH	1	3	16.3

Pořadí řešitelů KSP

39.	Přemysl Šťastný	GŽamberk	0	3	16.0
40.	Petro Kostyuk	GEBenešeKL	4	2	12.1
41.	Radovan Švarc	G_ČTřebová	3	1	8.0
42.	Václav Končický	GSOŠ_FrMís	3	1	7.4
43.	Tadeas Friedrich	GOhradníPH	4	1	6.3
44.	Jan Horešovský	GMělník	4	1	6.2
45.	Michal Martinek	GHavPodl	3	1	6.0
46.	Josef Čech	GJMasar_JI	2	1	5.7
47.	Marek Židek	GTomkovaOL	4	1	4.0
48.	Ladislav Tlapák	G_Břeclav	−1	1	2.5
49.	Michal Kužela	GSlavičín	2	2	2.0

Jiří Setnička a kolektiv

Korespondenční seminář z programování XXVI. ročník

Autoři a opravující úloh:
Jan Bok, Karolína Burešová, Lukáš Folwarczný, Jan Hadrava
Ondřej Hlavatý, Mark Karpilovskij, Dominik Macháček, Martin Mareš
Lucie Mohelníková, Petra Pelikánová, Michal Pokorný, Vojtěch Sejkora
Jiří Setnička, Filip Štědronský, Karel Tesař, Michal Vaner
Pavel Veselý

Autoři příběhů v zadání:
Jiří Setnička, Karolína Burešová, Lukáš Folwarczný, Martin Mareš

Vydal MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 467. publikaci.

TEX-ová makra pro sazbu ročenky vytvořili Martin Mareš, Jan Matějka, Radim Cajzl a Jiří Setnička.

S jejich pomocí ročenku vysázel Radim Cajzl.

Obrázek na obálce nakreslila Petra Pelikánová.

Sazba byla provedena písmem Computer Modern v programu TEX.

Vytisklo Reprostředisko UK MFF.

Vydání první, 278 stran
Náklad 200 výtisků
Praha 2014

Vydáno pro vnitřní potřebu fakulty.
Publikace není určena k prodeji.

ISBN 978-80-7378-273-3

ISBN 978-80-7378-273-3