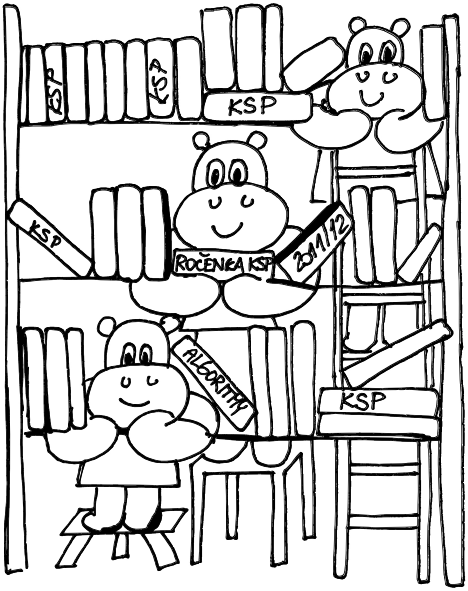
MARTIN BÖHM A KOLEKTIV

Korespondenční seminář z programování

XXIV. ročník – 2011/2012



matfyzpress
VYDAVATELSTVÍ
MATEMATICKO-FYZIKÁLNÍ FAKULTY
UNIVERZITY KARLOVY V PRAZE

MARTIN BÖHM A KOLEKTIV

Korespondenční seminář z programování

XXIV. ročník – 2011/2012

matfyzpress

Praha 2012

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

ISBN 978-80-7378-227-6

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož dvacátý čtvrtý 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 sedmi úloh okořeněné příběhem. Poslední úloha je doplněna tzv. *seriálem*, což je povídání o nějakém zajímavém informatickém tématu prolínající se celým ročníkem. Ten je zde uveden samostatně.

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

KSP se i přes svou dlouhou tradici neustále vyvíjí. V tomto ročníku jsme začali u některých úloh nabízet jejich jednodušší verze, zejména s myšlenkou, že budou sloužit jako návodné úlohy. Také jsme letos zorganizovali praktickou programovací soutěž Kasiopea. V blízké budoucnosti chceme řešitelům umožnit komunikaci s námi čistě online, bez využití papírové pošty.

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
KSVI MFF
Malostranské náměstí 25
118 00 Praha 1

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

(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.

Zadání úloh

První série

```
FD 60 BK 120 FD 60 RT 90 FD 50
LT 90 FD 60 BK 120 FD 60
```

```
10  PRINT "e"
```

```
++++++++++[>++++++++++<-]>--.
```

```
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook! Ook? Ook! Ook!
Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook? Ook.
Ook? Ook! Ook. Ook? Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook! Ook. Ook? Ook.
```

```
print split /[^g-q]/, lc sub {};
```

```
#include <stdio.h>
int main() {
  printf("%c", (107^25)-70&99);
  return 0;
}
```

```
h(X,[]) :- put(X), !.
h(X,[A|B]) :- Y is X + A, h(Y,B).
?- h(42, [5, 6, 7, 8, 7]).
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SERIE1.
PROCEDURE DIVISION.
    DISPLAY 'S'.
    STOP RUN.
```

```
(defun gen (i j)
   (if (<= i j)(cons i (gen (1+ i) j)) nil))
(defun split (s)
   (mapcar (lambda (i) (substring s (1- i) i))
           (gen 1 (length s))))
(defun GAPS (x)
   (princ (caddr (split (symbol-name x)))))
(defun Q (x)
   (eval (list x (list 'quote x))))
(Q 'GAPS)
```

```
class Program {
  static void Main() {
    System.Console.Write((char)0x21);
  }
}
```

Toto je jen malý náhled do zvěřince programovacích jazyků. Dokážete rozpoznat jednotlivé jazyky? Umíte zjistit, co programy v nich udělají a co z toho vznikne dohromady? Pokud ne, může vám pomoci malý výlet do historie.

První předzvěstí programovacího jazyka byl v roce 1801 vynález tkalcovského stavu ovládaného děrnými štítky (kartami z tvrdšího papíru, jež obsahovaly data zakódovaná do děr). Jednalo se však spíše o kód než jazyk.

Dalším významným počinem se stalo v půli 19. století napsání prvního programu, a to na počítání Bernoulliho čísel. Kupodivu ho nenapsal muž, ale Ada Lovelace v korespondenci s Charlesem Babbagem pro jeho analytický stroj.

Až doba elektromechanických počítačů z konce 30. let a ze 40. let 20. století odstartovala rychlý vývoj programovacích jazyků. Hybatelem pokroku byla nepřekvapivě druhá světová válka, zejména touha prolomit šifry nepřátelské strany.

Tehdy se programovalo pomocí přepínačů či děrných štítků (v podstatě sekvencí 0 a 1) a pro každý počítač jinak (bylo jich tehdy naštěstí jen několik na světě). Problémy tohoto způsobu se však objevily celkem rychle, především v množství chyb, jež raní programátoři udělali.

To vedlo k vývoji vyšších jazyků. První návrh, pojmenovaný Plankalkül, vymyslel Konrad Zuse (autor elektromechanických počítačů Z1, Z2, Z3 a Z4), nikdy ho však neimplementoval.

Po strojových kódech přišly „assemblery“, které nahrazovaly binární zápisy anglickými slovy jako „add“ a „xor“.

24-1-1 Podvádíme s XORem — 8 bodů

Představte si, že jste s kolegou z práce dostali obrovskou hromadu hardwarových součástek, přičemž každá má nějakou cenu danou přirozeným číslem. Chcete je rozdělit na dvě části, aby byl v obou stejný součet cen.

Kolega však není váš kamarád, a tak ho zkusíte podvést. Vy rozdělíte součástky na dvě hromádky a on si součty překontroluje programem v assembleru, jenže nebude tušit, že dělá operaci XOR místo sčítání (což jste mu nenápadně prohodili).

Operace XOR (exkluzivní OR, neboli vylučovací nebo) pracuje se dvěma čísly po bitech tak, že ve výsledném čísle je na i-tém místě jednička, když byla jednička na i-tém místo právě v jednom ze vstupních čísel (tzn. ne v obou).

Příklad (čísla jsou v binárním zápisu, v závorce desítkově):

```
    11001001 (201)
XOR 01100101 (101)
    --------------
 =  10101100 (172)
```

Máte tedy seznam přirozených čísel, který chcete rozdělit tak, aby XOR všech prvků byl v obou částech stejný, ale rozdíl součtů co největší (menší připadne přirozeně kolegovi).

K této úloze není potřeba vymyslet algoritmus nebo napsat program, jde spíše o nalezení způsobu rozdělování čísel.

Prvním z moderních vyšších programovacích jazyků, jež se stále používají, je FORTRAN (FORmula TRANslator) z roku 1955, původně určený pro vědeckotechnické výpočty. Stal se předchůdcem dnešních imperativních jazyků, v nichž se program zapisuje jako posloupnost příkazů s přesně daným pořadím vyhodnocení.

Brzy ho následoval zcela odlišný LISP (LISt Processor), první funkcionální jazyk. Zlí jazykové mu přezdívají Lots of Irritating Superfluous Parentheses (spousta otravných nadbytečných závorek), protože téměř každý příkaz je ohraničen kulatými závorkami.

Funkcionální se podobným jazykům říká, protože zachází s výpočtem jako s vyhodnocováním matematické funkce. Představují jeden z přístupů deklarativního programování, v němž se na rozdíl od imperativního jen určuje, co se má udělat, kdežto imperativní jazyk popisuje i postup.

Druhým rozšířeným deklarativním přístupem je logické programování, které vzniklo začátkem 70. let. Nejznámějším zástupcem je Prolog, v němž jsou jednotlivé části programu v podstatě logické formule.

Koncem 50. let do rodiny imperativních jazyků přibyl ALGOL (ALGOrithmic Language) uzpůsobený pro přehlednější zápis algoritmů. Jako první přišel s bloky příkazů, které byly vyznačeny slovy begin a end.

Že jste už begin a end někde viděli? Ano, z ALGOLu se koncem 60. let vyvinul Pascal, nejdříve určený pro výuku programování, ovšem dodnes rozšířený i v komerční sféře.

Od 60. let se s jazyky doslova roztrhl pytel. Jmenujme jen ty významné, rozšířené nebo alespoň něčím zajímavé.

V roce 1964 byl vytvořen BASIC (Beginner's All-purpose Symbolic Instruction Code), stejně jako FORTRAN a ALGOL imperativní. Při jeho vytváření byl kladen důraz na snadné používání a podobnost angličtině. Jeho dialekty a pokračovatelé jako Visual BASIC jsou dodnes hojně používané.

24-1-2 Rozházené řádky v BASICu — 7 bodů

Programátorský šotek měl veselou náladu, a tak přeházel řádky ve vašem už značně dlouhém programu v BASICu. Naštěstí je na řádcích na začátku napsáno jejich číslo (na rozdíl od starých verzí BASICu, kde se typicky číslovalo po desítkách, čísla v tomto programu začínají od 1 a přibývají po jedné).

Soubor lze spravit pouze prohazováním dvojic řádků, ale vy se jako správní programátoři nechcete moc nadřít a rádi byste provedli co nejméně prohození, abyste dostali původní program.

Vymyslete algoritmus, který dostane na vstupu posloupnost N čísel od 1 do N, v níž se žádné neopakuje (tedy permutaci), a má určit, na kolik nejméně prohození 2 řádků lze dostat seřazenou posloupnost od 1 do N.

Příklad: pro permutaci `3, 10, 8, 4, 6, 5, 9, 1, 2, 7` je správnou odpovědí 6.

Z konce 50. a začátku 60. let pochází také zvláštní programovací jazyk APL založený na matematické notaci. Programy v něm jsou typicky jednořádkové a vyhodnocují se striktně zprava doleva (tedy v APL neexistuje nic jako priorita operátorů).

Ptáte se, jak se program dokáže vejít na jednu řádku? Docela pěkně, když jsou operátory jednoznakové, jen je pro ně třeba použít tolik znaků, že většina není na běžných klávesnicích. Pár příkladů: ÷ (dělení), ⍴ (zjištění rozměrů pole), více jich můžete najít v předloňském seriálu.[1]

Na počátku 70. let vznikl v Bellových laboratořích současně se systémem Unix jazyk C vyvinutý z B (B už však nepředcházelo žádné A, zato jazyk D z přelomu tisíciletí navazuje na C).

C bylo navrženo více nízkoúrovňově, a tudíž je vhodné na systémové a výkonově náročné aplikace. Po svých předchůdcích zdědilo označování bloků znaky `{` a `}`.

24-1-3 Turnaj jazyků — 12 bodů

Když už jsme si tu několik programovacích jazyků představili, můžeme mezi nimi uspořádat velký turnaj, jehož se zúčastní i jazyk budoucnosti, BestLang. Ten je přirozeně zcela nejlepší a vždy vyhraje.

[1] `http://ksp.mff.cuni.cz/viz/22-4-7`

V každém kole turnaje je zadána úloha, přední odborníci na jednotlivé jazyky v každém napíší řešení a do dalšího kola postupují ty jazyky, jejichž programy doběhly do dvojnásobku času nejlepšího řešení.

Jak bylo řečeno, BestLang vyhraje. Ale chce vyhrát s co nejvíce body a ty se počítají za každé kolo vzorcem

$$\frac{\textit{počet vyřazených} \cdot 100\,000}{\textit{počet na začátku kola}}.$$

Dělení ve vzorci je celočíselné (počítá se dolní celá část) a *počet na začátku kola* obsahuje i BestLang. Celkový počet bodů určuje součet bodů ze všech kol.

BestLang je navíc tak dobrý, že si může v každém kole vybrat, kolik jazyků vyřadí (všechna řešení v ostatních jazycích doběhnou ve skoro stejném čase a BestLang dokáže zjistit, v jakém). V kole nemusí vyřadit žádný jazyk.

Máte daný počet jazyků (N) a počet kol (K), platí $N \leq 1\,000$ a $K \leq 1\,000$, a úkolem je najít takovou posloupnost počtu vyřazených v jednotlivých kolech, aby BestLang získal co nejvíce bodů.

V jakémkoliv kole může klidně vyřadit všechny, ale po posledním kole musí zůstat v turnaji sám.

Příklady: pro $N = 500$ a $K = 3$ je řešením `437, 55, 7`, pro $N = 15$ a $K = 8$ je to `3, 3, 2, 2, 1, 1, 1, 1`.

Pojďme si povědět ještě něco o programovacích jazycích obecně – hlavně o tom, jaké jsou jejich druhy. Mezi nejvýraznější patří rozdělení na nízkoúrovňové (více se blížící strojovému kódu, tedy jazyku počítače) a vyšší (bližší člověku).

Zástupcem první skupiny je např. assembler. Dalších nízkoúrovňových není tolik, pokud nebudeme hledět na odlišnosti dané hardwarem, a programátoři se s nimi setkávají málo, většina vývoje i dělení se proto týká jen vyšších jazyků.

Při procházení historických jazyků jsme už nakousli dělení na jazyky imperativní (program je posloupnost příkazů) a deklarativní (zapisuje se, co se má spočítat, a nezáleží tolik na tom, jak). Známými deklarativními jazyky jsou LISP, Haskell (oba funkcionální) a Prolog (logické programování).

Všechny imperativní jazyky jsou dnes procedurální, ačkoliv zprvu neobsahovaly podprogramy, neboli procedury či funkce. Často se proto musel používat příkaz `goto` *(skok na jiné místo v programu), což vedlo k nepřehlednosti. Dnes už se* `goto` *téměř nepoužívá, i když v programovacích jazycích často bývá.*

O objektový přístup obohatil programování jazyk Simula určený pro diskrétní simulace. Obsahuje objekty, třídy, dědičnost, a dokonce i garbage collection *(automatickou správu paměti).*

Z hlediska rychlosti provádění kódu a pohodlnosti při psaní jsou dvěma protipóly jazyky kompilované (kompilátorem převáděné do strojového kódu) a inter-

pretované (program, jemuž se říká interpretr, čte kód a rovnou ho provádí, což bývá pomalejší).

Pokud vám na předchozím odstavci něco nesedí, pak je to dobře. Jazyk je totiž jen forma zápisu, a tak existují interpretry jazyka C, typického zástupce kompilovaných, a naopak kompilátory pro skriptovací jazyk Python.

24-1-4 Složitá složitost — 7 bodů

Ačkoliv si v této sérii vyprávíme o programovacích jazycích, při řešení úloh KSPčka nám o ně obvykle moc nejde – víc než použitý jazyk, ať už kompilovaný nebo interpretovaný, nás zajímá nás tzv. asymptotická časová složitost. Ta je zcela nezávislá na jazyce a umožňuje rozlišit, jak je který algoritmus rychlý, aniž bychom ho museli spouštět na skutečném počítači.

Více se o složitosti dozvíte z kuchařky na konci letáku. Její přečtení doporučujeme před řešením této úlohy všem, kdo ještě nikdy složitost neurčovali nebo jim stále přijde poněkud složitá. Ostatním může kuchařka sloužit pro osvěžení znalostí před novým ročníkem.

V této úloze jsme si pro vás připravili pseudokód (zjednodušený kód) algoritmu. Jeho úkolem je setřídit zadané pole čísel, čili jeho prvky přerovnat do vzestupného pořadí. Vaším úkolem pak je určit časovou a paměťovou složitost tohoto algoritmu a zdůvodnit, proč tomu tak je. Zajímá nás složitost v nejhorším případě.

Ještě poznámka pro pořádek: pole indexujeme od 1 a parametr `n` říká, kolik v něm je uloženo prvků.

```
Funkce Setřiď(pole, n):
   odm = odmocnina z n zaokrouhlená dolů
   i = 1
   Dokud i <= n:
      změna = true
      Dokud změna je true:
         změna = false
         Pro j od i do min(i+odm-2, n-1):
            Jestliže pole[j] > pole[j+1]:
               Prohoď pole[j] a pole[j+1]
               změna = true
      i = i + odm
   vysl = vytvoř pole délky n
   zač = vytvoř pole délky odm+1
   Do všech prvků pole zač vlož 0
   Pro i od 1 do n:
      vysl[i] = nekonečno
      minIndex = 1
```

```
        j = 1
        k = 1
        Dokud j <= n:
           Jestliže zač[k] < odm a j+zač[k] <= n:
              a = pole[j + zač[k]]
              Jestliže a < vysl[i]:
                 vysl[i] = a
                 minIndex = k
           j = j + odm
           k = k + 1
        zač[minIndex] = zač[minIndex] + 1
     Vrať vysl
```

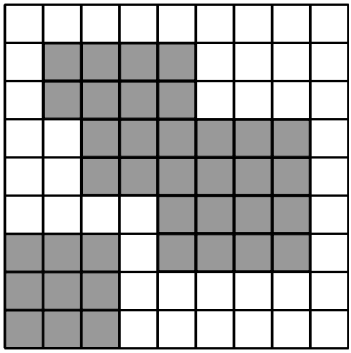
Zvláštní kategorii tvoří výukové jazyky, snažící se být co nejjednoduššími a zároveň poutavými pro děti. Někdy jsou v nich textové příkazy nahrazeny ikonkami, z nichž se vytváří program metodou táhni a pusť.

V Logu, starém již přes 40 let, se ovládá želva, která chodí po ploše, a když spustí ocásek, kreslí za sebou stopu. Tomuto způsobu kreslení se dodnes říká želví grafika.

24-1-5 Razítková grafika — 12 bodů

Představme si vytváření grafiky trochu podobné želví, ale s jedinou operací – otisk čtvercového razítka. Kreslit budeme na klasickou bitmapu (čtvercovou síť pixelů) jedinou barvou, například černou na bílé pozadí.

Dostali jste černobílý obrázek o rozměrech $N \times M$ pixelů a vaším úkolem je určit, pomocí jakého největšího razítka mohl být vytvořen. Žádný pixel nesmí být orazítkován dvakrát.



Na obrázku je příklad vstupní bitmapy. Největší razítko, kterým se dá vytvořit, má velikost 1 pixel, razítkem se 2 pixely by nešel nakreslit čtverec 3×3

vlevo dole.

Dalším jazykem pro děti je v ČR vytvořený Karel (pojmenovaný po Karlu Čapkovi), v němž se na čtvercové síti ovládá robot. Domácího původu je i Baltík obsahující postavičku čaroděje, který chodí po ploše a čaruje na políčka obrázky.

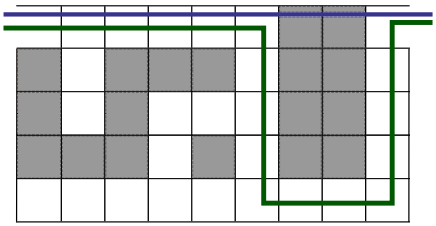
24-1-6 V bludišti s krumpáčem — 9 bodů

V Baltíkovi i v Karlovi je typickou úlohou pro začátečníky procházení bludiště. Postavička chodí po čtvercové (popř. obdélníkové) síti, musí se vyhýbat zdem, ale občas se může rozhodnout nějakou zbourat.

Máte mapu velkého bludiště na čtvercové síti s rozměry $N \times M$. Políčko je buďto prázdné, nebo je na něm zeď. Úkolem je najít pro postavičku nejrychlejší cestu od vchodu do bludiště k východu (je tam jen jeden) s tím, že zbourání zdi stojí stejně času jako ujít $K - 1$ políček (takže políčko se zdí postavička projde celkově za stejný čas jako K prázdných).

Pro jednoduchost stačí vypsat dobu na projití nalezené trasy. Můžete také předpokládat, že K je nejvýše 10.

Příklad bludiště vidíte níže na obrázku. Pro K menší než 5 se vyplatí probourat dvě zdi, pro K větší než 5 je lepší zdi obejít a pro $K = 5$ jsou stejně dobré obě cesty.



Léta vývoje programovacích jazyků přinesla i mnohé plody, jež jsou hezké, zajímavé či alespoň vtipné, leč v praxi naprosto nepoužitelné. Jedním z nejznámějších je Brainfuck, který si své jméno skutečně zasluhuje.

Běh programu v něm si lze představit jako operace nad polem bytů, přičemž je k dispozici jen jeden ukazatel na aktivní buňku, s níž jedinou lze pracovat bez změny ukazatele. K tomu všemu stačí 8 instrukcí reprezentovaných 8 znaky, ostatní se ignorují.

ZOO takovýchto esoterických jazyků je opravdu pestrá. Obsahuje nejen příbuzné Brainfucku (např. Ook!), ale třeba i Malbolge, který se snaží, aby bylo programování v něm co nejobtížnější, INTERCAL, v němž je nutno mimo jiné o provedení příkazu prosit, avšak ne moc, a Whitespace, který využívá jen znaků mezera, tabulátor a nová řádka.

Jelikož má Brainfuck stejnou výpočetní sílu jako jiné běžné jazyky (populárně řečeno), jako demonstraci užitečnosti uveďme jazyk HQ9+ se 4 instrukcemi pokrývajícími typické testovací úlohy pro jazyky, leč nic jiného:

`h` – *vypíše „Hello, world!“,*

`q` – *zobrazí zdrojový kód programu,*

`9` – *vypíše text k písničce „99 Bottles of Beer on the Wall“ (ano, má 100 slok),*

`+` – *zvýší o 1 hodnotu akumulátoru.*

Že programování je umění (dokonce abstraktní) a psát není potřeba, dokazuje Piet, pojmenovaný po holandském malíři Pietu Mondrianovi. Některá jeho abstraktní díla vypadají skoro stejně jako programy v Pietu, reprezentované bitmapou s maximálně 20 barvami.

Pokud vás netradiční programování zaujalo, pěknou sbírku roztodivných jazyků najdete na specializované wiki.[2]

Jak bude vypadat vývoj jazyků v budoucnosti? Někteří tvrdí, že nic nového, převratného do 10 let nepřijde a na špičce se udrží stávající jazyky, které se budou jen pomalu vyvíjet a dále přejímat prvky z funkcionálních jazyků. Třeba nás ale někdo překvapí!

Jedním z nových trendů, jež se nyní rychle rozvíjí, je paralelismus, vycházející z myšlenky rozdělit dlouho trvající výpočty mezi několik procesorů nebo počítačů. Třeba se dají takto prolamovat některé jednodušší šifry.

24-1-7 Distribuované výpočty — 10 bodů

Firma Hack & Crack vlastní N (tedy mnoho) počítačů vzájemně propojených mezi sebou (ne nutně každý s každým). Rozhodla se, že prolomí šifru americké armády, a na výpočet nasadila veškeré síly.

Uběhlo pár dní a programátoři z hrůzou zjistili, že je ve výpočtu chyba a musí se přerušit. Postupně tedy vypínají počítače, chtějí však, aby se v každém okamžiku mohly všechny běžící počítače spolu domluvit (tj. mezi každými dvěma lze přes nějaké jiné poslat zprávu).

Pomozte jim najít pořadí, v němž mají vypínat počítače (očíslované od 1 do N). Na vstupu kromě N dostanete i seznam dvojic kabelem propojených počítačů (propojení je obousměrné).

Můžete předpokládat, že na začátku lze poslat zprávu mezi každými dvěma počítači. Je-li řešení více, stačí najít jen jedno.

Příklad: ve firmě je 7 počítačů a propojené jsou `1-2, 1-3, 2-3, 3-4, 3-5, 3-6, 3-7, 5-6, 6-7`. Řešením je například vypínat v pořadí `4, 5, 7, 6, 3, 1, 2` nebo `2, 5, 6, 7, 4, 1, 3` a nebo mnoha jinými způsoby.

[2] `http://esolangs.org/wiki/`

Ve vzdálené budoucnosti by se klidně mohlo programovat v angličtině nebo i v češtině. Hello world by vypadal třeba takto:

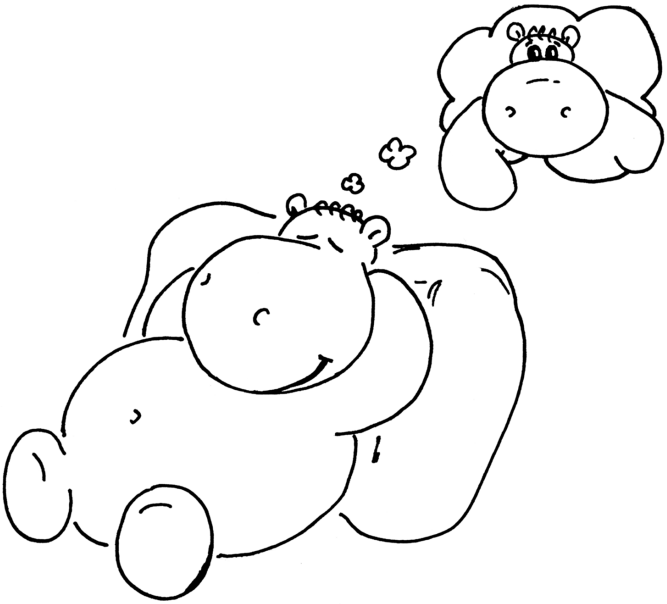
```
Vypiš „Dobrou noc, světe!" (bez uvozovek)
a skonči.
```

Co byste říkali na takovýto program?

```
Vyřeš všechny úlohy z 1. série.
Zapiš jejich řešení po jednom do PDF.
Odevzdej řešení přes web KSP.
```

Zatím jediným alespoň trochu funkčním překladačem češtiny se zdají být čeští programátoři. Dokážete napsat kompilátor pro počítač, který bude dobrý alespoň jako člověk, co neprogramuje?

Povídání o jazycích sepsal Pavel Veselý.



Druhá série

Bylo nebylo, povídali si takhle dva členové prvobytně pospolné společnosti, kolik má který ovcí. Jeden měl donedávna deset ovcí, leč před pár dny se jedné z nich narodilo jehně, a nyní má tedy ovcí jedenáct.

Onen přírůstek jedenácté ovce byl sám o sobě radostnou událostí, nicméně jejího šťastného majitele trápil drobný problém. Už nemohl jednoduše ukázat na prstech, kolik ovcí má, musel ukázat celých 10 a poznamenat k tomu, že má ještě jednu navíc.

Když si postěžoval kamarádovi, oba se zamysleli, co s tím. Po chvíli vzal jeden z nich poměrně ostrý primitivní nástroj, jenž by se dal označit jako nůž, sebral ze země delší klacík a udělal na něm 11 zářezů.

Možná tak, možná nějak jinak vznikla tato primitivní metoda počítání ovcí. Jistě se však hodila jednomu z jejich potomků, který tuhle na louce pásl stádo, když tu najednou zjistil, že v sousedním lese hoří. Navíc jediná rozumná cesta z té louky vedla právě tím lesem.

24-2-1 Požární poplach — 11 bodů

V lese hoří. Představme si les jako čtvercovou síť, na každém políčku je buďto skála (neprůchozí políčko), požár, nebo les. Požár se za jednotku času rozšíří na všechna sousední políčka, na kterých je ještě les.

Lesem je však potřeba bez úhony projít a nás zajímá, jak dlouho to ještě bude možné. Váš program dostane na vstupu nejprve rozměry lesa (R a S) a potom R řádků délky S složených ze znaků `@` (oheň), `#` (skála) a `.` (les).

```
6 6
@#...@      @#@@@@
.#.###      @#@###
......      @@@@@@
####.# →    ####@#
.....#      ....o#
####..      ####..
```

Na výstupu vypíšete, kolik jednotek času bude les ještě průchozí zleva doprava. To znamená, že z alespoň jednoho políčka na levém okraji bude existovat cesta pouze nehořícím lesem do nějakého políčka na pravém okraji. Pohyb je povolen pouze svisle a vodorovně, nikoli šikmo.

Pro zobrazený vstup je správným řešením číslo 7 – oheň se rozhoří jako na druhém obrázku. O chvíli později by již hořelo i políčko označené `o` a les by byl neprůchozí.

Jste jistě zvědavi, k čemu pasáčkovi byla ona slibovaná metoda. Jednoduše si po průchodu lesem spočítal, kolik ovcí mu zbylo a kolik ovcí uhořelo. Co jiného byste čekali?

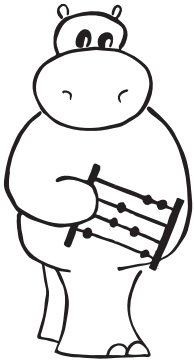
* * *

Uplynulo mnoho vody v řece, přes kterou potom pasáček převedl ovce, aby neuhořely v rozsáhlém lesním požáru, než někoho napadlo počítat třeba přesouváním kuliček na počítadle. I to je však nástroj značně dávného původu.

Každý z nás snad někdy počítal na počítadle v první třídě, občas někdo slyšel o ruském sčotu.

Dovolme si malou odbočku. V Rusku bylo ještě před nějakou dobou naprosto běžné počítat na sčotech. Byl o ně tudíž velký zájem, a tak byly nedostatkovým zbožím.

Problém byl v chybně umístěném centrálním skladu. Velení armády totiž rozhodlo (sčoty byly strategickým zbožím), že všechny vyrobené sčoty se budou svážet do centrálního skladu do Moskvy a odtamtud distribuovat po celém Rusku.



24-2-2 Centrální sklad — 9 bodů

Pro zjednodušení si představme celé Rusko jako přímku. Na přímce leží N bodů – výrobců sčotů. Nalezněte ideální místo pro centrální sklad – takové, že průměrná vzdálenost mezi centrálním skladem a výrobci bude minimální.

Na vstupu je na prvním řádku číslo N a na druhém N čísel oddělených mezerou – souřadnic výrobců. Vypište jediné číslo – souřadnici centrálního skladu. Je-li více možných řešení, vypište libovolné z nich.

Generální štáb vzápětí zjistil, že chyba byla nejen ve špatně umístěném centrálním skladu, ale i v centralizaci celého zásobování, takže sčoty byly nedostatkovým zbožím i nadále.

Zajímavým stupněm vývoje byly takzvané Napierovy kostky.[3] *John Napier na přelomu 16. a 17. století vynalezl zajímavou dřevěnou pomůcku, která usnadňovala zvláště násobení dlouhého čísla jednociferným.*

Pomůcka obsahovala podlouhlé hranoly, pro každé číslo od 0 do 9 jeden, na kterých byly vhodně napsané násobky těchto čísel – postupně 0-násobek až 9-násobek. Když se pak poskládaly hranoly správně k sobě, stačilo už akorát přepočítat přenosy.

6	**3**	**5**
06	03	05
12	06	10
18	09	15
⋮	⋮	⋮
48	24	40
54	27	45
60	30	50

→

Řádek 9:

	6	**3**	**5**
· 9	54	27	45
5	7	1	5

Sčítáme zprava doleva našikmo . . .

Tedy $635 \cdot 9 = 5715$.

24-2-3 Odčítání — 7 bodů

Následující program simuluje něco jako mechanické počítadlo… tedy aspoň jej simulovat má. Jestli to je pravda, ověřte vy.

Předložená funkce má odčítat dvě čísla a vracet jejich rozdíl. Její vstup má být zadán tak, že na pozici `[0]` je číslice nejvyššího řádu.

Varianta v C bere jako první dva parametry vstup, ve třetím vrací výstup a čtvrtý určuje, kolik cifer čísla mají. Program v Pythonu bere vstup ve svých parametrech a pole vrací přímo.

Zadání je v desítkové soustavě (cifry 0–9) a čísla musí mít stejně cifer, byť by jedno z nich mělo začínat nulami.

[3] `http://en.wikipedia.org/wiki/Napier%27s_bones`

Když tedy chcete odčítat $635 - 21$ v C, voláte

```
int p[] = {6, 3, 5};
int d[] = {0, 2, 1};
int v[3];
funkce(p, d, v, 3);
```

a v Pythonu jednoduše `funkce([6,3,5], [0,2,1])`.

V kódu nehledejte ani syntaktické chyby, ani podivný styl, ani jiné formální problémy. Vaším úkolem je dokázat nebo vyvrátit jeho správnost a v každém případě určit jeho složitost (časovou i paměťovou).

Zde je kód v C:

```
void funkce(int prvni[], int druhe[],
            int vysledek[], int delka) {
  int index = 0, i;
  for (i = 0; i < delka; ++ i)
    vysledek[i] = prvni[i] + 9 - druhe[i];
  vysledek[delka - 1] ++;
  while (index < delka) {
    if (vysledek[index] >= 10) {
      vysledek[index] -= 10;
      index --;
      if (index >= 0)
        vysledek[index] ++;
      else
        index ++;
    } else
      index ++;
  }
}
```

V Pythonu:

```
def funkce(prvni, druhe):
  vysledek = prvni[:]     # Kopie prvního pole
  for i in range(0, len(druhe)):
    vysledek[i] += 9 - druhe[i]
  vysledek[-1] += 1       # -1 = poslední prvek
  index = 0
  while index < len(vysledek):
    if vysledek[index] >= 10:
      vysledek[index] -= 10
      index -= 1
```

```
      if index >= 0:
        vysledek[index] += 1
      else:
        index += 1
    else:
      index += 1
  return vysledek
```

Napierovy kostky byly mimochodem v 19. století překonány ještě šílenějším vynálezem – Genaillovými-Lucasovými pravítky.[4] *Tato pravítka počítala automagicky i přenosy.*

Autor příběhu si pravděpodobně jednu takovou sadu pořídí a bude s ní machrovat na zkoušce z Analýzy III.

⋆ ⋆ ⋆

Tou dobou také začínaly vznikat první mechanické počítací stroje, obvykle na objednávku bankovních ústavů nebo zámožných obchodníků, kteří potřebovali (jak jinak) počítat peníze.

Autory těchto strojů byli například Blaise Pascal nebo Gottfried Wilhelm Leibniz. Roku 1820 pak šéf pojišťovacích společností Charles Thomas sestrojil už poměrně sofistikovaný stroj, který uměl sčítat, odčítat, násobit i dělit ve velmi rychlém čase.

Onomu stroji se říkalo Arithmometer a zvládnul vynásobit dvě osmimístná čísla za 18 sekund a na dělení potřeboval necelou půlminutu.

Byl to na svou dobu dokonalý výrobek, takže Charles Thomas založil první továrnu na výrobu počítacích strojů. Jeden její obchodní cestující prý prodal Arithmometer i na tehdejších Královských Vinohradech (součástí Prahy se staly až v roce 1922). Jak by to vypadalo dnes?

24-2-4 Odbočení vlevo — 7 bodů

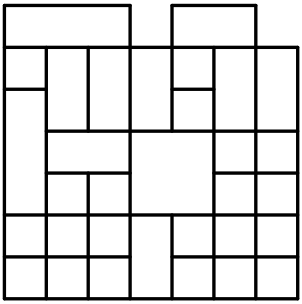
Pražské Vinohrady se vyznačují tím, že na žádné křižovatce není povoleno odbočit vlevo. Vždy se smí jet jen doprava a rovně. Alespoň to tvrdí zlí jazykové.

Obchodní cestující se potřebuje dostat z jednoho místa na Vinohradech na druhé. Vaším úkolem bude najít mu nejkratší cestu, přičemž je potřeba respektovat globální zákaz odbočení vlevo.

Na vstupu dostanete mapu Vinohrad – čtvrti s pravoúhlou soustavou ulic (což je podmnožina jednotkové čtvercové mřížky); dále pak start a cíl cesty (nějaké dva úseky ulic). Výstupem vašeho algoritmu bude itinerář sestávající z příkazů typu „jeď rovně“ a „odboč vpravo“.

[4] `http://en.wikipedia.org/wiki/Genaille%E2%80%93Lucas_rulers`

Jednu možnou mapu Vinohrad jsme vám zde připravili jako příklad. Nějaký start a cíl si jistě vymyslíte sami.



Počítací stroje se dále vyvíjely, v polovině 20. století bylo například možno občas potkat příruční kalkulačku Curta, která se vešla do dlaně. Dlouho ji prý používali například v rallye, a to i v době elektronických kalkulaček, které nevydržely otřesy při jízdě.

⋆⋆⋆

V první polovině 20. století začínali konstruktéři počítacích strojů pomalu opouštět plně mechanická zařízení. Vznikaly například reléové počítací stroje, nebo později elektronkové počítače.

Tehdejší počítače však zpočátku nebyly dvojkové – neměly logické obvody, ale složitější členy. Nepočítalo se v nich pouze v nulách a jedničkách, ale spojitě v napětí mezi nulou a nějakým maximem.

Pak však kohosi napadlo, že by se dalo počítat jinak než analogově – číslicově, ale ne v desítkové soustavě, jak bývalo zvykem, ale ve dvojkové. Vznikaly tedy stroje počítající ve dvojkové soustavě, průkopníkem byl například Zuse Z3.

Jiné stroje, například ENIAC, počítaly v desítkové soustavě, ale každá cifra byla kódována do 4 bitů, se kterými se počítalo dvojkově (BCD – Binary Coded Decimal).

24-2-5 Logická formule — 11 bodů

V průkopnické době digitálních přístrojů řešili vývojáři a konstruktéři různé zajímavé úlohy. Například tuto.

Mějme zadaný neuzávorkovaný logický výraz, který obsahuje pouze nuly, jedničky, AND a OR. Například

```
0 AND 1 AND 0 OR 1.
```

Nalezněte, kolika různými způsoby je možno zadaný výraz úplně uzávorkovat, aby jeho hodnota byla 1, a kolika způsoby naopak dostaneme nulu. V našem příkladu by třikrát vyšla 0 a dvakrát 1:

```
(0 AND 1) AND (0 OR 1) = 0
0 AND (1 AND (0 OR 1)) = 0
0 AND ((1 AND 0) OR 1) = 0
((0 AND 1) AND 0) OR 1 = 1
(0 AND (1 AND 0)) OR 1 = 1
```

Pak už nastoupila éra tranzistorů a šlo to ráz na ráz. Výhodou tranzistorů byla značná úspora místa. Najednou mohly počítače zabírat ne jednu velkou místnost, ale jen jednu velkou plechovou bednu. A nebo se do té velké místnosti vešlo víc výpočetního výkonu.

Tou dobou bylo prodáno okolo 10 000 kusů IBM 1401. Z toho počítače už byla největší součástí čtečka děrných štítků...

Navíc byly tranzistory na výrobu výrazně levnější než elektronky.

Takové stroje už byly dostatečně výkonné na to, aby dokázaly počítat všemožné zajímavé úlohy. Nebyly však ještě dostatečně výkonné na to, aby počítaly neefektivně.

24-2-6 Závorky — 13 bodů

⚠ Mějme na začátku N levých závorek. Nyní nám začnou chodit příkazy „otoč závorku na pozici i“. Po každém takovém příkazu vypíšete, jestli je teď řetězec dobře uzávorkovaný.

Dobře uzávorkovaný řetězec levých a pravých závorek splňuje podmínku, že levé a pravé závorky je možno přirozeným způsobem spárovat.

Program si na začátku může něco předpočítat – na vstupu dostanete N, což bude počet závorek v řetězci. Potom bude postupně dostávat výše definované příkazy a hned je bude zpracovávat.

Nemůžete si tedy počkat, až dostanete třeba celou posloupnost příkazů, nebo je brát po několika. Vždy přijde příkaz a vy jej zpracujete. Pak teprve přijde další příkaz atd.

Nějakou dobu vývojáři zmenšovali tranzistory, až někoho napadlo dát jich do jednoho pouzdra víc – v jednu dobu se sešly rovnou dva vynálezy integrovaných obvodů – Jack St. Clair Kinby a Robert Norton Noyce nezávisle na sobě vynalezli mikročip na konci 50. let 20. století.

Trvalo už jen pár let, než byl vynalezen mikroprocesor. V roce 1971 byl vyroben mikroprocesor Intel 4004.

V roce 1981, o celých 10 let později, pak miniaturizace dosáhla takového stavu, že bylo možno vydat IBM PC.

To byl první počítač, který se vešel na stůl a byl rozumně levný, takže jeho rozšíření nabylo značných rozměrů a firma IBM měla značné zisky.

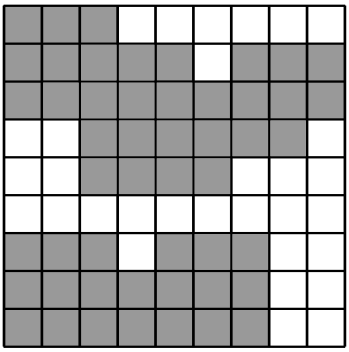
Na výsluní slávy se pomalu začal dostávat Microsoft se „svým“ MS-DOSem (slovy zlých jazyků, Messy, Slow and Dirty Operating System)...

24-2-7 Štětcování — 10 bodů

V době vydání IBM PC bylo potřeba vyrobit propagační plakát.

Grafici dostali podivné zadání (ale není se čemu divit vzhledem k tomu, jak vypadá například instrukční sada procesoru Intel 8088...) – plakát je potřeba nakreslit co největším štětcem.

Jak se kreslí štětcem velikosti K? Vybereme si na čtvercové síti libovolný čtverec velikosti $K \times K$. Ten vybarvíme; postup opakujeme.



Obrázek je černobílý (tedy políčko je buďto vybarvené, nebo nevybarvené). Políčka je možno obarvit opakovaně.

Na vstupu dostanete obrázek jako tabulku `1` a `0`; na výstupu vypište jedno celé číslo – K – maximální možnou velikost štětce.

Například pro uvedený obrázek platí $K = 2$.

IBM PC v podstatě vytlačil z trhu veškerou konkurenci. Jeho jednoduchá a modulární architektura spolu s procesory firmy Intel (8088 apod.) způsobila, že náklady na výrobu i opravy byly (na svou dobu) velmi nízké.

Navíc v podstatě všechny další procesory, které Intel vyvíjel, byly s 8088 zpětně kompatibilní, co se týče instrukční sady. Nebyl proto problém spustit starý program na novějších strojích, což byl další důvod masivního rozšíření této platformy.

I současné počítače v sobě v drtivé většině mají procesory, na kterých při troše vůle i prastaré programy půjdou spustit. Nebude to asi ani pohodlné, ani rychlé, nicméně s velkou pravděpodobností poběží.

Z notebooku s procesorem Intel Core 2 Duo se s vámi loučí autor příběhu

Jan „Moskyto“ Matějka

Třetí série

Odpadla poslední cihla a archeolog-dobrodruh se konečně dostává do hrobky, kde na něj čeká hora pokladů. Ouha, při vší nedočkavosti došlápne na špatný panel na zemi a jakoby odnikud se na něj valí obří balvan...

O čem asi dnešní příběh bude? O archeologii? Brakové literatuře? Tak alespoň o sférických objektech? Kdepak, o té nejzajímavější *části minulého odstavce, o vstupních zařízeních. Hlavně o těch, co se dají zapojit do našeho nejlepšího kamaráda – počítače.*

Pokud vás téma nenadchlo, soucítíme s vámi. Zde je úloha na usmířenou.

24-3-1 Intervalové duplicity — 12 bodů

Máme posloupnost přirozených čísel délky N. Na vstupu kromě této posloupnosti dostaneme také K dotazů – intervalů. Máme rozhodnout pro každý dotaz zvlášť, jestli se v intervalu čísel ze zadané posloupnosti opakuje alespoň jedna hodnota.

Pokud přeskočíme pár tisíciletí a podíváme se do 30. let 19. století, možná nás překvapí, že spolu s Babbageovým známým Analytickým strojem už byly vynalezeny jak děrné štítky, tak klávesnice – tehdy ovšem ještě odděleně, klávesnice jako součást pouze prvních psacích strojů. Děrné štítky nejprve sloužily jako instrukce jednoduchým automatům (jako samohrajícím piánům na Divokém západě).

Psací stroje se začaly více prodávat a šířit až někdy v 60. a 70. letech 19. století, kdy také vzniklo dnes takřka standardní rozložení kláves QWERTY. Počítače na svůj vzestup ještě čekaly a tehdejší prototypy byly stále ovládány děrnými štítky.

Děrné štítky nakonec na svůj soumrak čekají dlouhou dobu – informatici na Matfyzu jsou stále strašeni historkami o ladění programů zadávaných pomocí děrných štítků. Příliš flexibilní vstupní zařízení to vskutku nebyla.

Klávesnice se u počítačů (po experimentech ve 40. letech 20. století) objevily až v 60. letech, společně s prvními video terminály (na počítačích MULTICS). Poměrně rychle vytlačovaly děrné štítky, neboť to bylo vylepšení opravdu podstatné. Tedy, alespoň tam, kde na novější počítače byly peníze.

24-3-2 Nemnoho počítačů — 10 bodů

V jedné méně bohaté společnosti mají N počítačů, kde každý počítač má přiřazeno přirozené číslo – typ počítače. Dozvěděli jsme se, že firma vlastní maximálně $\log_2 N$ různých druhů počítačů (tedy je na vstupu jen $\log_2 N$ různých hodnot).

Našim úkolem je vymyslet algoritmus, který za takovéto podmínky setřídí N čísel (typů počítačů) ze vstupu co nejrychleji.

Například vstup `3 2 4 4 2 3 4 2` má jen 3 různé hodnoty, což je $\log_2 8$.

Setříděná posloupnost je potom `2 2 2 3 3 4 4 4`.

I na klávesnicích samotných se zrcadlil rychlý vývoj 20. století. Psací stroje používaly kladívka, která se po stisku klávesy obtiskla na papíře. Při vyšších rychlostech psaní se však kladívka často zasekávala, což neúměrně zpomalovalo psaní. Proto prý vzniklo rozložení QWERTY – cílem bylo minimalizovat počet stisků kláves blízko u sebe (což bylo hlavní příčinou zasekávání).

První klávesnice zapojené do počítače byly prostě jen namačkané přepínače na sobě, každý pro jednu klávesu zvlášť. Takové řešení však bylo příliš drahé a tehdy také hůře použitelné.

Poměrně rychle bylo tedy vynalezeno dnes nejčastější řešení – membránové klávesy s gumovými čepičkami, které po stisku spojí desku s tištěnými spoji dole s grafitovou částí uvnitř, což vyšle signál pro danou klávesu.

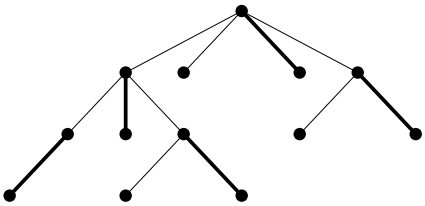
Druhý nejčastější typ kláves jsou nůžkové spínače, které jsou založeny opět na gumové membráně a jejím stisku, ovšem stisku napomáhají křížící se podpěry s malou volností, díky čemuž klávesy nepotřebují tak velký prostor a zdají se placatějšími.

Tyto klávesy jsou časté u laptopů a také u některých výrobců stolních klávesnic. Oba dva hlavní typy klávesnic jsou velmi levné, a tak většina populace zná jen tyto.

24-3-3 Párování znalců — 10 bodů

Vrátili jsme se do naší hypotetické firmy a nyní koukáme na hierarchii zaměstnanců a jejich šéfů. Každý zaměstnanec (kromě ředitele) má jednoho přímého nadřízeného, ředitel šéfuje celé firmě a ve firmě nejsou žádní lidé, co by si byli navzájem šéfem i podřízeným. (Jinak řečeno, zaměstnanci tvoří strom.)

Kvůli kurzu psaní všemi deseti potřebujeme rozdělit zaměstnance do co nejvíc nepřekrývajících se dvojic tak, že ve dvojici je vždy jeden zaměstnanec a jeho přímý nadřízený. Nalezněte algoritmus, který pro danou firmu spočítá maximální možný počet těchto dvojic.



Jaké další typy klávesnic ti „znalci“ vlastně znají? Kromě membránových klávesnic existují ještě klávesnice, kterým se říká mechanické. Tyto klávesnice

sice také často využívájí membránové čepičky, ale často je kombinují s jinými technologiemi, které mají za cíl vyvážit nevýhody membránových klávesnic.

Za hlavní nevýhodu je považována velmi malá odezva kláves, takže pisatel musí vyvinout větší tlak na klávesu, aby ji stiskl.

Mezi tyto klávesnice patří například jejich hlavní zástupce, mechanické spínačové klávesnice, které podobně jako jejich předkové používají spínač pro každou klávesu zvlášť, v kombinaci s pružinkou a zvukovou odezvou po stisknutí. Dále k nim řadíme také kapacitní klávesnice a klávesnice s ohýbající se pružinkou.

Ačkoli klávesnicoví gurmáni mnohdy preferují mechanické klávesnice nad těmi levnými, jejich skutečné výhody mohou být hodně subjektivní a hlavně malé v porovnání s řádově vyšší cenou klávesnice.

I u membránových klávesnic existují kvalitní produkty s rozumnou odezvou i cenou – je třeba být pyšný na českou firmu ZF Electronics Klášterec s.r.o, výrobnu kvalitních jak membránových, tak mechanických klávesnic v ČR.

Ironií ovšem je, že ačkoli se klávesnice stále vyrábí, většina jejich produktů není dostupná na českém trhu a musíte je dovézt například ze sousedního Německa či Rakouska.

Mechanické klávesnice (hlavně díky své vysoké ceně) tedy nezískaly na popularitě a byly zcela poraženy jejich levnějšími bratránky, zatímco jiné oblasti práce s počítačem (grafické prvky, zvuk) si udržely na trhu kvalitní produkty díky použitelnosti v profesionální sféře.

Až bude soused vyhazovat svoji starou klávesnici z 90. let, raději se na ni běžte podívat – mnohdy lidé našli v sousedství starou klávesnici IBM, která se pak dala prodat za pěkný peníz na internetu.

24-3-4 Návrat do podposloupnosti — 12 bodů

Představme si na chvíli, že jsme firmou IBM a držíme v ruce seznam všech našich verzí klávesnic. Verze jsou vlastně přirozená čísla. Občas ale proběhne v IBM reorganizace, a tak posloupnost čísel verzí nemusí být rostoucí.

Zajímalo by nás, jakou nejdelší souvislou rostoucí podposloupnost čísel verzí klávesnic v seznamu najdeme. Avšak nejsme jen obvyklá firma, jsme IBM – máme ve skladu stroj času na jedno použití. Můžeme se tedy vrátit v čase a jeden souvislý úsek verzí (rostoucí nebo ne) prostě ze seznamu škrtnout – a pak hledat nejdelší souvislou rostoucí podposloupnost čísel ve zbytku.

Vymyslete algoritmus, který nám v tomto hledání pomůže. Pokud existuje více řešení, vypište libovolné z nich.

Například pro seznam `1 4 7 2 3 5 9 1` vyškrtneme část `4 7` a dostaneme tak rostoucí souvislou podposloupnost `1 2 3 5 9`.

A tak mnoho z nás píše na levných klávesnicích a jsme šťastni ve zdraví. Nebo nejsme? Syndrom karpálního tunelu je skutečná záležitost, a přestože mno-

ho z nás se stále těší dobrému ručnímu zdraví, autor tohoto článku nedoporučuje pohodlí při psaní zanedbat.

Jste-li pyšní na to, jak rychle píšete, může se vám lehce stát, že za pár let pyšní nebudete – možná ve svých 40 letech už nebudete moci psát vůbec.

Nicméně neznamená to, že musíte hned přejít na pohodlnější klávesnici. Existuje třída klávesnic, které se snaží zachránit pisatelům ruce, nazývají se ergonomické.

Z osobní zkušenosti mohu potvrdit, že na některých se opravdu píše pohodlněji. Stejně jako u mnoha jiných nemocí ovšem není zcela prokázáno, že syndrom karpálního tunelu je tvořen psaním na špatné klávesnici a že ergonomické klávesnice mají jakýkoli prospěšný efekt.

Budete-li přemýšlet nad svým zdravím, zamyslete se hlavně nad tím, jak u počítače sedíte a na jaké židli.

24-3-5 Součin zlomků — 10 bodů

Nelamte si u počítače páteř, zkuste si radši zlámat pár zlomků. Na vstupu máte seznam racionálních čísel – zlomků zapsaných v základním tvaru. Vaším úkolem je zlomky vynásobit a vypsat výsledek opět v základním tvaru.

Pozor, zlomků může být hodně a přestože se každé číslo na vstupu i výsledek vejdou do celočíselného datového typu, už neplatí, že by se každý mezivýsledek musel do takového typu vejít. Celý vstup se také do paměti vejde.

Například na vstup `17/63 100/99 81/85 77/20` vypíšete výstup `1` nebo `1/1`. Vstup i výstup má jistě čitatele i jmenovatele menší než bajt, nicméně po vynásobení prvních dvou členů získáme `1700/6237`...

Nejen klávesnicemi vstupuje člověk do počítače. V 70. letech vzniklo další dnes všudypřítomné vstupní zařízení – myš. K zrodu myši se váže tato anekdota:

Když Steve Jobs přijel do vývojového střediska Xeroxu v Palo Alto, ukázali mu tam třítlačítkové zařízení za 300 dolarů – myš. Byl jí tak unesen, že se rozhodl myši dodávat ke svým počítačům Apple.

Apple v té době ovšem neprodával předražené produkty, takže se jim podařilo zjednodušit původní myš od Xeroxu a snížit cenu za 15 dolarů, což byl první odraz myši do světa (stolních) počítačů.

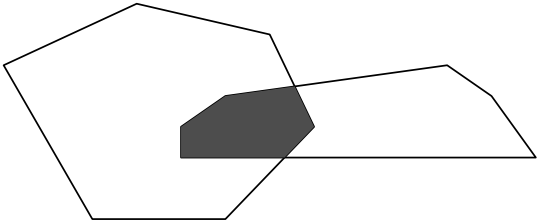
Přiznejme si, že toto vše trochu minulo český trh, neboť po revoluci v roce 1989 už prodávali myši všichni velcí hráči. Postupně myš ztratila své kolečko, které se muselo čistit od prachu, počet tlačítek se neustále měnil, až se vrátil na 3 i více...

Mimochodem, pokud vám bude po odstavci o mechanických klávesnicích líto, že jednu takovou nemáte doma, můžete se utěšit tím, že vlastně máte – akorát je třítlačítková a jmenuje se myš.

24-3-6 Průnik plánů — 13 bodů

Zaměstnanec Applu plánuje proniknout do sídla Xeroxu, aby mohl co nejlépe okopírovat jejich myš. Drží před sebou plány obchodního a výzkumného střediska, obě budovy se trochu překrývají. Představme si je jako dva konvexní mnohoúhelníky.

Po spuštění poplachu nebude mít mnoho času a chce tedy projít jen tu oblast, které tyto dvě budovy mají společnou – průnik. Vymyslete program, který mu pomůže tento průnik najít.



Na závěr nám dovolte malý pohled do budoucnosti. Klávesnice byla vynalezena hlavně proto, aby zrychlila převod textu do tisku (a později do počítače), neboť psaní rukou bylo dosti nepohodlné a pomalé. Nebylo to ale zrychlení na všech frontách (například matematické přednášky se stále dost špatně zapisují do počítače bez použití OCR nebo vlastních notací).

Jak zrychlit vstup ještě více? Ačkoli jsme v tom stále ještě břídilové, rozpoznávání zvuku a subjektivně zajímavější rozpoznavání mozkových vln postupuje dále a je možné, že brzy se stanou dominantními technikami zaznamenávání lidských myšlenek do nul a jedniček. Už teď jsou na trhu poměrně zajímavé hračky.[5]

Klávesnice, myši a jiné sice nezmizí zcela ze světa (u nás doma máme stále videokazety a videopřehrávač), ale možná je naši potomci budou znát jen z vyprávění nás melancholických stařiků. Třeba jsme jedna z posledních generací, která na nich bude umět psát. To je fajn, ne?

Mimochodem, Češi na klávesnicích psát celkem umí – i v psaní na počítači se konají mezinárodní soutěže (i pro středoškoláky) a Češi jsou často na prvních příčkách. Například Intersteno.[6]

Programátorské soutěže však obsahují stále ještě o trochu víc přemýšlení nad úložkami, jako je tato:

24-3-7 Mazání závorek — 8 bodů

Na vstupu se nachází uzávorkování délky N s K různými druhy párových závorek. Uzávorkování může a nemusí být korektní. Vašim úkolem je zjistit,

[5] `http://en.wikipedia.org/wiki/Brain%E2%80%93computer_interface`

[6] `http://www.intersteno.org/`

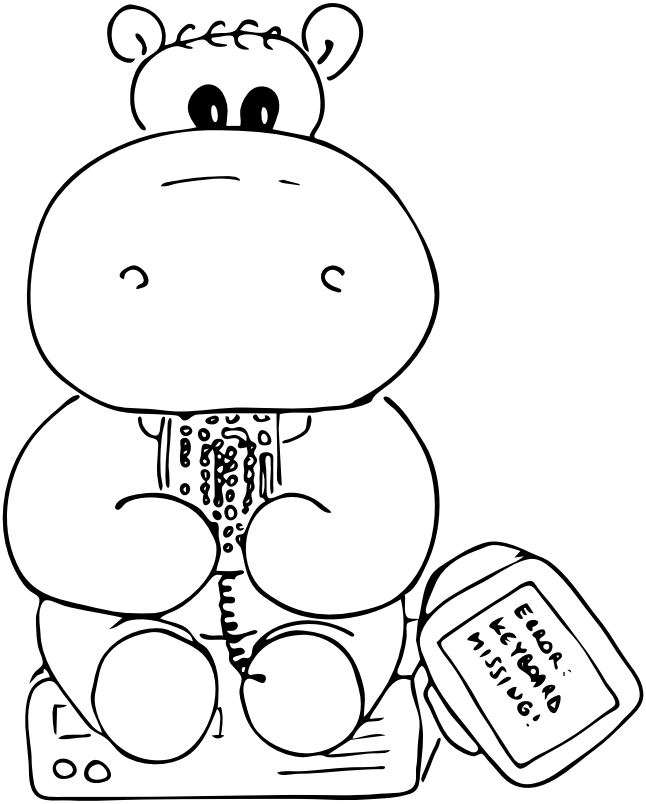
jestli je korektní, a pokud není, jestli existuje nějaký druh závorek takový, že po odebrání všech závorek tohoto typu bude zbytek už korektně uzávorkován.

Například uzávorkování `([{]})` pro $N = 6$ a $K = 3$ není korektní, ale po odebrání závorek typu `[]` se takovým stane, stejně jako když odebereme závorky typu `{}`.

Povídání o vstupních zařízeních bylo dlouhé, ale mnoho oblastí jsme prakticky zatajili. Joysticky, volanty, pedály, rozložení kláves na klávesnici, jakékoli detaily a tak dále.

Pokud by vás zajímalo víc... odložte psací stroje a zkuste se podívat na internet. Slyšeli jsme, že se na něm dá najít spousta věcí.

Martin Böhm



Čtvrtá série

Den první: „Já se na to tedy podívám“ povzdechl si John a vstal od papírování. Přitom hodil okem po kalendáři. Pátý leden 2137, už jenom tři dny do slavnostního otevření sekce výstupních zařízení muzea počítačů. A pořád nemáme funkční exponáty, zaklel v duchu a poklusem se dal k oddělení elektronkových počítačů.

U přesné kopie prvního elektronkového počítače ENIAC z roku 1946, jak hlásal holografický panel, jej přivítal jeden z techniků. „Problém je s tímhle panelem,“ řekl a ukázal na desku se žárovičkami. Ta, jak si John pamatoval, neměla u původního ENIACu žádný klíčový význam, ale měla sloužit pro vizualizaci výpočtu. Běžní lidé chtěli vidět, že ta ohromná konstrukce něco dělá a blikající světla byla nejlepší. Od té doby také několik desítek let přežívala představa počítače, jako stroje s chaoticky blikajícími kontrolkami.

24-4-1 Iniciály předků — 10 bodů

Tým techniků chce nechat na panelu se žárovičkami postupně zobrazovat nějaký řetězec znaků, aby panel pěkně blikal a upoutalo to procházející lidi. Panel je tvořený spoustou sloupců žárovek a každý sloupec umí zobrazit jeden znak.

Technici si jako správní hračičkové sepsali iniciály všech svých předků až do 20. století a ty chtějí nechat zobrazovat na panelu.

Nicméně, čím je řetězec delší, tím déle se musí vkládat do paměti počítače. Technici si chtějí ušetřit práci, a tak by chtěli znát takovou jeho nejkratší část, jejímž opakováním se dá vypsat celý řetězec.

Vaším úkolem je tedy napsat program, který si na vstupu přečte řetězec znaků (složený z velkých písmen anglické abecedy), nalezne v něm nejkratší úsek, jehož opakováním vznikne celý řetězec, a vypíše jeho délku.

vstup	odpověď
`ABABAB`	2
`ABABA`	5
`AAA`	1

John se po vyřešení problému ještě chvíli procházel oddělením nejstarších počítačů a zkoumal další výstupní zařízení. U jedné staré tiskárny se dal do řeči s průvodcem, který si zrovna cvičil svůj výklad.

„Než přišly na scénu různé obrazovky, velmi oblíbenou metodou výstupu byl tisk na různý tiskárnách či elektronických psacích strojích,“ začal průvodce. „Neuměly samozřejmě tisknout nějakou grafiku, natožpak trojrozměrně, jako ty dnešní. Ale zvládaly celkem rychle tisknout znaky. Tiskárny zvládající tisk nějakých jiných obrazců než znaků přišly až v 60. letech 20. století.“

Přešel k prvnímu exponátu „Nejdříve se používaly jehličkové tiskárny, které se stylem fungování podobaly psacím strojům. Pomocí jehliček přitiskávaly barevnou pásku na papír... pozor, pane, ta páska pořád barví. Až teprve počátkem 70. let se začal prosazovat laserový tisk. Ten funguje v základě tak, že se pomocí laseru na správných místech vybije elektrostaticky nabitý rotující selenový válec. Toner se přichytí pouze na vybitá místa, pohybem válce následně přilne na papír. A nakonec se toner do papíru zapeče.“

„A co inkoustové tiskárny, myslel jsem, že přišly dříve než laserové?“

„To je častý omyl. Inkoustové tiskárny se začaly prosazovat až počátkem 80. let, ale protože byly levnější na výrobu, prosadily se hlavně v domácím prostředí. Fungují tak, že se inkoust v tryskách tiskové hlavy zahřeje pomocí maličkého elektrického tělíska asi na 300°C *a pak vlivem tlaku ve velké rychlosti vystříkne z trysky na papír.“*

Výklad o tiskárnách byl ale přerušen jedním ze strážných muzea. „Problém šéfe, spadnul nám systém zjišťování polohy exponátů. Víme, kde exponát je, ale systém už nevyhodnotí, jestli je pořád uvnitř budovy, nebo ne.“ To tu ještě scházelo, pomyslí si John, ale nedávaje na sobě znát únavu posledních dní se vydává za strážným do velínu bezpečnosti, kde si nechává vysvětlit fungování celého systému, tedy spíše jeho nefungování. Bude nutné ho celý přepsat.

24-4-2 Sledování exponátů — 8 bodů

Hranice muzea počítačů má tvar nekonvexního mnohoúhelníku. Na sledovaném exponátu je připevněno čidlo, které vysílá jeho aktuální polohu (určenou například pomocí GPS).

Měli byste strážným v muzeu pomoci tím, že vymyslíte postup, jak zjistit, jestli je exponát ještě na území muzea, nebo ne. Jediné, co máte k dispozici, je poloha exponátu a posloupnost vrcholů hranice muzea.

Byla už skoro půlnoc, když John konečně vítězoslavně klepl do potvrzovací klávesy a zvedl se od počítače. Tak, další problém vyřešený, poklepal se v duchu po rameni. Teď ale musím stihnout ten banket v aule muzea.

Rychle proběhl skrz kancelář, vzal na sebe společenský oblek a pospíchal, ať stihne alespoň půlnoční přípitek.

24-4-3 Cinkání skleničkami — 8 bodů

Přípitky ve 22. století mají několik základních pravidel. Stojí se v kruhu, všichni si musí cinknout se všemi a dále se nesmí cinkat „křížem“ (když si dva páry lidí cinkají, nesmí se jim zkřížit ruce).

A aby to nebylo tak jednoduché, cinká se v taktech. Vždy na úder gongu si člověk buď cinkne s někým, nebo zůstane stát. Pak na další úder gongu s dalším člověkem a tak dále, dokud si necinkne se všemi.

Vás zajímá, kolik nejméně takovýchto taktů bude potřeba, aby si navzájem cinklo N lidí a také správný postup, jakým si budou cinkat. Nezapomeňte dokázat, že to na méně taktů nejde.

Druhý den ráno přišel John do práce s hrozným bolením hlavy – neměl to včera s těmi přípitky přehánět. V kanceláři si jenom vzal něco na bolest, počkal, až prášek zabere, a pak šel zkontrolovat konečné přípravy otevření expozice. Jen co vešel do hlavní haly, všiml si podivného ruchu u dveří skladu.

„Jako by nám někdo nepřál otevření," uvítal ho vrchní skladník. „Máme výpadek proudu ve skladu. A to zrovna potřebujeme navézt několik beden s těmi, jak se jim říkalo... monitory, myslím. Jsme schopni nabít každému vozíku baterie trochou energie, ale chtělo by to nějak optimalizovat jejich trasy, jinak to prostě z toho skladu nestihneme vyvézt."

24-4-4 Vozíky ve skladu — 10 bodů

Moderní sklad 22. století je obsluhován pouze automatickými elektrickými vozíky. Ty normálně čerpají energii z rozvodné sítě vozíků, ale při výpadku této sítě jsou schopné fungovat i na baterie. Samotný sklad je spleť křižovatek a uliček. Uličky jsou obousměrné, mohou se křížit i víceúrovňově a setkávají se pouze na křižovatkách.

Navíc pod podlahou některých uliček jsou silnoproudé vodiče, které svým magnetickým polem ztěžují vozíkům průjezd. Silnoproudé vodiče jsou napojeny na oddělený okruh, výpadek je tedy neovlivní. Samozřejmě v nich „teče" střídavý proud, který indukuje (střídavé) magnetické pole – to v jedné půlce svojí periody vozík zpomaluje, ve druhé zrychluje.

Skladníci zjistili, že by toto pole mohli využít – nastavili vozíky tak, aby jim průjezd libovolnou uličkou trval vždy stejně dlouho, a to právě půlku periody střídavého proudu. Délka cesty a magnetické pole ovlivňují jen spotřebu energie.

Vzhledem k výpadku napájení se hlavní skladník pokouší optimalizovat trasy jednotlivých vozíků a potřebuje od vás najít energeticky nejúspornější trasu (tj. trasu, při níž vozík spotřebuje nejméně energie z baterií) mezi dvěma křižovatkami, které si určí.

Na vstupu dostanete mapu skladu popsanou jednotlivými křižovatkami spolu s uličkami, které mezi nimi vedou. Každá ulička má danou spotřebu energie při průjezdu. Dále dostenete seznam uliček, pod kterými vedou silnoproudé vodiče – můžete si je představit tak, že každý lichý průjezd libovolnou křižovatkou zdvojnásobí jejich energetickou náročnost, každý sudý průjezd ji vrátí do počátečního stavu.

Samozřejmě víte i odkud kam má vozík jet – tedy startovní a cílovou křižovatku. Nejúspornější trasu vypište jako pořadí křižovatek. Nezapomeňte, že skladníci spěchají, vozík tedy nesmí nikdy stát.

Příklad: máme 5 křižovatek očíslovaných 0 až 4 a chceme vozík přepravit z 0 do 1. Všechny uličky obsahují silnoproudé vodiče a vedou mezi křižovatkami (v závorce je energie spotřebovaná při průjezdu): 0 a 1 (21), 0 a 2 (10), 2 a 3 (5), 3 a 4 (2), 2 a 4 (5), 4 a 1 (10).

Nejvýhodnější je použít cestu $0 \to 2 \to 3 \to 4 \to 1$, při níž se spotřebuje 39 jednotek energie. Kdyby se jelo uličkou z 0 rovnou do 1, stálo by to 42 jednotek; cesta $0 \to 2 \to 4 \to 1$ by stála 45 jednotek.

Když se ze skladu konečně dostaly i poslední palety s monitory, John si oddechl. Mezitím, co se hlavní skladník zabýval vozíky, si John dokonce něco stihl nastudovat i o prastarých monitorech.

Jak psali ve starém propagačním letáku, první monitory byly jednobarevné, napevno vestavěné do počítačů a nebylo možné k jakémukoliv počítači připojit jakýkoliv monitor. Za první univerzální grafickou kartu se standardizovaným adaptérem se dá považovat až Monochrome Display Adapter (MDA) z roku 1981 od Intelu, k němuž se dal přes konektor podobný pozdějšímu VGA (ale s méně piny) připojit jakýkoliv monitor s tímto konektorem. MDA umožňoval výstup 80 sloupců na 25 řádků znaků.

Později se objevily i karty podporující nejen znakový, ale i grafický režim a v roce 1987 přišel standard Video Graphics Array (VGA) se svým konektorem, který přežil přes 25 let. Stále se ale jednalo o analogový výstup. První digitální výstup do připojeného monitoru přišel až v roce 1999 společně se standardem a konektorem Digital Visual Interface (DVI).

John přestal pročítat brožuru, rychle nalistoval poslední kapitolu se základním rozebráním principu dvou nejrozšířenějších zobrazovačů přelomu tisíciletí a četl. Staršího byla technologie katodové trubice, tedy CRT monitory. Fungovaly na stejném principu jako tehdejší televize. Elektronové dělo vysílalo proud nabitých částic, které byly usměrňovány velkými elektromagnety, na dopadovou plochu zvanou stínítko. Tam se pomocí látky zvané luminofor proud elektronů měnil na viditelné světlo.

Druhou technologií, která se začala kvůli ceně prosazovat až počátkem 90. let a k jejímuž masovému rozšíření došlo až po přelomu tisíciletí, byla technologie tekutých krystalů LCD. Pracovala na principu zastiňování světla. Za deskou z tekutých krystalů bylo osvětlovací těleso, produkující bílé viditelné světlo. Samotná deska z tekutých krystalů pak v závislosti na natočení krystalků buď světlo v daném bodě propouštěla, nebo ne. Natočení krystalků v jednotlivých bodech bylo řízeno pomocí slabého elektrického proudu.

„Teda, ti si s tím vyhráli," hvízdl obdivně John a odložil brožuru. Pak se podíval směrem ke vstupu do expozice, kde měla skupinka pracovníků problém s naprosto současnou zobrazovací technikou, s holografickými projektory.

24-4-5 Holografické projektory — 12 bodů

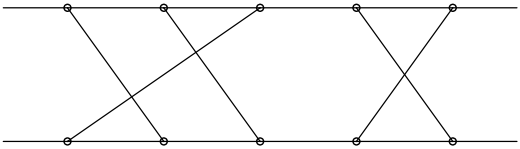
Do expozice muzea se vstupuje dlouhou chodbou, v níž jsou na jedné stěně instalovány holografické projektory. Každý projektor promítá na přesně určené místo na druhé stěně chodby.

Žádné dva promítané obrazy na stěně se sice nepřekrývají a žádné dva projektory nejsou na jednom místě, ale může se stát (a vzhledem k návrhům uměleckého designéra na uspořádání se stává dost často), že se paprsky nějakých dvou projektorů cestou kříží. A aby u holografických projektorů nedošlo k nechtěné interferenci a rozmazání obrazu, musí v takovém případě pracovat oba projektory na jiné frekvenci.

Technik, který už tak má bolení hlavy z návrhů designéra, zároveň chce, aby bylo použito co nejméně frekvencí, protože je to jednodušší na údržbu. Když se projektory nekříží, mohou mít klidně stejnou frekvenci. Ale žádné dva křížící se projektory nemůžou pracovat na stejné frekvenci.

Navrhněte tedy postup, jak co nejrychleji určit, na jakých frekvencích mají pracovat které projektory, tak, aby počet použitých frekvencí byl co nejmenší. Od designéra dostanete pouze rozmístění promítaných obrazů na stěně (například očíslované podle pořadí odpovídajících projektorů na druhé stěně).

Příklad: pro vstup `3 1 2 5 4` se paprsky na stejné frekvenci nekříží například při rozdělení: frekvence 1 – projektory 1, 2, 4, frekvence 2 – projektory 3, 5. Viz obrázek:



„Tak vidíte, že to šlo. A vypadá to pěkně.“ usmál se John na designéra, když mu konečně vymluvil některé jeho šílenější nápady s umístěním holografických projektorů. Rozloučili se a John se vydal dál, až úplně dozadu celého prostoru připravované expozice. Tam sídlila výstava netradičních výstupních zařízení.

Na podstavci u vstupu stál Braillský řádek. Jak říkal popisek, tato pomůcka pro nevidomé mohla zobrazovat až 80 znaků v Braillově písmu. Zobrazení jednotlivých znaků měly na starost většinou malé elektromagnety, které nadzvedly odpovídající výstupky. Nevidomý tedy mohl procházet jakýkoliv textový obsah na obrazovce a na Braillském řádku si ho přečíst.

Další zajímavostí byla ukázka technologie, která se začala rozvíjet na přelomu prvního a druhého desetiletí 21. století, takzvaných generátorů vůně. Vstupní data pro tyto věci mohla pocházet buď ze speciálního programu, nebo z chemického čidla na druhé straně komunikační linky. Generátor pachu pak z několika

základních chemických vůní (podobně jako monitor z několika základních barev) poskládal pach nebo vůni, která se tomu co nejvíce blížila.

„Něco takového bych si domů asi nepořídil," řekl John a pak leknutím uskočil, neboť se hned vedle něj náhle rozsvítila velká obrazovka plná spousty znaků. „Promiňte pane, jenom tady procházíme stará záznamová média a koukáme, co by šlo zobrazit na těchto obrazovkách, hned to dám pryč."

„Počkejte!" vyhkrl John, v němž se probudila zvídavost. „Vždyť to je kus nějakého starého programového kódu, k čemu asi... hmm... počkejte, už je mi to asi jasné. Ale proč je to napsané takhle neefektivně?"

24-4-6 Starý kód — 9 bodů

Pracovníci muzea počítačů nalezli na jednom starém disku následující kód. Zkuste zjistit, co vlastně kód dělá, a zamyslete se nad tím, jestli by nešel přepsat nějak efektivněji.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_H 1000000
#define MAX_V 1001

typedef struct {
  int x, y;
}H;
int N, M;
H h[MAX_H];
int v[MAX_V][MAX_V];
int p[MAX_V];
int f[2*MAX_V];
short b[MAX_V];

int main() {
  scanf("%d%d", &N, &M);
  if (N>MAX_V || M>MAX_H) {
    printf("Chybny vstup.\n");
    return 1;
  }
  for (int i=0; i<M; i++) {
    int x, y;
    scanf("%d%d", &x, &y);
    if (x>N || x<1 || y>N || y<1) {
      printf("Chybny vstup.\n");
      return 1;
    }
```

```
      h[i] = (H){x, y};
      v[x][p[x]++] = y;
      v[y][p[y]++] = x;
    }
    printf("Vysledny seznam:\n");
    for (int k=0; k<M; k++) {
      int a = 0;
      int z = 0;
      for (int i=1; i<=N; i++)
        b[i] = 0;
      b[h[k].x] = 1;
      f[z++] = h[k].x;
      while (a<z && b[h[k].y]==0) {
        int q = f[a++];
        for (int i=0; i<p[q]; i++) {
          if (b[v[q][i]]==0 && !(q==h[k].x
               && v[q][i]==h[k].y)) {
            f[z++] = v[q][i];
            b[v[q][i]] = 1;
          }
        }
      }
      if (b[h[k].y]==0)
        printf("%d %d\n", h[k].x, h[k].y);
    }
    return 0;
  }
```

Vedle něj byl objeven druhý, podobný kód, u kterého byla poznámka, že až na nepodstatný rozdíl ve čtení vstupu dělá to samé:

```
import sys
MAX_H = 1000000
MAX_V = 1001
v = []; p = []; f = []; b = []; h = []
for j in range(MAX_V+1):
  v.append([]); b.append(0); p.append(0)
for j in range(2*MAX_V):
  f.append(0)

N, M = raw_input().split(' ')
N = int(N); M = int(M)
if N > MAX_V or M > MAX_H:
  sys.exit("Chybny vstup.")
```

```
for i in range(M):
  x, y = raw_input().split(' ')
  x = int(x); y = int(y)
  if(x>N or x<1 or y>N or y<1):
    sys.exit("Chybny vstup.")
  h.append((x, y))
  v[x].append(y); p[x] += 1
  v[y].append(x); p[y] += 1

print "Vysledny seznam:"
for k in range(M):
  a = 0; z = 0
  for i in range(1,N+1):
    b[i] = 0
  b[h[k][0]] = 1
  f[z] = h[k][0]; z += 1
  while(a<z and b[h[k][1]] == 0):
    q = f[a]; a += 1
    for i in range(p[q]):
      if b[v[q][i]] == 0 and not (
        q == h[k][0] and v[q][i] == h[k][1]
      ):
        f[z] = v[q][i]; z += 1
        b[v[q][i]] = 1

  if b[h[k][1]] == 0:
    print h[k][0], h[k][1]
```

Když John dozkoumal kód, pozval ho ten stejný technik, který ho vylekal, dál. „Nechcete se podívat na ty staré helmy virtuální reality, co jsme zrovna vybalili?“

První helmy virtuální reality se začaly objevovat koncem 80. a počátkem 90. let. V podstatě šlo o helmu se dvěma malými obrazovkami, pro každé oko jedna. Ve spojení ještě například s rukavicemi poskytujícími hmatovou odezvu se tak člověk mohl ponořit do světa virtuální reality.

Helmy se ale díky své ceně a váze nikdy příliš neuplatnily. Na přelomu prvního a druhého desetiletí nového tisíciletí jejich funkci částečně převzaly technologie trojrozměrných brýlí a odpovídajících obrazovek, které byly mnohem dostupnější než drahé helmy.

„Nechcete si třeba vyzkoušet nějakou starou hru?“ zeptal se technik a aniž by čekal na odpověď, spustil hru s nejnápadnějším názvem.

24-4-7 Čtvercové bombardování — 13 bodů

⚠ Představte si, že máte velké město a chcete ho srovnat se zemí. Třeba protože se vám už nelíbí a chcete místo starých domů postavit nové, moderní.

Máte k dispozici bombardér se speciální demoliční bombou. Na demoličních bombách je zajímavé to, že jsou pečlivě sestrojené tak, aby srovnaly se zemí pouze přesně danou čtvercovou oblast. A protože jste nakoupili kvalitní demoliční bomby, bouchají navíc pouze směrem na východ a jih.

Tedy pokud shodíte demoliční bombu s rázem D do místa $[x, y]$, budou zdemolovány všechny budovy ve čtverci vymezeném body $[x, y]$ a $[x + D, y + D]$, ale nic jiného. Protože ale chcete demolovat efektivně, bude lepší si vše předem propočítat.

Pro zjednodušení budeme budovy považovat za body – na vstupu dostanete jejich počet $B < 250\,000$ a jejich souřadnice $[x_i; y_i]; -10^9 < x_i, y_i < 10^9$. Zkuste vymyslet program, kterého se budete moci ptát, kolik budov bude zbouráno, když do místa $[x, y]$ hodíte bombu s rázem $0 < D < 2 \cdot 10^9$. Všechny souřadnice jsou celočíselné.

Počítejte s tím, že těchto dotazů bude program dostávat řádově statisíce, takže se pokuste, aby odpovědi na dotazy byly rychlé i za cenu delšího úvodního předpočítání.

„To teda byla hra!“ smál se John, když sundaval helmu. „Děkuju.“

Technik s úsměvem převzal helmu a podíval se na obrazovku, kde svítilo „Úroveň New York dokončena, přejete si pokračovat?“

I nadešel poslední den před otevřením, do slavnostního přestřižení pásky zbývalo již jen několik hodin a vše konečně vypadalo připravené. Projektory svítily, panel se žárovičkami blikal, vozíky ve skladu opět jezdily a sledovací systém exponátů spokojeně předl.

Nestrhne se na poslední chvíli ještě nějaká pohroma? Bude konečně dopřáno Johnovi přestřihnout v klidu slavnostní pásku? Prozradím vám, že ano. Co se ale stane několik sekund po přestřižení pásky, to je už jiný příběh. Možná někdy příště...

Od klávesnice se s Vámi loučí váš dnešní průvodce muzeem počítačů

Jirka Setnička

Pátá série

„Dobrý den, pane, máte tu jedno rekomando. Prosil bych jeden podpis... výborně, pěkný den přeju!“

Zvláštní – doporučeně už mi delší dobu nikdo nic neposlal, vynechám-li soudní obsílky... Tohle ani nevypadá úředně.

Milý příteli,

už je tomu dlouho, kdy jsem se naposledy ozval. Nezapomněl jsem na Tebe – jen jsem měl poslední dobou hodně práce kvůli té naší chatě. Před časem jsi nám říkal, že se máme ozvat, až budeme potřebovat pomoc – tak Ti tedy píšu.

Chata už je skoro hotová, jen bychom potřebovali pomoc s jedním výkopem. Mohl by ses někdy stavit v jižních Čechách? Sešli bychom se na obvyklém místě, dopravu na chatu zajistím.

Měj se pěkně,

Edo

Hmm... Mohl by to být normální dopis. Nebýt toho, že žádného Edu neznám, a tím méně jeho chatu. Vyvolalo to ve mně značnou nostalgii. Je tomu už hodně dávno, co jsem dostal něco podobného – podobné šifry už dnes chodí zásadně oknem.

Ani nevím, kdo dostal ten báječný nápad používat ke komunikaci poštovní holuby. Klasickou poštu i telefony od nepaměti kontroluje StB. Veškeré zprávy jsme museli šifrovat velmi podivným způsobem, aby nebudily podezření. A dostat cokoliv za hranice bylo až donedávna prakticky nemožné.

To všechno se s příchodem poštovních holubů změnilo. Jsou podstatně rychlejší, než klasická pošta. Ale hlavně – StB není schopná je jakkoliv kontrolovat. Takže si můžeme dovolit zprávy posílat prakticky nešifrovaně. A od dob RFC 1149[7] *ani nemusíme řešit nejednoznačnosti datových paketů.*

I holubi však mají spoustu chyb – například jednosměrnost přenosu. Takový poštovní holub umí jen jednu věc – ať ho dovezete kamkoliv, vždycky trefí domů. Pokud potřebujete poslat zprávu někam jinam, máte prostě smůlu... anebo musíte použít prostředníka (nebo jiného holuba).

24-5-1 Holubí centrála — 9 bodů

Typickým problémem bývalo svolávání srazu. Sraz může vyhlásit libovolný člen organizace, jen musí zajistit, že se informace o času a místě dostane ke všem ostatním.

[7] `http://www.faqs.org/rfcs/rfc1149.html`

Vás by zajímalo, kteří členové mohou sraz vyhlásit. Dostanete seznam členů včetně poštovních holubů, které mají jednotliví členové u sebe. Každý poštovní holub má určeno, ke kterému členovi doletí. Máte vypsat ty členy organizace, od kterých vede spojení pomocí holubů ke všem ostatním. Takové spojení samozřejmě může vést přes prostředníky.

Pokud má například organizace 6 členů (s čísly 1 až 6), člen číslo 3 má holuby letící ke členům 1, 2 a 5, člen 5 holuby pro 3 a 6, člen 6 umí poslat zprávu členovi 4 a ostatní nemají žádného holuba, je správným řešením vypsat členy 3 a 5.

Naše ornitologické oddělení nedávno vymyslelo i efektivní broadcasting (všesměrové vysílání): stačí využít hejna labutí. Labutě jsou při přesunu dobře vidět. Navíc se vyskytují ve dvou barvách: černé a bílé.

24-5-2 Labutí broadcasting — 11 bodů

Zpráva pro broadcast se sestavuje následujícím způsobem: Nejprve ji převedete do posloupnosti nul a jedniček, poté seřadíte labutí hejno. Každá labuť odpovídá jednomu bitu. Pokud je bit nulový, zařadíte černou labuť; pokud je jedničkový, zařadíte bílou. Takto seřazené hejno poté vypustíte na oblohu a doufáte, že poletí správným směrem.

V labutím hejnu má první labuť nejtěžší úkol – rozráží vzduch. Proto se labutě postupně střídají. Vždy, když je první labuť unavená, zařadí se na konec hejna, přičemž vedoucí pozici převezme labuť za ní.

Ornitologické oddělení dosud nevymyslelo vhodný přenosový protokol; proto se obracíme na vás.

Máte vymyslet co nejefektivnější přenosový protokol – víte, že při poslání N bitů příjemci dorazí N stejných bitů, ale náhodně rotovaných. Když tedy odešlete 1101, tak může přijít 1101, 1011, 0111 a 1110.

Vymyslete, jak tímto způsobem odeslat zprávu o K bitech, aby na její zakódování bylo potřeba co nejméně reálně odeslaných bitů a stále byla jednoznačně dekódovatelná.

Příklad: Pro $K = 1$ je řešení triviální, vyšleme tu správnou jednu labuť. Pro $K = 2$ vyšleme bity tak, jak jsme je dostali, a druhý z nich zopakujeme. Tedy pokud chceme odeslat xy, tak odešleme xyy. Na zprávu délky 2 jsme tedy spotřebovali 3 bity. Pro $K = 3$ potřebujeme 5 labutí.

5 bodů dostanete, pokud vymyslíte efektivní protokol pro $K = 8$.

Nostalgie bylo dost. Asi bych nás měl trochu představit, když už jsem to nakousl... jsem členem jedné organizace, která má za svůj cíl postavit tajnou necenzurovanou telefonní linku z ČSSR do Rakouska – snažíme se vybudovat ro-

zumné spojení se sítí EARN.[8] *Což se samozřejmě nelíbí vládě ani StB – vznikl by nekontrolovatelný komunikační kanál se zahraničním disentem, navíc podstatně rychlejší než holubi a labutě dohromady. Takže pracujeme tajně.*

Činnost organizace je pochopitelně časově i organizačně velmi náročná.

24-5-3 Struktura organizace — 11 bodů

Abychom minimalizovali riziko odhalení, rozhodli jsme se pro zvláštní organizační strukturu. Každý člen zná jen své podřízené a svého přímého nadřízeného, od kterého dostává rozkazy. Podřízených může být i víc, nadřízeného má každý jediného, s výjimkou právě jednoho velkého šéfa, jenž už nadřízeného nemá. Nikdo není nadřízeným sám sobě, a to ani nepřímo.

Do akce je posílána vždycky skupina členů. Ti mezi sebou potřebují komunikovat, proto skupina musí zůstat souvislá. To znamená, že po odeslání do akce musí každý člen být schopný odeslat zprávu všem ostatním. Zprávy se samozřejmě mohou předávat pouze mezi známými, tedy mezi podřízenými a nadřízenými.

Vás by zajímalo, kolika způsoby můžeme vytvořit libovolně velkou skupinu, kterou pošleme do akce.

Například pokud máme 3 zaměstnance, přičemž zaměstnanec číslo 3 je přímý nadřízený zaměstnanců 1 a 2, tak máme dohromady 6 možností, jak skupinku vytvořit: $\{1\}$, $\{2\}$, $\{3\}$, $\{1,3\}$, $\{2,3\}$, $\{1,2,3\}$. Zaměstnance 1 a 2 vyslat nemůžeme, protože pak by byli naprosto oddělení.

7 bodů dostanete, pokud úlohu vyřešíte pro strukturu tvořící úplný binární strom. Zde má každý dva nebo žádného podřízeného, navíc všichni bez podřízených „jsou si rovni“ – mají nad sebou stejný počet nadřízených.

Tentokrát to vyšlo na mě. Abych to nezakecal, to rekomando znamená zahájit stavbu, sraz ve městě, kde by chtěl žít každý. Zajištění dopravy znamená, že nemusím shánět bagr.

Tak už jen zabalit několik kilometrů kabelu a hurá na cestu!

Kdo jste někdy viděli sraz členů tajné organizace na veřejném místě, jistě dáte za pravdu, že to není nic jednoduchého. Nemůžete si prostě vzít transparent hlásající „Hledám své tajné kolegy!“ a stoupnout si doprostřed náměstí.

Místo toho je nutné mít předem domluvený způsob, jak se poznat. Samozřejmě dostatečně nenápadný. My většinou využíváme zeměpisných vlastností dané lokace.

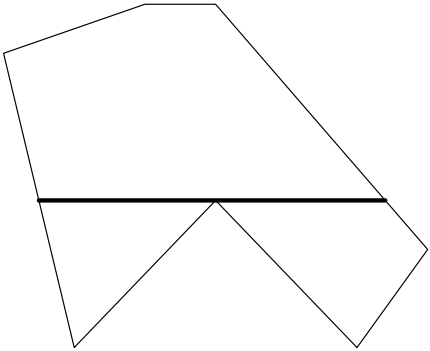
Protentokrát jsme zvolili sraz na západním konci nejdelší úsečky vedoucí ve východozápadním směru, kterou je možné na náměstí najít. Mapu máme. Pomůžete nám s hledáním takové úsečky?

[8] `http://en.wikipedia.org/wiki/European_Academic_Research_Network`

24-5-4 Sraz na náměstí — 11 bodů

Na vstupu dostanete (ne nutně konvexní) mnohoúhelník představující náměstí, zadaný například posloupností vrcholů. Máte vypsat nejdelší úsečku ve vodorovném směru, která je v mnohoúhelníku celá obsažena.

Příklad:



Tučně je vyznačena hledaná úsečka.

6 bodů dostanete, pokud vyřešíte úlohu pro konvexní náměstí.

Nakonec jsme se našli a snad nás přitom nikdo neviděl.

Na podobně dlouhých linkách se hodně projevuje rušení, zejména proto, že nemáme finance na dostatečně stíněné kabely – ty jsou moc drahé. Proto je občas nutné kabel přerušit a umístit stanici, která detekuje příchozí signál a předá ho dál.

Polohy těchto zesilovacích stanic jsou dány částečně technickými limity a rušením signálu, hlavně však tím, kde všude máme svoje lidi a elektřinu.

Řezání kabelů (a připojování koncovek) také není jednoduché. Pokud to jde, snažíme se kabely nařezat na příslušné délky pěkně v klidu někde v továrně.

24-5-5 Řezání kabelů — 9 bodů

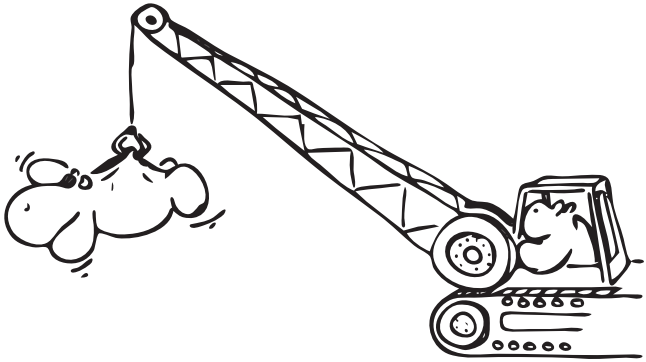
Máte dlouhý kabel a chtěli byste ho co nejrychleji nařezat na kusy o délce $k_1, k_2, \ldots, k_n$. Kabel má celkovou délku $K = k_1 + k_2 + \ldots + k_n$. Je namotaný na cívce, před řezáním ho musíte celý odmotat a přeměřit. Při řezání rozdělíte jednu souvislou část kabelu na dvě menší o příslušných délkách. Odmotané kusy jsou dlouhé, takže je musíte ihned namotat na jinou cívku.

Nejvíce času zabere neustálé namotávání a smotávání, samotné řezání lze zanedbat. Každý řez tedy trvá tak dlouho, jaká je délka řezaného kusu kabelu.

Na vstupu dostanete počet úseků a jejich délky. Máte vypsat takové pořadí řezů, které zabere co nejméně času.

Příklad: Pro úseky délek 3 3 3 3 8 je optimálním řešením posloupnost řezů $20 \to 8 + 12$, $12 \to 6 + 6$, $6 \to 3 + 3$, $6 \to 3 + 3$.

Po nařezání kabelů jsme se dali do stavby. Občas se nás místní ptali, co to vlastně děláme. Na podobné dotazy jsme připraveni – hlavně proto, že někdy provádíme neohlášené výkopy na cizích zahradách. Vždy se stačilo vymluvit na tajnou linku od Správy pošt a telekomunikací stavěnou pro armádu – tím jsme úspěšně odradili jak vojáky, tak „kolegy" od SPT. Majitele pozemků jsme typicky odbyli slovy „Když nesledujete vývěsní desku, tak se nedivte." Než si to stihli ověřit, už jsme byli pryč.



Brzy jsme dorazili k hraničnímu pásmu, tady si nemůžeme dovolit být tak drzí. Našli jsme jedno slabší místo, kudy se dostaneme zhruba kilometr od hranic bez jakéhokoliv rizika odhalení. Má to však jeden problém – po celé délce je minové pole.

24-5-6 Minové pole — 13 bodů

Taková typická hraniční mina má určenou oblast, kde detekuje pohyb – když se sem něco dostane, tak vybouchne a celou ji zničí. Miny byly pokládány do čtvercové sítě, navíc při výbuchu zničí pouze obdélníkovou oblast kolem sebe. Minové pole je obdélníkové.

Máte detektor kovů, víte tedy, kde se jaká mina nachází, a z jejich velikostí víte, jakou oblast daná mina kontroluje. Pro každé pole čtvercové sítě by vás zajímalo, kolik min vybouchne, když na něj šlápnete.

Na vstupu dostanete rozměry minového pole (počet řádků a počet sloupců čtvercové sítě: R a S) a seznam min spolu s oblastí, kterou daná mina kontroluje (zadanou levým horním a pravým spodním rohem).

Pokud obdélník začíná a končí na stejném řádku, resp. sloupci, tak je jedno políčko široký, resp. vysoký.

Vypište matici o rozměrech $R \times S$, kde je na pozici (i, j) uvedeno číslo udávající počet min, které vybouchnou při šlápnutí na toto pole.

U prvních 5 vstupů bude zadané pole jednorozměrné – vyřešením získáte 7 bodů.

Explodující minové pole jsme úspěšně nechali za sebou. Vzniklé krátery se dají skvěle využít pro položení kabelu!

„Taky sis vzpomněl na hru Čtvercové bombardování?“

„Pst! Někoho slyším!“

Mezi námi a hranicí zbývá jen pohraniční stráž. Teď už se nevzdáme! Naštěstí máme na jejich velitelství své lidi a známe denní rozpisy hlídek – ukrýt se tak, aby nás nenašli, není těžké. Dokonce jsme zvládli i zamaskovat výkopy.

Kousek za hranicí nás už netrpělivě očekávali rakouští kolegové. Spojili jsme natažené kabely a pak nás kolegové odvezli do Linze na svou centrálu. Zároveň jsme morseovkou poslali prvních několik krátkých zpráv, abychom ověřili, že naše linka funguje.

Byla v pořádku! Rakouští kolegové okamžitě začali posílat informace, které se k nám jinak nedostanou.

Vypadá to, že fyzická část spojení je hotová. Ještě zbývá vyřešit softwarovou část, abychom mohli propojit počítače a zbavili se zdlouhavé práce telegrafistů. K tomu se nám bude hodit pomoc zkušeného odborníka.

24-5-7 Cesta přes hranice — 13 bodů

Odborník sídlí v německém Pasově. Potřebujete se k němu dostat a následně ho dopravit do Prahy. Cestou budete muset několikrát překročit hranice. To je menší problém, protože nemáte platný pas. Máte však několik výmluv, které můžete při průjezdu použít – abyste zabránili odhalení, můžete každou použít pouze jednou. Samozřejmě jich máte jen konečně mnoho a neměli byste jimi plýtvat, aby vám něco zbylo i na příště. Na druhou stranu si vás celníci zapamatují a při každém dalším průjezdu stejnou celnicí vás už kontrolovat nebudou.

Na vstupu dostanete mapu oblasti – seznam měst a cest mezi nimi, včetně vzdáleností. Dále u každého města víte, jestli je v něm celnice, nebo ne. Taky dostanete pozici Linze (zde začínáte), Pasova (tam se musíte zastavit) a Prahy (tam musíte skončit).

Nalezněte a vypište nejefektivnější cestu. Primárně se snažíte minimalizovat počet průjezdů celnicemi, sekundárně ujetou vzdálenost.

7 bodů získáte za vyřešení úlohy pro zapomnětlivé celníky. Ti si váš průjezd celnicí nezapamatují, takže při každém dalším průjezdu jejich celnicí musíte použít novou výmluvu.

Cestou do Prahy bylo jasně poznat, že se něco děje. Oblohu křižovala černobílá labutí hejna, noviny byly plné zahraničních informací a málem jsme srazili dva poštovní holuby.

Očividně si toho všimla i StB – tolik silničních kontrol jsme už hodně dlouho nepotkali. Ale je vidět, že absolutně netuší, co se stalo.

Povedlo se!

Radim „Rumcajz“ Cajzl

Herní seriál

Lukáš Lánský & Pavel „Paulie“ Veselý

24-1-8 Pojďte pane, budeme si hrát — 14 bodů

Letos se bude seminářem jako červená nit proplétat seriál o hrách a jejich matematickém a výpočetním řešení. To důležité, co si z něj můžete odnést, je přehled o tom, jakým způsobem současné počítače získávají náskok před lidským rozumem a v jakém vztahu mohou koexistovat chytrá pozorování a hrubá výpočetní síla.

Definovat si, co znamená *hra,* zní nanejvýš otravně, ale je to pojem tak obecný, že s nějakým vymezením začít musíme. Nebo od nás čekáte, že budeme studovat, jak počítačem řešit schovávanou?

- Mějme právě dva hráče.
- Hráči se střídají v tazích.
- Každý tah se vybírá z konečné sady možností. Piškvorky tedy budeme nazývat hrou jen pro předem omezenou velikost čtverečkovaného papíru.
- Oba hráči znají celou informaci o hře, takže žádný z nich neskrývá karty.
- Průběh hry je závislý pouze na tazích hráčů, takže neexistuje náhoda a nehází se kostkou.

Můžeme začít!

Matematika funguje

Přestože víme, že počítače jsou v šachách lepší než lidé, neplatí, že by šachy byly vyřešená hra – neví se totiž, že by nějaká strategie zaručovala vítězství proti libovolnému oponentovi.

Existují hry, jako je *anglická dáma,* které vyřešené jsou, ale tak nějak „suše“. Máme v nich strategii, která zaručí, že nikdy neprohrajeme, nejde však o elegantní matematický nápad, jako spíš o velmi dlouhý seznam (či spíš strom) pravidel.

Vzhledem k tomu, jak arbitrární jsou pravidla oblíbených her, nedá se ani čekat, že by pro ně někdo někdy takový hezký matematický nápad našel. Existují ale hry *matematické.* Říká se jim tak, protože mají pravidla formulovatelná v řeči matematiky tak snadno či příznivě, že očekáváme, že by krásná řešení mít mohly.

Jednu takovou matematickou hru si ukážeme. Překvapivě, tuto hru lidé občas stále hrají – a hráli dlouho předtím, než byla jako důležitá matematická hra rozpoznána.

Mějme tři hromádky nerozlišitelných žetonů. Hráči se střídají v tom, že z jedné hromádky seberou a zahodí libovolné nenulové množství žetonů. Prohrává ten, na kterého žádný žeton nezbude.

Když si tuto hru zkusíte zahrát, zjistíte, že úplně triviální není. Má však elegantní matematické řešení, které nám dává rychlou metodu, jak určit, jestli má hráč na tahu zajištěnou výhru a pokud ano, jak by měl táhnout.

Zjednodušme si situaci a redukujme počet hromádek na dvě. Jak hrát tuto hru je nasnadě, ale rozmyslete si to. Pokušení číst dál, aniž byste řešení našli, je třeba odolat, protože spoilery v matematice hrají stejně zápornou roli, jako spoilery u filmů s důležitým zvratem.

Takovou hru má vyhranou první hráč na tahu právě tehdy, je-li na hromádkách rozdílný počet žetonů. Táhnout bude tak, že sebere z početnější hromádky tolik žetonů, aby počet dorovnal. Protivníkovi tak nezbude, než rovnost opět porušit.

To se bude opakovat do té doby, než hráč, co dostává situaci s rozdílným počtem žetonů, dostane jednu z hromádek prázdnou – vyhraje pak sebráním celé druhé hromádky. Hráči, co dostává situaci s tím samým počtem žetonů, se něco takového evidentně stát nemůže – jedním tahem nikdy dvě neprázdné hromádky neodstraní.

Dobře tedy! Při třech hromádkách budeme hledat podobné *smutné stavy* hry,

- do nějakého z nich bude mít jeden hráč možnost druhého vždy uvrhnout z každého stavu, který smutný není,
- které budou zahrnovat prázdnou (prohrávací) pozici,
- a všechny tahy ze smutných pozic vedou do pozic, které smutné nejsou.

Řešením je vyjádřit si počty žetonů na hromádkách jako binární čísla a provést po číslicích jejich XOR (ten jsme milou náhodou zavedli už v úloze 24-1-1). Stav jako smutný označíme tehdy, vyjde-li nám nula.

Úkol 1 [5b]: Ověřte, že taková definice splňuje tři požadavky, které jsme měli.

Uvědomte si, že tímto pozorováním získáme strategii, jak hru se třemi hromádkami vyhrát pokaždé, když nejsme ve smutném stavu, kdy nad námi naopak bude moci vždy vyhrát protivník.

Můžeme si také všimnout, že popsaná strategie pro dvě hromádky je ve skutečnosti ten samý postup. Ještě zajímavější je, že nám strategie funguje pro libovolný konečný počet hromádek!

Generování možných tahů

V druhé části seriálu se zaměříme na hry, které efektivně vyřešit neumíme. Zřejmě se nebudeme snažit, aby za nás počítač *pochopil,* jaké strategie jsou dobré, protože počítač je v chápání opravdu nemožný. Co mu naopak velmi jde, je procházení všech možností.

Máme výchozí situaci a chceme udělat první půltah (půltah je zahrání jednoho hráče a tah je půltah hráče společně s následujícím půltahem protihráče).

Můžeme si spočítat, jak bude vypadat deska po každém z možných půltahů, a uvažovat nad tím, je-li to pro nás dobrá pozice. Asi ale tušíte, že z toho mnoho nezjistíme. Potřebujeme rozmýšlet na více tahů dopředu.

Dobře. Nagenerujeme všechny možné situace desky po 8 půltazích. Třeba rekurzivně:

Funkce *generuj* (*pozice*, *hloubka*, kdo je *na tahu*):

Pokud je *hloubka* $= 0$ nebo je pozice vyhrávající či prohrávající:
Vypiš *pozice*.

Pro všechny možné tahy t hráče, který je *na tahu*,
z *pozice*:
generuj (pozice po tahu t, *hloubka* $- 1$, druhý hráč)

Co nevidíme, je tam pozice, ve které jsme vyhráli!

Slavnostně si vybavíme, jaká sekvence půltahů vede do této pozice a zahrajeme první z nich. Ale co se nestalo, protivník se svou brzkou záhubou nesouhlasí a hraje jinam, do pozice, která pro nás nevypadá dobře.

Minimax aneb „O tom nerozhoduješ!“

Byli jsme nerozvážní. Nemůžeme si jen tak vybrat, kam se dostat – musíme počítat s tím, že naše možnost ovlivňovat hru je jaksi poloviční a navíc že protivník je inteligentní a druhá polovina tahů povede proti našemu zájmu.

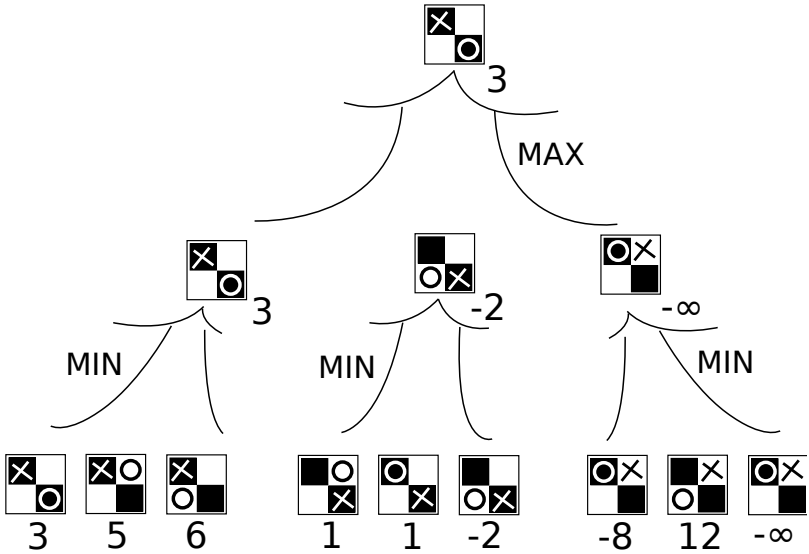
Nalezení prostého maxima z nalezených pozic se tedy vyvarujeme. Využijeme zvoleného rekurzivního způsobu generování tahů a budeme si vracet maxima tam, kde máme volbu, a minima tam, kde volbu nemáme a kde očekáváme, že půjde protihráč proti nám.

Funkce *generuj* (*pozice*, *hloubka*, kdo je *na tahu*):

Pokud je *hloubka* $= 0$ nebo je pozice vyhrávající či prohrávající:
Vrať *pozice*.

Zavedeme prázdný seznam *možnosti*.
Pro všechny možné tahy t hráče, který je *na tahu*, z *pozice*:
Přidej do *možnosti* hodnotu
generuj (pozice po tahu t, *hloubka* $- 1$, druhý hráč).

Pokud jsem *na tahu* já:
Vrať nejlépe ohodnocenou pozici ze seznamu *možnosti*.
Jinak:
Vrať nejhůře ohodnocenou pozici ze seznamu *možnosti*.



Tomuto algoritmu se obvykle říká *minimax*. Abychom ho mohli implementovat, potřebujeme počítači vysvětlit, jak poznat, která situace je pro nás lepší, než jiná. Obvyklou volbou je napsat funkci, která pozici přiřadí hodnotu z množiny reálných čísel obohacených o hodnoty $+\infty$ a $-\infty$, které slouží jako indikace „výhry“ a „prohry“. Vysoká kladná čísla budou znamenat, že je pozice velmi dobrá, záporná, že je pozice slabá.

Kvality této *ohodnocovací funkce* budou do značné míry ovlivňovat kvalitu výsledného algoritmu. Uvědomme si, že minimax zaručuje, že se dostaneme do relativně dobře ohodnocené situace, pokud si ale ohodnocovací funkce spletla dobrý skutečný stav věcí se špatným, prohráváme.

Pokud bychom mohli minimax spustit neomezeně půltahů hluboko, do konce každé hry, stačilo by, aby ohodnocovací funkce poznala vítězství a prohru, a náš algoritmus by hrál nejlépe možně – pokud by mohl vyhrát, vyhrál by. To však není realistické očekávat – s prohledáváním o každý půltah hlouběji se běh násobně zpomalí.

Ohodnocovací funkce může pozici zkoumat podrobně a strávit nad tím hodně času, minimax ale potom nestihne postoupit do tak hluboké úrovně, jako kdyby bylo ohodnocování pozic povrchní a nespolehlivé. To, jak pečlivé zkoumání stojí za to zvolit, záleží především na povaze řešené hry.

Druhá důležitá funkce generuje možné půltahy. Ve většině her je jen několik málo půltahů rozumných a u části nerozumných půltahů to můžeme algoritmicky rozeznat ihned, aniž bychom pouštěli minimax.

Kupříkladu v piškvorkách nemá cenu hrát příliš daleko od bojiště. Nemůžeme si být jisti, jestli některá optimální strategie s takovým dalekým tahem nepočítá, ale ani ve hrách velmi dobrých hráčů se nic takového nevyskytuje a my si tak jen na základě tohoto pozorování můžeme dovolit násobně zmenšit počet vygenerovaných tahů.

Dobře zvolit, které situace z generátoru pouštět a které ne, je důležité rozhodnutí, protože jím omezujeme, jak moc se nám bude strom volání větvit, tedy (opět) jak hluboko budeme moci propočítávat.

Úkol 2 [7+2b]: Napište ohodnocovací funkci a generátor možných tahů pro hru šestvorky na desce 15×15. Tato hra se od běžných piškvorek liší ve dvou „drobnostech“:

- Vyhrává linie šesti, ne pěti značek.
- Hráči pokládají hned dvě své značky ve svém půltahu místo jedné, s výjimkou úplně prvního tahu začínajícího hráče, při kterém se pokládá značka toliko jedna. Je to hezké pravidlo, protože činí hru vyrovnanější.

Od 21. srpna (tedy od skončení nulté série) do uzávěrky série této bude na stránkách semináře aréna, ve které se budou vaše funkce zasazené do minimaxového algoritmu bít. Dozvíte se tam i technické detaily. Zde uvádíme jen,

- že čas na vyhodnocení jednoho tahu bude zhruba deset sekund,
- že pozici budete dostávat jako obyčejné dvojrozměrné pole a že žádné jiné informace nebudete smět použít (nepůjde si ukládat data mezi ohodnoceními)
- a že programovacím jazykem bude Python.

S účastí není potřeba zvlášť spěchat, protože doba, po kterou se bude algoritmus soutěže účastnit, závěrečné vyhodnocení sama od sebe neovlivní. Přirozeně je dobrý nápad zkusit si včas, jak si na tom stojíte, a tak se v případě neuspokojivého výsledku motivovat k další práci.

Všechna řešení, která překonají námi připravenou dvojici funkcí, dostanou sedm bodů. Zbylé dva body rozdělíme podle umístění v tabulce.

Lukáš Lánský

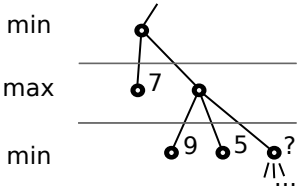
24-2-8 Alfa-beta ořezávání a piškvorky — 14 bodů

V prvním dílu jsme si představili minimaxový algoritmus. Zjistili jsme také, že je možno zrychlovat jeho běh tím, že neuvažujeme některé možné tahy vedoucí z aktuální pozice. Za to však platíme možným přehlédnutím (nepravděpodobně) optimálního tahu.

Pro jednoduchost si označme hráče táhnoucího v maximalizačních úrovních jako Max (to je ten, pro něhož hledáme nejlepší tah) a jeho soupeře Min (ten táhne v úrovních, kde se vybírá minimum hodnot ze synů).

Alfa-beta ořezávání je úprava minimaxu založená na jednoduchém pozorování, které znatelně urychluje průměrnou dobu běhu prohledávání, aniž by ovlivnilo výsledek.

Uvažme následující situaci:



Potřebuje stroj znát hodnotu pozice s otazníkem, aby mohl vrátit výsledek? Nikoliv, protože hodnota pozice v nadřazeném vrcholu už větší nebude, ať připočítáme cokoliv.

Proč tomu tak je? Když bude na pozici s otazníkem více než 5, tak minimem v této úrovni bude 5; v opačném případě bude minimum menší než 5.

Hodnota pozice v nadřazeném vrcholu tudíž bude maximálně 5, ale to už je méně než 7 – hodnota vedlejší pozice – takže pozice s otazníkem už rozhodování nikterak neovlivní.

Uvědomme si, jak je v dané pozici takové pozorování cenné – skutečně můžeme přestat počítat (říká se *zaříznout*) celou *větev*. Zkoumat ji by byla spousta práce.

Kdy přesně lze větev zařízнout? Pokud se nacházíme v minimalizační fázi a v libovolném vrcholu v maximalizační fázi na cestě ke kořeni prohledávacího stromu existuje už dopočítaný syn, který má vyšší hodnotu než aktuální minimum v této úrovni.

To vše obdobně pro vrcholy ve fázi maximalizační.

Nyní je vhodná chvíle přestat na chvíli číst a rozmyslet si, jestli opravdu všemu rozumíte. Zkuste si nakreslit složitější situaci a chvíli ji analyzujte.

Takové chování se příjemně implementuje za použití dvou proměnných, kterým budeme nečekaně říkat *alfa* a *beta*. *Alfa* bude maximum z dopočítaných synů v maximalizačních fázích a *beta* minimum z fází minimalizačních. S těmi pak bude stačit nově dopočítané hodnoty vrcholů porovnávat.

Na začátku zvolíme *alfu* jako $-\infty$ (Max může prohrát) a *betu* jako $+\infty$ (i Min může prohrát). V maximalizačních úrovních pak upravujeme dle hodnoty v synech *alfu*, v minimalizačních *betu*.

Jelikož *alfa* udává, jakou nejlepší hodnotu v prozkoumané části stromu může získat Max, a *beta* nejlepší nalezenou hodnotu pro hráče Min, platí v každém okamžiku, že *alfa* $<$ *beta*. Když se tedy v průběhu výpočtu dostaneme s *alfou*

nad *betu* (či s *betou* pod *alfu*), můžeme skončit prohledávání synů zkoumaného uzlu.

Znovu se zde ujistěte, jestli tušíte, proč si algoritmus může takové ořezávání dovolit.

K lepšímu pochopení může posloužit pseudokód.

```
def alfabeta(pozice, hloubka, alfa, beta, natahu):
  if hloubka == 0 or konec_hry(pozice)
    return hodnota(pozice)

  if natahu = Max:
    for p in mozne_tahy(pozice, Max):
      alfa = max(alfa, alfabeta(p, hloubka-1, alfa, beta, Min) )
      if beta <= alfa:
        break
    return alfa

  if natahu = Min:
    for p in mozne_tahy(pozice, Min):
      beta = min(beta, alfabeta(p, hloubka-1, alfa, beta, Max) )
      if beta <= alfa:
        break
    return beta
```

Alfa-beta ořezávání je nejúčinnější, jsou-li první prozkoumávané tahy nejlepší možné pro hráče, v jehož úrovni se nacházíme. Pak je možné větve pod ostatními tahy rychle zaříznout a minimax tak výrazně zrychlit. Máme-li však smůlu v seřazení tahů od nejhoršího, alfa-beta nám vůbec nepomůže.

Zkoumat na začátku dobré tahy ale samozřejmě není tak jednoduché – víme-li, které tahy jsou nejlepší, žádný minimax už nepotřebujeme. Rozhodně se nám ale vyplatí začít vymýšlet heuristiky pro generátory tahů, které se budou snažit nějak chytře předřazovat.

Často používaná je metoda *killer move.* Pamatujeme si, které tahy v jiných větvích výpočtu vedly k dobrým výsledkům, a ty pak upřednostňujeme při prohledávání.

Trochu jednodušší je *iterativní prohlubování,* při kterém prostě minimax pouštíme pro stále větší hloubku propočítávání, tj. nejdříve procházíme strom do hloubky 1, pak do hloubky 2, 3 atd. Kromě nejnižšího patra stromu používáme pro řazení tahů výsledky z předchozího výpočtu.

Neplýtváme časem, když některé části stromu prohledáváme vícekrát? Ne, protože exponenciální časová složitost minimaxu vede k tomu, že všechna hledání dohromady zpravidla zaberou méně času než to poslední, nejhlubší.

Piškvorková teorie

Poodstupme opět od vulgárního světa strojového prohledávání pozic. Matematika nám v minulém díle pomohla při analýze Nimu. Piškvorky jsou podobně jednoduše definovatelná hra – dokážeme vyřešit tu?

Záleží trochu, co přesně si pod piškvorkami představíme. Nejčastěji asi hru na neomezeném čtverečkovaném papíře, kde se střídáme v pokládání značek, přičemž výhry dosáhne hráč, který jich první vytvoří pět v řadě.

O takových piškvorkách tušíme, že v nich má výhodu začínající hráč. Jak ale velkou? Má vyhrávající strategii? Může alespoň vždy zabránit prohře? Jednoduchá, ale důležitá skutečnost zní: druhé tvrzení platí, začínající hráč v piškvorkách má *neprohrávající* strategii.

Představíme-li si pro spor, že druhý hráč má vyhrávající strategii (což je negace tvrzení „začínající hráč má strategii neprohrávající"), můžeme aplikovat přeslavný *argument o kradení strategie.*

První hráč by mohl svůj první tah zahrát libovolně na desku, zapomenout na něj (tj. nadále přemýšlet o desce, jako by tam nebyl) a potom hrát podle postulované vyhrávající strategie druhého hráče.

Onen libovolný tah by mu v žádných možných pokynech takové strategie nepřekážel – dostal-li by za úkol hrát na toto místo, učinil by prostě libovolný jiný tah a zapomněl by na ten.

Z toho však plyne, že vyhrávající strategii má jako hráč začínající (tj. zloděj), tak hráč druhý! To je spor, ke kterému jsme chtěli dospět. Vyhrávající strategie druhého hráče neexistuje, první hráč má strategii neprohrávající.

Argument je to slavný, protože je široce aplikovatelný – můžeme jej použít na všechny *poziční hry,* což je terminus technicus pro hry, ve kterých hráči trvale zabírají části herní plochy a vítězí hráč, který zabere některou z vítězných podmnožin těchto částí.

Nabízí se ohromně zajímavá otázka, má-li první hráč vyhrávající strategii, nebo je-li to remíza. Někde daleko před odpovědí na tuto otázku však možnosti současné matematiky končí. Pro modifikovanou variantu piškvorek, kde vyhrává řada osmi značek, je dokázáno, že bezchybná hra již remízou končí.

Úkol 1 [5b]: Určete a dokažte výsledek piškvorek, kde vyhrává řada čtyř značek.

Úkol 2 [9b]: Dokažte, že v piškvorkách, kde vyhrává řada devíti značek, má druhý hráč neprohrávající strategii. *Nápověda: Rozdělení do párů.*

Dobře, neomezená herní plocha může být háčkem. Pojďme hrát piškvorky na omezené desce n^d, což je d-rozměrná krychle o hraně n. Za vítěze budeme pokládat hráče, který první udělá sérii n značek ve stejném směru.

Pro $n = 3$ a $d = 2$ dostáváme známé tic-tac-toe, pro $n = 4$, $d = 3$ oblíbené trojrozměrné piškvorky...

Ani zde není situace příliš růžová a za to, že víme, že má v uvedených trojrozměrných piškvorkách začínající hráč vítěznou strategii, vděčíme dosti komplikovanému důkazu plného rozborů případů. Situace pro $n = 5$, $d = 3$ je zatím otevřená.

Lukáš Lánský a Pavel Veselý

24-3-8 Sčítáme hry s panem Conwayem — 14 bodů

Minule jsme v matematické části rozebrali některé případy piškvorek jako zástupců pozičních her (hráči obsazují pozice, dokud není jedním hráčem zaplněna jedna z výherních linií).

V dnešním díle našeho konečného seriálu navážeme na vyřešení Nimu v první sérii a představíme neformálním způsobem slavnou teorii pana Conwaye.

Pokud jste do seriálu v první nebo v druhé sérii nenahlédli, nevadí, nebude to potřeba. Připomeňme si jen pravidla Nimu: máme několik hromádek žetonů. Dva hráči se střídají v odebírání libovolného počtu žetonů z jedné hromádky. Komu nezbyl žádný žeton na odebrání, prohrál.

Zopakujme si také, jakými hrami se zabýváme:

- hrají 2 hráči, kteří se střídají v tazích,
- z každé pozice má hráč jen konečně mnoho tahů,
- bez náhody (nehází se kostkou),
- s tzv. úplnou informací (oba hráči mají všechny informace o stavu hry, takže nikdo z nich neskrývá karty),
- s tzv. nulovým součtem (zisk jednoho hráče znamená ztrátu druhého hráče).

Pro matematické řešení her se hodí, když se pozice neopakují a prohrává ten, kdo nemá tah (např. v Nimu nezbyla žádná hromádka). Neopakující se pozice zaručují, že každá partie skončí výhrou jednoho z hráčů nebo remízou (existuje-li).

Malá poznámka na úvod: v této teorii se často myslí pod pojmem *hra* konkrétní pozice v nějaké hře s danými pravidly. Proto pozici budeme značit G od anglického *game*. *Pozici* chápeme jako celý stav hry (rozložené kameny, všechny povolené tahy), ale někdy bez informace, který hráč je na tahu.

Oproti hrám jako šachy je na Nimu zvláštní, že z dané pozice mají oba hráči stejné tahy a záleží jen, kdo má právě odebírat žetony. Takovým hrám se říká *nestranné*.

Opakem jsou *zaujaté* hry – možné tahy hráčů se pro danou pozici mohou lišit, například v šachách smí bílý pohnout jen s bílými figurkami. I piškvorky jsou zaujaté, ačkoliv hráči mohou táhnout na stejná místa.

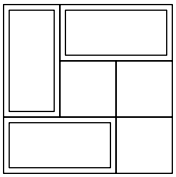
Všimněte si souvislosti s obtížností her. Nim jakožto nestrannou hru jsme kompletně vyřešili (dokážeme zjistit, kdo vyhraje a jak má táhnout), kdežto v případě šachů nebo Go je jen mizivá naděje, že by se je podařilo vyřešit (ať už s pomocí počítače nebo matematicky).

Dokonce neexistuje žádná nestranná hra, kterou by dnes bylo těžké vyřešit. Každou lze totiž převést na Nim[9] i hrát podle strategie v něm. Nestranné hry tedy nebudou ty zajímavé.

Jednou z motivací pro Conwaye k vývoji teorie zaujatých her bylo pozorování, že v koncovkách Go se hra rozpadá na několik oddělených částí, jež se ve výsledku sečtou.

Sčítání pozic si lze představit snadno na Nimu: v jedné hře mám dvě hromádky, v druhé tři, jejich součtem bude hra s pěti hromádkami o velikostech stejných jako v původních hrách. Hráč pak má možnost vybrat si, do jaké hry ze součtu bude táhnout.

Go je však velmi složité i pro dnešní počítače, které nejsou zdaleka schopné vyhrát nad profesionály. Proto si sčítání ukážeme na jednodušší zaujaté hře: *dominování*.



Máme hrací desku se čtvercovou sítí, na kterou hráči střídavě pokládají dominové kostky o rozměrech 2×1 na volná políčka. Kdo nemá tah, prohrál.

Aby se nám lépe uvažovalo, označme si hráče: jeden bude Levý a druhý pRavý (zkratky L a R pocházejí samozřejmě z angličtiny).

Ve hře pokládá levý domina svisLe, pravý vodoRovně.

Dále rozdělíme pozice do čtyřech skupin podle vítěze, přičemž nás bude zajímat, kdo vyhraje, když začne levý a když začne pravý. Tyto skupiny budeme nazývat *výsledkové třídy*:

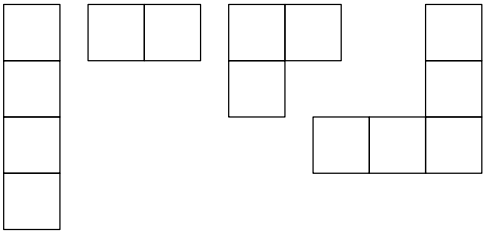
- vyhraje hráč, který bude táhnout jako první, ať už je to levý nebo pravý. Takovým pozicím budeme říkat vyhrané (pro začínajícího hráče) a jejich třídu (něco jako množinu) označovat V.
- začínající hráč prohraje, pozice je tedy prohraná. Třídu prohraných pozic budeme značit P.
- levý vždy vyhraje, ať začne kdokoliv. Pozice je levého a L označuje třídu takových pozic.
- analogicky se třída pozic pravého hráče značí R.

[9] `http://en.wikipedia.org/wiki/Sprague-Grundy_theorem`

Řečeno tabulkou:

		R začne	
	vyhraje	L	R
L začne	L	L	V
	R	P	R

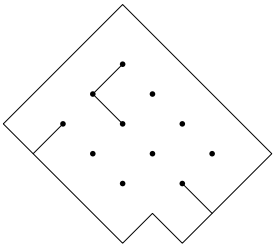
Zleva je příklad pozice levého (tj. hry náležející do L), pravého, pozice vyhrané a prohrané.



Všimněte si, že pozici rozebíráme, jako by v ní mohl začít hrát kterýkoliv hráč, což se bude hodit pro sčítání.

Nestranné hry jsou mnohem jednodušší: jelikož nelze rozlišit levého a pravého hráče, pozice jsou buď vyhrané, nebo prohrané pro začínajícího. Jak jste ověřili v první sérii, v Nimu jsou v třídě prohraných ty, v nichž je XOR hromádek 0, všechny ostatní jsou vyhrané.

Úkol 1 [3b]: Hra *Maze* spočívá v posouvání žetonu v bludišti(viz obrázek). Levý posouvá žeton šikmo doleva a dolů o kolik políček chce (avšak alespoň o jedno), pravý šikmo doprava a dolů také o libovolný počet políček. Není však možné táhnout skrz vnitřní nebo vnější obvodovou zeď.

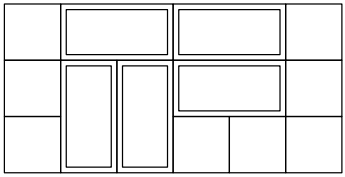


I v této hře prohrává, kdo nemůže táhnout. Na následujícím herním plánu určete pro každé z možných počátečních políček žetonu, jak dopadne hra.

Jinými slovy řečeno – zařaďte každé políčko do jedné ze tříd L, R, V, P.

Hra + hra = hra

Hurá na sčítání her! Mějme v dominování třeba tuto pozici:



Volná políčka jsou rozdělena dominovými kostkami uprostřed na dvě nezávislé části. Můžeme tedy vyřešit hru pro obě části a pak dát výsledky dohromady – čili obě části sečíst.

Právě ona nezávislost pozic je pro sčítání důležitá. Pokud lze zahrát do obou částí najednou, přesunout kámen z jedné části do druhé nebo něco podobného, nemusí platit nic, co si dále ukážeme.

Levá část pozice na obrázku je jasně vyhraná pro levého (má jeden tah, kdežto pravý tam nemůže položit ani jednu dominovou kostku). Pro pravou pozici si lze snadno rozmyslet, že oba hráči tam položí jedno domino, ať už začíná kdokoliv. Pak už nelze zahrát ani tah, takže pozice je v třídě P.

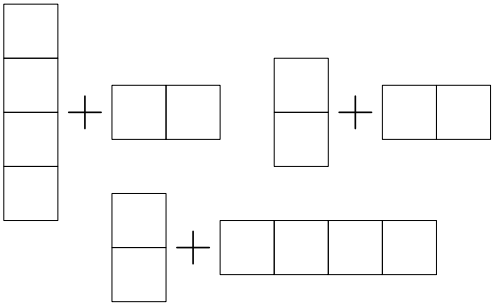
Jak dopadne součet? Levý (svislý) má dva tahy bez ohledu na to, kdo začne, pravý (vodorovný) jen jeden, pozici tedy vyhraje vždy levý.

Obecně platí, že součet hry G a pozice prohrané pro začínajícího je ve stejné výsledkové třídě jako G. Zkusme si to dokázat. Označme si prohranou pozici jako H a rozlišíme čtyři případy podle toho, kam patří G:

- G je v třídě L a na tahu je pravý: levý hraje vždy do stejné hry jako předtím pravý. Jinými slovy, když pravý zahraje do G, levý následně taky, když táhne do H, i levý táhne do H. Tím se hra rozpadne na dvě nezávislé části (tahy lze rozdělit ty, co jsou do G, a ty do H). Obě části má vyhrané levý, díky čemuž vyhrává i součet $G + H$.
- G je v L a na tahu je levý: levý zahraje do G, čímž ji změní na hru z P nebo z L (nic jiného není možné, jinak by v ní mohl vyhrát pravý). Pak už levý opět jen hraje do stejných her jako pravý, čímž vyhrává a součet náleží do L – ať začne kdokoliv, vyhraje levý.
- G je v R: analogický důkaz jako pro L.
- G je v P: druhý hráč na tahu se zase „opičí“ po prvním: hraje do stejné hry jako předtím první. Hra se tedy v podstatě rozpadne na dvě oddělené části, které jsou obě prohrané pro prvního hráče, a součtem je tedy prohraná hra.
- G je ve V: ten, kdo je na tahu, zahraje do G, čímž ji změní na prohranou hru nebo hru prvního hráče (pokud je to levý, tak hru z třídy L). Už víme, že součet dvou prohraných her je prohraná hra a součet hry z L (popř. R) a prohrané hry

náleží do L (popř. R), tudíž první na tahu v součtu $G + H$ vyhrává a součet je ve třídě V.

Dále platí, že součet dvou pozic vyhraných pro levého patří opět do třídy L, což si lze snadno rozmyslet. Analogicky dvě pozice pravého dávají v součtu hru z R. Jak však dopadne součet pozice levého a pozice pravého?



V součtu vlevo nahoře má levý o jeden tah více (tedy vyhraje bez ohledu na to, kdo začne), podobně v pozici dole má pravý o tah více. Součet vpravo je prohraná hra.

Úkol 2 [6b]: Pro každé přirozené k zjistěte, do jaké třídy patří dominování na mřížce $2 \times 4k$. Nikde zatím není položeno žádné domino (mřížka je prázdná).

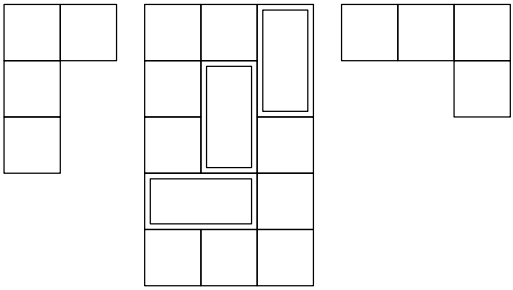
Jak je to se součtem více než dvou pozic? Aby tento součet nějak fungoval, bylo by třeba dokázat asociativitu, tedy že $(G + H) + I = G + (H + I)$, což je spíše technická záležitost. Komutativita je celkem zřejmá a už můžeme s hrami počítat téměř jako s čísly.

Jako s čísly? A co z her udělat rovnou čísla, ať se nám lépe sčítá? Pojďme na to! Jelikož však číslování her vydá na pěkných pár odstavců a dnes jich bylo už dost, na čísla si zatím jen připravíme půdu a necháme je na příště.

Bude se nám hodit umět rozeznat, které hry jsou shodné. Hry G a H se rovnají, tedy $G = H$, pokud dopadnou stejně, když ke každé přičteme libovolnou jinou hru. Přesněji řečeno pro každou hru X náleží hry $G + X$ a $H + X$ do stejné výsledkové třídy.

Speciálně jsou dvě hry stejné, pokud mají stejné herní stromy (až na pořadí synů).

Dále lze hru obrátit: hráči si vymění možné tahy, které od teď povedou do podobně obrácených pozic. V dominování si to lze představit tak, že levý bude pokládat vodorovná domina a pravý svislá nebo že hrací desku prostě otočíme o 90°. Obrácená hra G se značí $-G$.



Na obrázku je popořadě hra G, hra $H = G$ a hra $-G$. Všimněte si, že H vzniklo z G jednoduše přičtením prohrané pozice (volná políčka vpravo a dole).

Úkol 3 [5b]: Mějme dvě hry G a H, přičemž $G = H$. Dokažte, že hra $G+(-H)$ (intuitivně lze psát také $G - H$) náleží do třídy P. Nepůjde-li vám to, ukažte alespoň, že $G - G$ je prohraná hra.

Pavel Veselý

24-4-8 O hrách a číslech — 15 bodů

Vítejte u dalšího dílu herního seriálu. Podrobněji rozvineme Conwayovu teorii her, konkrétně se naučíme hry porovnávat a některým přiřazovat čísla.

Zopakujme si nejdůležitější pojmy z minula. Zajímají nás hry dvou hráčů označovaných jako *Levý* a *pRavý*. Vše jsme si dosud ukazovali na *dominování*, které spočívá v pokládání dominových kostek do čtvercové mřížky. Levý pokládá svisLe, pravý vodoRovně.

Dále jsme rozdělili hry (přesněji řečeno pozice) do čtyř tříd dle vítěze:

- vyhrané pozice: vyhraje hráč, který bude táhnout jako první; třída V
- prohrané hry: začínající hráč prohraje; třída P
- pozice levého: levý vždy vyhraje, ať začne kdokoliv; třída L
- pozice pravého: analogicky k levému; třída R

Důležité bylo sčítání pozic, které jsou nezávislé (nelze zahrát do obou najednou), přičemž začínající hráč si vždy může vybrat, kam potáhne.

V tomto díle se nám bude hodit rovnost her: hry G a H se rovnají, tedy $G = H$, pokud dopadnou stejně, když k oběma přičteme libovolnou jinou hru. Rovněž budeme využívat i obrácené hry značené $-G$, v níž si hráči vymění možné tahy, které od teď povedou do podobně obrácených pozic. V dominování odpovídá $-G$ otočení herního plánu o 90°.

Posledním úkolem v minulém díle bylo ukázat, že z $G = H$ vyplývá, že $G - H$ je prohraná hra. Tvrzení však platí i opačně: pokud $G - H$ je prohraná hra, pak $G = H$.

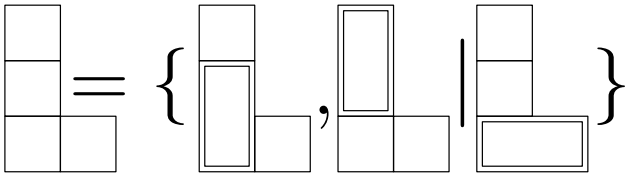
Když víme, které hry se rovnají, jak poznat, že nějaká hra je lepší pro levého než jiná? Opět budeme zkoumat hru $G - H$. Je-li to pozice levého (levý vyhraje, ať začíná kdokoliv), pak G je pro levého lepší než H, což se zapisuje jako $G > H$.

Pokud je pozice $G - H$ v třídě R, tak $G < H$. Dvojicím her, pro něž $G - H$ je pozice vyhraná pro začínajícího hráče, budeme říkat *neporovnatelné*, což se značí $G \parallel H$.

Tímto jsme si definovali částečné uspořádání na hrách. Stejně jako pro jiná uspořádání z $G < H$ a $H < I$ vyplývá $G < I$ (analogicky pro pravého).

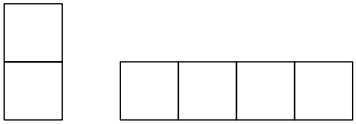
Číslování her

Než začneme číslovat hry, doplníme ještě abstraktní zápis her, v němž bude pozice G vypadat takto: $G = \{\mathcal{G}_L \mid \mathcal{G}_P\}$. $\mathcal{G}_L$ je množina pozic, kam může táhnout levý hráč, obdobně $\mathcal{G}_P$ jsou hry po tazích pravého hráče. Jelikož takto suchá definice může snadno zmást, podívejte se na obrázek:



Nyní se můžeme pustit do přiřazování čísel hrám. Ty budou vyjadřovat, jak moc velkou má levý nebo pravý výhodu v pozici oproti soupeři. Velikost čísla bude udávat, kolik tahů má jeden z hráčů navíc (kupodivu to vyjde někdy neceločíselně).

Kladná čísla budou znamenat výhodu levého a záporná výhodu pravého. Všimněte si, že pozici levého hráče nemůžeme přiřadit záporné číslo, jelikož v ní má alespoň malou výhodu levý. Obdobně pozice z třídy R nemohou dostat kladné číslo.

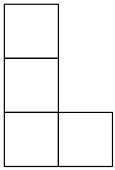


V pozici vlevo má levý hráč jeden tah navíc, hra dostane tedy číslo 1. V pozici vpravo má pravý dva tahy a levý nic, proto -2. V abstraktním zápise se hra s kladným číslem n dá zapsat jako $\{n - 1 \mid \}$ (levý hráč může táhnout do hry s číslem $n - 1$), hra $n < 0$ analogicky jako $\{ \mid n + 1\}$.

Když máme kladná i záporná čísla, je logické se ptát, co bude 0. V souladu s definicí kladných i záporných her znamená 0, že žádný z hráčů nemá výhodu, ani ten, co je na tahu, čili je to prohraná pozice.

Nejjednodušší taková hra je $\{ \mid \}$ (žádný hráč nemá tah) a každá prohraná hra se rovná 0. (Pro prohranou hru G není těžké ověřit, že $G = 0$. Nejsnadněji asi důkazem, že $G - 0$ je prohraná hra.)

Na začátku jsme tvrdili, že čísla her mohou být neceločíselná. Konkrétně mohou být jen tzv. *diadická racionální*, tedy zlomky $n/2^m$ v základním tvaru, kde n je celé číslo a m přirozené (včetně 0).



Hra s hodnotou $\frac{1}{2}$, levý získá tah navíc, když začne pravý:

Jak zjistit hodnotu dané pozice, když máme rozebrány hry, do nichž hráči mohou táhnout, nám řekne *pravidlo jednoduchosti*.

Mějme hru $G = \{\mathcal{G}_L \mid \mathcal{G}_R\}$, přičemž všechny tahy obou hráčů vedou do pozic, jež jsou čísla, a každý tah levého vede do pozice s číslem menším, než mají všechny pozice po tazích pravého hráče (formálně $\forall G_L \in \mathcal{G}_L$, $\forall G_R \in \mathcal{G}_R$: $G_L < G_R$). Potom G je nejjednodušší číslo x, nacházející se mezi hodnotami pozic levého a pravého ($G_L < x < G_R$).

Nejjednodušší v minulém odstavci znamená, že x je diadické, čili $n/2^m$ v základním tvaru, m je co nejmenší (preferují se celá čísla) a mezi čísly se stejným m se vybere to s menší absolutní hodnotou n.

Příklady:

- $\{-1 \mid 1\} = 0$
- $\{-100 \mid 10\} = 0$
- $\{-42 \mid -4\} = -5$
- $\{0 \mid 1\} = 1/2$
- $\{\{1 \mid -1\} \mid \{1 \mid -1\}\} = 0$ (je to prohraná hra)
- $\{1 \mid -1\}$ není číslo
- $\{-5 \mid -10\}$ není číslo (hra je však vyhraná pro pravého)
- $\{0 \mid 0\}$ není číslo

Poslední hra v seznamu, $\{0 \mid 0\}$, je nejjednodušší vyhranou hrou a značí se $*$.

Všimněte si, že některé hry levého či pravého jsou čísla a jiné ne. Žádná vyhraná hra však není číslo (výhodu má ten, kdo začne, ne vždy levý nebo pravý) a naopak každá prohraná hra se rovná 0, i když se to nemusí nahlédnout přes pravidlo jednoduchosti.

Her, které nejsou čísla, je hodně. Například $\uparrow = \{0 \mid *\}$, analogicky $\downarrow = \{* \mid 0\} = -\uparrow$. Hrám $\{a \mid b\}$, kde $a > b$, se říká přepínač, speciálně $\pm a = \{a \mid -a\}$ pro $a > 0$.

Co se týče uspořádání dle výhodnosti pro levého (či pravého), tak platí například (ověření necháme jako cvičení):

- $\{-10 \mid 10\} = \{-1 \mid 1\} = 0$,
- $\{1 \mid 2\} < \{1 \mid 6\}$,
- $\uparrow > 0$ a symetricky $\downarrow < 0$,
- $* \parallel 0$,

- $\pm 1 \parallel 0$,
- $\pm 10 \parallel \pm 5$.

Bylo by nyní potřeba ukázat, že hry, jež se rovnají, mají stejná čísla (mají-li vůbec nějaká). Nebo že součet her G a H má číslo rovné součtu čísel G a H.

Celkově by však důkazy zabraly možná celou sérii, takže případné zájemce odkáži na literaturu a internet (viz níže). Jejich hlavním důsledkem je, že lze libovolně zaměňovat hru a k ní příslušející číslo.

Místo důkazů zkusíme hry zjednodušovat. Podívejme se třeba na hru $G = \{3, 2, *, 0 \mid 4, 6, 8, 10\}$. Levý nemá žádný důvod táhnout do 2, $*$ nebo 0, podobně pravý potáhne určitě do 4, tedy $G = \{3 \mid 4\} = 3,5$.

Obecně lze v možnostech levého vyškrtat pozice horší pro levého než nějaká jiná pozice a analogicky mezi tahy pravého škrtáme pozice větší než nějaká jiná. Formálně zapsáno (pro možnosti levého): je-li $G = \{A, B, \ldots \mid C, D \ldots\}$, přičemž $A > B$ nebo $A = B$, pak $G = \{A, \ldots \mid C, D \ldots\}$ (nerovnosti mezi hrami jsou ty samé, co byly definovány na začátku tohoto dílu).

Úkol 1 [4b]: Mějme hromádku n sirek. Je-li n sudé, levý na tahu odebírá dvě sirky a pravý jednu. Je-li n liché, vezme levý jednu sirku a pravý dvě. Určete číslo hry pro každé $n \geq 0$.

Úkol 2 [5b]: Výše jsme si ukazovali pozici v dominování s hodnotou 1/2 (označme ji G). Ověřte, že $G + G = 1$. Dále nalezněte v dominování pozici H, jež není číslo, ale $H + H = 1$.

Úkol 3 [6b]: Hra *Padající domino* se hraje s bílými a černými dominovými kostkami postavenými v řadě za sebou. Tah hráče spočívá ve výběru jedné své kostky a jejím shození doleva nebo doprava, přičemž díky tomu spadnou všechny kostky ve směru, kam padala.

Levý hraje s bíLými kostkami, pravý s čeRnými a opět platí, že prohrál ten, kdo nemůže táhnout. Pro jednoduchost budeme posloupnost kostek zapisovat jako řetězec s písmeny B a C zastupujícími bílé a černé kostky. Například z pozice CBC má levý dva tahy, oba vedoucí do pozice C.

Pro hry BCBBBBC a BBCCBC najděte co nejjednodušší abstraktní zápis, jež neobsahuje konkrétní pozice, ale jen čísla, $*$, $\uparrow$ apod. (tedy třeba $\{\{4 \mid 2\} \mid -6\}$). Vyškrtáváte-li nějakou možnost, je třeba zdůvodnit, proč.

Tím jsme zakončili spíše neformální úvod do Conwayovy teorie her. Toto odvětví matematiky je však podstatně košatější, vynechali jsme dost důkazů (i zajímavých!), teploty a teploměry her (určování, jak moc výhodné je do hry zahrát) a mnoho dalších zajímavých věcí.

Co se týče praktické využitelnosti teorie, je na ní založený algoritmus pro řešení koncovek v Go nazvaný *Decomposition search*.

Prohloubit své znalosti teorie si můžete mimo jiné přečtením *Winning Ways for your Mathematical Plays* od Berlekampa, Conwaye a Guye a *On Numbers and Games* od Conwaye (ani jedna z nich bohužel nemá český překlad).

Mimochodem, hry v zápise $\{A, B, C, \ldots \mid P, Q, R, \ldots\}$ jsou tzv. *nadreálná čísla* (jejich speciálním případem jsou i reálná čísla), o nichž se dočtete více na Wikipedii.[10]

Hračičkům doporučujeme program CG Suite,[11] v němž lze zadávat pozice z různých her (nebo zapsané nadreálným číslem) a nechat určit jejich hodnotu, teplotu a další vlastnosti. (To může sloužit i pro kontrolu řešení úkolů, bude však třeba vše zdůvodnit.)

Příště se můžete těšit na návrat výpočetní části seriálu započaté v první a druhé sérii (algoritmy Minimax a Alfa-beta ořezávání). Nově nabyté znalosti si budete moct vyzkoušet na analýze jedné deskovky, kterou bude možné hrát online.

Pavel „Paulie“ Veselý

24-5-8 Jak hraje deskovky počítač? — 15 bodů

Herní seriál se blíží ke svému konci a je třeba mu nasadit korunku. Po dvou dílech o matematických hrách a jejich řešení přinášíme díl o hrách mnohem složitějších, které jen tak na papíře vyřešit neumíme. Můžete si představovat například šachy, dámu, piškvorky pět v řadě nebo jinou deskovku pro dva hráče.

V první sérii[12] byl probrán algoritmus Minimax, v druhé[13] jeho vylepšení pomocí Alfa-beta ořezávání. Pak uběhla celá zima, během níž možná leckomu algoritmy v paměti roztály jako jarní sníh. Zopakovat oba by však bylo na dlouho, takže se budeme muset spokojit s Minimaxem. K pochopení dalšího textu a úkolu by nám měl stačit.

Strom hry a Minimax

Situace je následující: máme hru bez náhody a chceme najít z její určité pozice co nejlepší tah. Když se však podíváme na jednotlivé tahy, neumíme jednoduše určit, který povede k výhře a který ne. Proto budeme muset prozkoumat i pozice, do nichž vedou naše tahy, což provedeme rekurzivně (tím samým algoritmem).

V podstatě procházíme tzv. *herním stromem* – jeho kořenem je pozice, pro niž hledáme nejlepší tah, synové kořene jsou pozice vzniklé po jednom našem tahu, jejich synové jsou pozice po tahu soupeře atd. Listy stromu jsou buď po-

[10] `http://cs.wikipedia.org/wiki/Nadre%C3%A1ln%C3%A9_%C4%8D%C3%ADslo`
[11] `http://www.cgsuite.org/`
[12] `http://ksp.mff.cuni.cz/viz/24-1-8`
[13] `http://ksp.mff.cuni.cz/viz/24-2-8`

zice, kde jsme vyhráli, nebo pozice, v nichž vyhrál soupeř (na remízu na chvíli zapomeňme).

Nechť jsme prošli rekurzivně celý strom. Jak zjistit, který tah vede do pozice pro nás vyhrané? (To je taková pozice, v které při *dokonalé* strategii obou hráčů vyhrajeme my.) Pomůže nám k tomu *ohodnocování* uzlů stromu, čili pozic.

Listy ohodnotíme tak, že pro nás vyhrané pozice budou ∞ a pro soupeře $-\infty$. Ostatním vrcholům přiřadíme hodnotu, až když máme ohodnocené jejich syny. Pokud jsme na tahu my, vezmeme maximum z ohodnocení synů (tedy ∞ odpovídající naší výhře, pokud tam je), soupeř na tahu zase bere minimum.

V praxi nejsme většinou schopni propočítat celý herní strom (s výjimkou jednoduchých her nebo pozic v koncovce), proto je dobré prohledávání ukončit v určité hloubce (odpovídající počtu odehraných tahů z kořene).

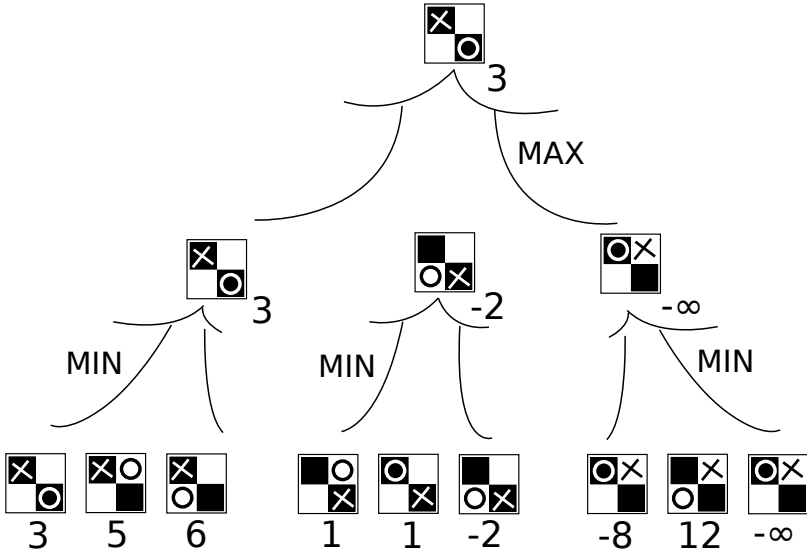
Prohledáváním jen do určité hloubky však získáme listy, které pro nás nejsou vyhrané či prohrané. Ty musíme ohodnotit heuristickou funkcí, která bude pro danou pozici vracet, jak moc pravděpodobné je, že v ní vyhrajeme. Když je lepší pro nás, vrátí kladné číslo, když pro soupeře, vrátí záporné. Vyrovnaná nebo remízová pozice obdrží 0.

Pokud jsme na tahu, vybíráme maximum ze synů (hráči, který vybírá maximum, budeme říkat *Max*), soupeř vybírá minimum (a nechť se jmenuje *Min*), algoritmus se tedy nazývá *Minimax*. Zde je jeho pseudokód:

```
// funkce vrací hodnotu pozice a nejlepší tah
def minimax(pozice, hloubka, natahu):
  // jsme v listu
  if hloubka == 0 or konecHry(pozice)
    return (hodnota(pozice), prazdnyTah)

  if natahu == Max:
    nejHodnota = -nekonecno - 1
    nejTah = prazdnyTah
    // projdeme tahy hráče Max
    for p in mozneTahy(pozice, Max):
      (hodnota, tah) =
        minimax(provedTah(p), hloubka - 1, Min)
      if hodnota > nejHodnota:
        nejHodnota = hodnota
        nejTah = p
    return (nejHodnota, nejTah)

  if natahu == Min:
    nejHodnota = nekonecno + 1
    nejTah = prazdnyTah:
```

```
// projdeme tahy hráče Min
for p in mozneTahy(pozice, Min):
  (hodnota, tah) =
    minimax(provedTah(p), hloubka - 1, Max)
  if hodnota < nejHodnota:
    nejHodnota = hodnota
    nejTah = p
return (nejHodnota, nejTah)
```

Pokud vám něco ohledně Minimaxu není jasné, nakoukněte do první série. Též přikládáme obrázek herního stromu prohledaného do hloubky 2:



Algoritmus lze zjednodušit tak, že pokaždé budeme vybírat maximum z hodnot synů, ale musíme pak mezi úrovněmi přenásobit hodnotu pozice číslem -1 a patřičně upravit hodnotící funkci. Zkuste si sami takto upravit pseudokód a ověřit, že dělá to samé. Zjednodušenému algoritmu se říká *Negamax* (násobení -1 je jakási negace a vždy vybíráme maximum).

Co se týče hodnotící funkce, měla by být velmi rychlá (rychlejší než prohledávání do hloubky o jedna větší s triviální ohodnocovací funkcí).

Minimax je sám o sobě dost neefektivní, protože zkouší všechny možné varianty, jak by hra dále mohla probíhat (i ty nesmyslné). Možným zrychlením výpočtu je proto negenerovat všechny tahy, což může být však mnohdy nebez-

pečné, protože lze přehlédnout dobrý tah... ale třeba u piškvorek přeskočení dobrého tahu zas tolik nehrozí, viz první sérii.

Algoritmus *Alfa-beta ořezávání* pak dostaneme z Minimaxu, když si všimneme, že některé uzly ve stromu mohou být pro jednoho z hráčů tak nevýhodné, že do nich určitě nebude hrát. Tyto části herního stromu tedy není potřeba prozkoumávat, mohou být tzv. *ořízntuty*. Z časoprostorových důvodů odkážeme na podrobnější popis Alfa-beta ořezávání[14] do druhé série.

Transpoziční tabulky

Často se také stane, že k jedné pozici se lze dostat několika různými posloupnostmi tahů, je tedy v herním stromě víckrát. Aby se vždy nemusela znovu a znovu prozkoumávat, uloží se poprvé výsledek výpočtu do tzv. *transpoziční tabulky*.

Když tedy máme prozkoumat nějakou pozici, nejprve nahlédneme do transpoziční tabulky, není-li tam. Pokud ano a byla už prohledána do stejné hloubky, jako chceme, vrátíme uložený výsledek, jinak provedeme výpočet a pozici uložíme.

Transpoziční tabulka technicky není nic jiného než hešovací tabulka (o nich se více můžete dočíst v kuchařce o hešování).[15] Z pozice vytvoříme obrovské číslo (třeba 64-bitové), které by mělo být pokud možno unikátní – nazývá se heš pozice.

Heš modulo velikost tabulky udává, kam máme pozici uložit. Jelikož velikost transpoziční tabulky bývá o dost menší než rozsah hodnot heše a také než počet dosažitelných pozic, často se stane, že políčko v tabulce, kam chceme pozici uložit, je už obsazené.

Tento problém se může řešit různými způsoby, ale vždy se nějaká pozice z tabulky za určitých podmínek vyhazuje (jinak by program spotřeboval moc paměti). Nově ukládaná pozice bývá vždy uložena.

Nejčastěji se do každého políčka tabulky dávají dvě pozice, aby nedocházelo tak často k vyhazování. Když už jsou před ukládáním na políčku dvě pozice, vyhodí se z tabulky ta, jež byla prohledána do menší hloubky, což se musí ukládat v tabulce.

[14] `http://ksp.mff.cuni.cz/viz/24-2-8`
[15] `http://ksp.mff.cuni.cz/viz/kucharky/hesovani`

Toto samozřejmě není jediný způsob, jak se chovat, když je buňka v tabulce obsazena, ale bývá lepší než ukládání jedné pozice do jednoho políčka tabulky, jak ukázaly testy.[16]

Abychom ověřili, že máme na konkrétním políčku uloženu hledanou pozici, musíme v tabulce uchovávat i heše pozic. Takže celkově pro každou pozici budeme ukládat její heš, vypočtenou hodnotu, nejlepší tah a hloubku, do níž byla prohledávána.

Může se také stát, že dvě různé pozice dostanou stejnou heš. Aby se to stávalo co nejméně, musí být funkce počítající heš dostatečně náhodná a rozsah hodnot heše velký. Když však problém nastane, často nelze zahrát z pozice tah uložený v transpoziční tabulce. Jinak se tento problém většinou neřeší, jeho výskyt bývá řídký.

Zbývá jen povědět, jak počítat onu heš. Často se používá *Zobristovo hešování*. Před výpočtem si pro každou kombinaci herního políčka a herního kamene (figurky) vygenerujeme náhodnou hodnotu (v rozsahu heše). Heš konkrétní pozice je XOR hodnot kombinací políčka a kamene, jež se momentálně nacházejí na herní desce.

Tedy např. v šachách se heš může počítat takto: náhodné číslo pro bílou věž na A1 XOR číslo pro bílého jezdce na B1 XOR atd.

Význam transpoziční tabulky vzroste při použití *iterativního prohlubování*. Při něm prostě prohledávání pouštíme do hloubky 1, pak 2, 3, ... , dokud nedojde čas nebo nezjistíme, že pozice je pro nás vyhraná či prohraná. Navíc při prohledávání upřednostňujeme nejlepší tahy z minulého prohledávání do menší hloubky (ty najdeme právě v transpoziční tabulce).

Dalších vylepšení Alfa-beta algoritmu je lidově řečeno hafo. Ostatní však ponecháme na dobrovolné samostudium, které se může hodit při řešení úkolu. Dobrým zdrojem může být Chess Programming Wiki.[17]

Úkol [14b]: Úkol spočívá ve zkoumání a analýze hry Dvonn, neboli jak by měl v takové hře počítač hledat z daného stavu nejlepší tah. Abyste se měli čeho chytit, dostanete návodné otázky. Odladěný program po vás chtít nebudeme, mohlo by vám to sebrat klidně celé jaro. :-)

[16] `http://mediocrechess.blogspot.com/2007/01/guide-transposition-tables.html`

[17] `http://chessprogramming.wikispaces.com/`

Aby se vám hra dobře analyzovala, je možné ji hrát třeba na BoardSpace.net[18] (s lidmi i roboty). Pravidla najdete na internetu i v češtině[19] a soupeře si můžete domlouvat na našem fóru[20] (třeba autor seriálu si s vámi rád zahraje). Bohatě stačí, když se zamyslíte nad fází hry po rozmístění kamenů (tj. když už se kameny přemísťují).

Algoritmus na hledání nejlepšího tahu už znáte, pár triků také. Představte si, že chcete robota pro Dvonn implementovat ve svém oblíbeném jazyce, který by ovšem sám o sobě měl být rychlý (což třeba Python není, C# také moc ne). Jak efektivně reprezentovat pozici? Jak s pomocí té reprezentace rychle generovat a provádět tahy?

Zamyslete se rovněž nad ohodnocováním pozice. (Výhra bílého je nějaká velká konstanta H, výhra černého $-H$, remízová nebo vyrovnaná pozice má 0, vše ostatní je na vás.) I toto by mělo být pekelně rychlé. Namísto slovního popisu můžete dodat rozumně čitelný (pseudo)kód, což lze udělat i u jiných částí úkolu.

Dalším námětem může být řazení tahů dle výhodnosti pro hráče na tahu, které se hodí pro Alfa-beta ořezávání (lepší tahy spíše způsobí ořezání pozice). Jak lze v této hře řadit tahy? Dají se generovat rovnou v nějakém „dobrém“ pořadí?

Úkol je v podstatě dost kreativní a klidně napište i o něčem jiném, co vás při zkoumání hry a přemýšlení o algoritmech napadne, bude to náležitě oceněno.

Udělá nám radost (a vám bodově přilepší) samostudium algoritmů a jiných technik z této oblasti (např. těch, co vylepšují Alfa-beta ořezávání). Z toho pak sepište vlastní poznámky o té technice, případně i o jejím nasazení na Dvonn. Stačí i pár odstavců.

Asi vás zajímá bodování. Plným počtem ohodnotíme řešení obsahující:

- vhodnou reprezentaci pozice a krátký popis, jak implementovat generování tahů,
- způsob ohodnocení pozice, neboli jak a proč se různé vlastnosti stavu hry započítávají do hodnoty, rovněž s krátkým nastíněním efektivní implementace,
- alespoň krátké zamyšlení nad řazením tahů v Dvonnu,
- jak zhruba vypadá herní strom, tedy jak dlouhá je běžná hra (měřeno tahy) a kolik má hráč průměrně tahů v různých částech hry.

Jednotlivé části hodnocení lze nahradit i jiným souvisejícím nápadem, tématem apod. Velmi dobrá řešení (po kvalitativní i kvantitativní stránce) možná obdrží nějaký ten bonusový bod.

[18] `http://www.boardspace.net/`

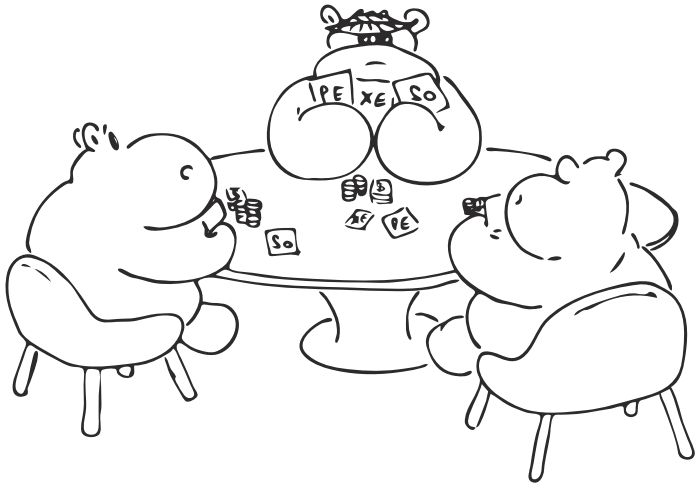
[19] `http://deskovehry.blogspot.com/2009/10/pravidla-dvonn.html`

[20] `http://ksp.mff.cuni.cz/forum/`

Alfa-beta není zdaleka jediným používaným algoritmem v oblasti her, i pokud pomineme algoritmy vhodné jen pro konkrétní hry. V koncovkách se často hodí nasadit *Proof Number Search*,[21] bylo jím nedávno také zjištěno, že počáteční pozice v anglické dámě je remízová. Dalším zajímavým algoritmem je *Monte-Carlo Tree Search*,[22] používající pseudonáhodné simulace hry.

Oba tyto algoritmy sice nejsou jednoduché, ale jsou obecně použitelné pro velké množství her. Existují také algoritmy určené jen pro jednu hru založené na jejích specifických vlastnostech.

Tak a je po seriálu o hrách matematických i výpočetně složitějších. Věříme, že vás zaujal a třeba se vám budou nabyté znalosti ještě někdy hodit.



[21] `http://fragrieu.free.fr/SearchingForSolutions.pdf`
[22] `http://senseis.xmp.net/?MonteCarlo`

Programátorské kuchařky

Kuchařka první série – složitost

Časová a paměťová složitost

V této kuchařce se můžete dočíst o základech časové a paměťové složitosti. Po přečtení byste měli být schopni sami rozebrat složitost jednoduchých algoritmů. To se hodí třeba při návrhu algoritmů a řešení algoritmických úloh, které můžete potkat například v KSP.

Nejdříve si ujasníme, co to ta složitost vlastně je, a ukážeme si pár příkladů. Pak si řekneme, s jakou přesností budeme složitost chtít určovat, a zavedeme si asymptotickou složitost. Na závěr si ukážeme běžné třídy složitosti.

Základní přehled

Pokud řešíme nějakou programátorskou úlohu, často nás napadne více různých řešení a potřebujeme se rozhodnout, které z nich je „nejlepší“. Abychom to mohli posoudit, potřebujeme si zavést měřítka, podle kterých budeme různé algoritmy porovnávat. Nás u každého algoritmu budou zajímat dvě vlastnosti: čas, po který algoritmus běží, a paměť, kterou při tom spotřebuje.

Čas nebudeme měřit v sekundách (protože stejný program na různých počítačích běží rozdílnou dobu), ale v počtu provedených operací. Pro jednoduchost budeme předpokládat, že aritmetické operace, přiřazování, porovnávání, apod. nás stojí jednotkový čas. Ona to není úplná pravda, tyto operace se ve skutečnosti přeloží na procesorové instrukce, které se teprve zpracovávají. Ale nám postačí vědět, že těch instrukcí bude vždy konstantní počet. A později se dozvíme, proč nám na takové konstantě nezáleží.

Množství použité paměti můžeme zjistit tak, že prostě spočítáme, kolik bytů paměti náš program použil. Nám obvykle bude stačit menší přesnost, takže všechna čísla budeme považovat za stejně velká a velikost jednoho prohlásíme za jednotku prostoru.

Jak čas, tak paměť se obvykle liší podle toho, jaký vstup náš program zrovna dostal – na velké vstupy spotřebuje více času i paměti než na ty malé. Budeme proto oba parametry určovat v závislosti na velikosti vstupu a hledat funkci, která nám tuto závislost popíše. Takové funkci se odborně říká *časová (případně paměťová, někdy též prostorová) složitost* algoritmu/programu.

Nyní si na příkladu ukážeme, jak se časová a paměťová složitost dá určovat intuitivně, a pak si vše podrobně vysvětlíme.

Představme si, že máme danou posloupnost N celých čísel, ze které chceme vybrat maximum. Použijeme algoritmus, který za maximum prohlásí nejprve první číslo posloupnosti. Pak toto maximum postupně porovnává s dalšími čísly

posloupnosti a pokud je některé větší, učiní z něj nové maximum. Zapsat bychom to mohli třeba takto:

```
posl[1...N] = vstup
max = posl[1]
Pro i = 2 až N:
   Jestliže posl[i] > max:
      max = posl[i]
Vypiš max
```

Není těžké nahlédnout, že algoritmus provede maximálně $N - 1$ porovnání. Intuitivně časová složitost bude lineárně záviset na N, protože porovnání dvou čísel nám zabere „jednotkový čas“ a paměťová složitost bude také na N záviset lineárně, protože si každé číslo z posloupnosti budeme uchovávat v paměti. Pokud bychom si nepamatovali celou posloupnost, ale vždy jen poslední přečtený člen, stačilo by nám jen konstantně mnoho proměnných, takže paměťová složitost by klesla na konstantní (nezávislou na N) a časová by zůstala stejná.

Jiný příklad: Mějme dané číslo K. Naším úkolem je vypsat tabulku všech násobků čísel od 1 do K:

```
Pro i = 1 až K:
   Pro j = 1 až K:
      Vypiš i*j a mezeru
   Přejdi na nový řádek
```

Tabulka má velikost K^2 a na každém jejím políčku strávíme jen konstantní čas. Proto časová složitost bude záviset na čísle K kvadraticky, tedy bude K^2. Paměťová složitost bude buď konstantní, pokud hodnoty budeme jen vypisovat, anebo kvadratická, pokud si tabulku budeme ukládat do paměti. Můžeme si také všimnout, že tabulku nemusíme vypisovat celou, ale bude nám stačit jen její dolní trojúhelníková část – i tak budeme muset spočítat $(K \cdot K - K)/2 + K = K^2/2 + K/2$ hodnot, což je stále řádově kvadratické vzhledem ke K.

U výběru algoritmu tedy bereme v potaz čas a paměť. Který z těchto faktorů je pro nás důležitější, se musíme rozhodnout vždy u konkrétního příkladu. Často také platí, že čím více času se snažíme ušetřit, tím více paměti nás to pak stojí. To kvůli chytré reprezentaci dat v paměti a různým vyhledávacím strukturám, o kterých se můžete dočíst v našich dalších kuchařkách. Nás u valné většiny algoritmů bude nejdříve zajímat časová složitost a až poté složitost paměťová. Paměti mají totiž dnešní počítače dost, a tak se málokdy stane, že vymyslíme algoritmus, který má dokonalý čas, ale nestačí nám na něj paměť. Ale přesto doporučujeme dávat si na paměťová omezení pozor.

Než se pustíme do podrobnějšího vysvětlování, ještě si ukážeme tzv. „metodu kouknu a vidím“, kterou můžeme použít na určování časové složitosti u těch nejjednodušších algoritmů. Spočívá jen v tom, že se podíváme, kolik nejvíc ob-

sahuje náš program vnořených cyklů. Řekněme, že jich je k a že každý běží od 1 do N. Potom za časovou složitost prohlásíme N^k.

Vzhledem k čemu budeme složitosti určovat?

Složitosti obvykle určujeme vzhledem k velikosti vstupu (počet čísel, případně znaků na vstupu). Tento počet si označme N. Časovou i paměťovou složitost pak vyjádříme vzhledem k tomuto N. To je vidět třeba na výběru maxima v předchozím textu.

Pokud by existovalo několik vstupů stejné velikosti, pro které náš algoritmus běží různě dlouho, bude časová složitost popisovat ten nejhorší z nich (takový, na kterém algoritmus poběží nejpomaleji). Stejně tak pro paměťovou složitost použijeme ten ze vstupů délky N, na který spotřebujeme nejvíce paměti. Dostaneme tzv. složitosti v nejhorším případě. Podrobněji si o tom povíme později.

Někdy se nám hodí určit složitost v závislosti na více než jedné proměnné. Pokud bychom například chtěli vypisovat všechny dvojice podstatného a přídavného jména ze zadaného slovníku, strávíme tím čas, který bude záviset nejen na celkové velikosti slovníku, ale i na tom, kolik obsahuje podstatných a kolik přídavných jmen. Rozmyslete si, jaká složitost vyjde, pokud víte, že velikost slovníku je S, podstatných jmen je A a přidavných jmen B.

Častým příkladem, kde si velikost vstupu potřebujeme rozdělit do více proměnných, jsou algoritmy pracující s grafy (viz grafová kuchařka).[23] V případě grafů obvykle vyjadřujeme složitost pomocí proměnných N a M, kde N je počet vrcholů grafu a M je počet jeho hran. I pro více proměnných vybíráme nejhorší případ.

Ne vždy ale určujeme složitosti v závislosti na velikosti vstupů. Například pokud je velikost vstupu konstantní, složitost určíme vzhledem k hodnotám proměnných na vstupu. Třeba u příkladu s tabulkou násobků jsme složitost určili vzhledem k velikosti tabulky, kterou jsme dostali na vstupu. Jiným příkladem může být vypsání všech prvočísel menších než dané N.

Asymptotická složitost

V této části textu se budeme věnovat pouze časové složitosti. Všechna pravidla, která si řekneme, pak budou platit i pro paměťovou složitost.

U určování časové složitosti nás bude především zajímat, jak se algoritmy chovají pro velké vstupy. Mějme například algoritmus A o časové složitosti $4N$ a algoritmus B o složitosti N^2. Tehdy je sice pro $N = 1, 2, 3$ algoritmus B rychlejší než A, ale pro všechna větší N ho už algoritmus A předběhne. Takže pokud bychom si měli mezi těmito algoritmy zvolit, vybereme si algoritmus A.

[23] `http://ksp.mff.cuni.cz/tasks/20/cook3.html`

U složitosti nás obvykle nebude zajímat, jak se chová na malých vstupech, protože na těch je rychlý téměř každý algoritmus. Rozhodující pro nás bude složitost na maximálních vstupech (pokud nějaké omezení existuje) anebo složitost pro „hodně velké vstupy“. Proto si zavedeme tzv. **asymptotickou časovou složitost.**

Představme si, že máme algoritmus se složitostí $n^2/4+6n+12$. Pod asymptotikou si můžeme představit, že nás zajímá jen nejvýznamnější člen výrazu, podle kterého se pak pro velké vstupy chová celý výraz. To znamená, že:

- Konstanty u jednotlivých členů můžeme škrtnout (např. $6n$ se chová podobně jako n). Tím dostáváme $n^2 + n + 1$.
- Pro velká n je $n+1$ oproti n^2 nevýznamné, tak ho můžeme také škrtnout. Dostáváme tak složitost n^2. Obecně škrtáme všechny členy, které jsou pro dost velké n menší než nějaký neškrtnutý člen.

Tahle pravidla sice většinou fungují, ale škrtat ve výpočtech přece nemůžeme jen tak. Proto si nyní zavedeme operátor $\mathcal{O}$ (velké O), díky kterému budeme umět popsat, co přesně naše „škrtání“ znamená, a používat ho korektně.

Definice: Mějme funkce $f : \mathbb{N} \to \mathbb{R}^+$ a $g : \mathbb{N} \to \mathbb{R}^+$. Řekneme, že $f \in \mathcal{O}(g)$, pokud $\exists n_0 \in \mathbb{N}$ a $\exists c \in \mathbb{R}^+$ tak, že $\forall n \geq n_0$ platí $f(n) \leq c \cdot g(n)$.

Nyní slovy: Mějme funkce f a g funkce z přirozených do kladných reálných čísel. Řekneme, že funkce f patří do třídy $\mathcal{O}(g)$, pokud existují konstanty n_0 a c takové, že f je pro dost velká n (totiž pro $n \geq n_0$) menší než $c \cdot g(n)$.

Někdy také píšeme, že $f = \mathcal{O}(g)$ nebo říkáme, že program má složitost $\mathcal{O}(f)$.

A zde je použití: $n^2/4+6n+12 \in \mathcal{O}(n^2)$, protože například pro $c = 10$ platí pro všechna $n > 1$ (tedy $n_0 = 2$):

$$n^2/4 + 6n + 12 \leq 10n^2.$$

Pokud vám tento způsob nevyhovuje a více se vám líbí metoda pomocí „škrtání“, tak ji klidně používejte, akorát všude pište $\mathcal{O}(\ldots)$. Někdy také říkáme, že se konstanty a méně významné členy v $\mathcal{O}$ ztrácí.

Ještě poznamenejme, že operátor $\mathcal{O}(\ldots)$ znamená asymptotický horní odhad funkce. Takže pokud funkce patří do $\mathcal{O}(N)$, tak patří i do $\mathcal{O}(N^2)$, $\mathcal{O}(N^3)$, ...

Nejhorší a průměrný případ

Opět si vše vysvětlíme jen na časové složitosti.

Velká část algoritmů běží pro různé vstupy stejné velikosti různou dobu. U takových algoritmů pak můžeme rozlišovat složitost v nejhorším případě (tu už známe), v nejlepším případě a třeba i průměrnou časovou složitost.

Vše si ukážeme na algoritmu BubbleSort (bublinkovém třídění), o kterém se můžete dočíst v kuchařce o třídících algoritmech.[24] Funguje tak, že se dívá na všechny dvojice sousedních prvků a kdykoliv je dvojice ve špatném pořadí, tak ji prohodí. Zde je pseudokód algoritmu:

```
BubbleSort(pole, N):
   Opakuj:
      setříděno = 1
      Pro i = 1 až N-1:
         Jestliže pole[i] > pole[i+1]:
            p = pole[i]
            pole[i] = pole[i+1]
            pole[i+1] = p
            setříděno = 0
   Skonči, až bude setříděno = 1
```

Časová složitost v nejhorším případě činí $\mathcal{O}(N^2)$ – v každém průchodu vnějším cyklem nám totiž největší hodnota „probublá“ na konec a ostatní se posunou o jednu pozici doleva. Rozmyslete si, proč. Průchodů je proto nejvýše $N - 1$ a každý z nich trvá $\mathcal{O}(N)$. Tento nejhorší případ může doopravdy nastat, pokud necháme setřídit klesající posloupnost. Tam provedeme přesně $N - 1$ průchodů.

Naopak v nejlepším případě bude časová složitost pouze $\mathcal{O}(N)$. To nastane, pokud na vstupu dostaneme už setříděnou posloupnost. U té algoritmus pouze zkontroluje všechny dvojice a pak se ihned zastaví.

Průměrná časová složitost nám udává, jak dlouho náš algoritmus běží průměrně. Co to ale znamená, není snadné definovat ani spočítat. U třídícího algoritmu bychom mohli počítat průměr přes všechny možnosti, jak mohou být prvky na vstupu zamíchané (tedy přes všechny jejich permutace). To nám někdy může dát přesnější odhad chování algoritmu.

Zrovna u BubbleSortu a mnoha jiných algoritmů vyjde průměrná časová složitost stejně jako složitost v nejhorším případě. Jedním z nejznámějších příkladů algoritmu, který je v průměru asymptoticky lepší, je třídící algoritmus QuickSort (opět viz třídicí kuchařka). Jeho průměrná časová složitost činí $\mathcal{O}(N \cdot \log N)$, zatímco v nejhorším případě může běžet až kvadraticky dlouho.

Často používané složitosti

Na závěr si ukážeme často se vyskytující časové složitosti algoritmů (ty paměťové jsou obdobné). Seřadili jsme je od nejrychlejších a ke každé připsali příklad algoritmu.

[24] `http://ksp.mff.cuni.cz/tasks/20/cook2.html`

$\mathcal{O}(1)$ – *konstantní* (třeba zjištění, jestli je číslo sudé)
$\mathcal{O}(\log N)$ – *logaritmická* (binární vyhledávání); všimněte si, že na základu logaritmu nezáleží, protože platí $\log_a n = \log_b n / \log_b a$, takže logaritmy o různých základech se liší jen konstanta-krát, což se „schová do $\mathcal{O}$-čka“.
$\mathcal{O}(N)$ – *lineární* (hledání maxima z N čísel)
$\mathcal{O}(N \cdot \log N)$ – *lineárně-logaritmická* (nejlepší algoritmy na třídění pomocí porovnávání)
$\mathcal{O}(N^2)$ – *kvadratická* (BubbleSort)
$\mathcal{O}(N^3)$ – *kubická* (násobení matic podle definice)
$\mathcal{O}(2^N)$ – *exponenciální* (nalezení všech posloupností délky N složených z nul a jedniček; pokud je chceme i vypsat, dostaneme $\mathcal{O}(N \cdot 2^N)$)
$\mathcal{O}(N!)$ – *faktoriálová*, $N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot N$ (nalezení všech permutací N prvků, tedy třeba všech přesmyček slova o N různých písmenech)

Složitosti ještě často rozdělujeme na *polynomiální* a *nepolynomiální.* Polynomiální říkáme těm, které patří do $\mathcal{O}(N^k)$ pro nějaké k. Naopak nepolynomiální jsou ty, pro něž žádné takové k neexistuje.

Do polynomiálních algoritmů patří i algoritmus se složitostí $\mathcal{O}(\log N)$. A to proto, že $\mathcal{O}(\log N) \subset \mathcal{O}(N)$ (každý algoritmus, který seběhne v čase $\mathcal{O}(\log N)$, seběhne i v $\mathcal{O}(N)$).

Nepolynomiální jsou z naší tabulky třídy $\mathcal{O}(2^N)$ a $\mathcal{O}(N!)$. Takové algoritmy jsou extrémně pomalé a snažíme se jim co nejvíce vyhýbat.

Pro představu o tom, jak se složitost projevuje na opravdovém počítači, se podíváme, jak dlouho poběží algoritmy na počítači, který provede 10^9 (miliardu) operací za sekundu. Tento počítač je srovnatelný s těmi, které dnes běžně používáme. Podívejme se, jak dlouho na něm poběží algoritmy s následujícími složitostmi:

funkce / $n =$	10	20	50	100	1 000	10^6
$\log_2 n$	3.3 ns	4.3 ns	4.9 ns	6.6 ns	10.0 ns	19.9 ns
n	10 ns	20 ns	30 ns	100 ns	1 µs	1 ms
$n \cdot \log_2 n$	33 ns	86 ns	282 ns	664 ns	10 µs	20 ms
n^2	100 ns	400 ns	900 ns	100 µs	1 ms	1 000 s
n^3	1 µs	8 µs	27 µs	1 ms	1 s	10^9 s
2^n	1 µs	1 ms	1 s	10^{21} s	10^{292} s	$\approx \infty$
$n!$	3 ms	10^9 s	10^{23} s	10^{149} s	10^{2558} s	$\approx \infty$

Pro představu: 1 000 s je asi tak čtvrt hodiny, 1 000 000 s je necelých 12 dní, 10^9 s je 31 let a 10^{18} s je asi tak stáří Vesmíru. Takže nepolynomiální algoritmy začnou být velmi brzy nepoužitelné.

Dnešní menu servírovali
Karel Tesař a Martin Mareš

Kuchařka druhé série – dynamické programování

Rekurzivní funkce a dynamické programování

Rekurzivní funkce je taková funkce, která při svém běhu volá sama sebe, často i více než jednou, což v důsledku může vést na exponenciální algoritmus.

Dynamické programování je technika, kterou jde z pomalého rekurzivního algoritmu vyrobit pěkný polynomiální (až na výjimečné případy). Ale nepředbíhejme, nejdříve se podíváme na jednoduchý příklad rekurze:

Fibonacciho čísla

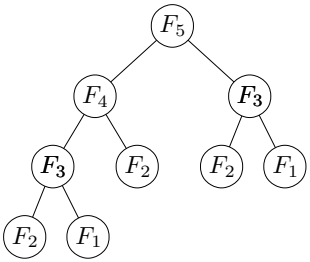
Budeme počítat n-té číslo Fibonacciho posloupnosti. To je posloupnost, jejímiž prvními dvěma členy jsou jedničky ($F_1 = 1$, $F_2 = 1$) a každý další člen je součtem dvou předchozích ($F_n = F_{n-1} + F_{n-2}$ pro $n > 2$). Začíná takto:

$$1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad 89 \quad \ldots$$

Pro nalezení n-tého členu (ten budeme značit F_n) si napíšeme rekurzivní funkci `Fibonacci(n)`, která bude postupovat přesně podle definice – zeptá se sama sebe rekurzivně, jaká jsou dvě předchozí čísla, a pak je sečte. Možná více řekne program:

```
function Fibonacci(n: integer): integer;
begin
  if n <= 2 then
    Fibonacci := 1
  else
    Fibonacci := Fibonacci(n-1) + Fibonacci(n-2)
end;
```

To, jak funkce volá sama sebe, si můžeme snadno nakreslit třeba pro výpočet čísla F_5:



Vidíme, že se program rozvětvuje, což tvoří strom volání. V každém vrcholu tohoto stromu trávíme konstantní čas, takže časová složitost celého algoritmu je až na konstantu rovna počtu vrcholů tohoto stromu. Kolik to je, spočítáme jednoduchou úvahou.

Každý vrchol stromu vrací hodnotu, která je součtem hodnot v jeho synech. Proto je hodnota v kořeni rovna součtu hodnot v listech. V listech jsou ovšem jedničky (F_1 a F_2), takže listů musí být právě F_n a všech vrcholů dohromady aspoň F_n.

Proto na spočítání n-tého Fibonacciho čísla spotřebujeme čas alespoň takový, kolik je ono číslo samo. Ale jak velké takové F_n vlastně je? Můžeme třeba využít toho, že

$$F_n = F_{n-1} + F_{n-2} \geq 2 \cdot F_{n-2},$$

z čehož indukcí dokážeme

$$F_n \geq 2^{n/2} \quad \text{pro } n \geq 6.$$

Funkce `Fibonacci` má tedy alespoň exponenciální časovou složitost což není nic vítaného.

Jak najít efektivnější algoritmus? Všimneme si, že některé podstromy jsou shodné. Zřejmě to budou ty části, které reprezentují výpočet stejného Fibonacciho čísla – v našem příkladě třeba třetího. Tyto výpočty opakujeme stále dokola.

Nenabízí se proto nic snazšího, než si jejich výsledky uložit a pak je kdykoliv vytáhnout jako pověstného králíka z klobouku s minimem námahy.

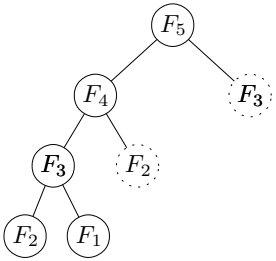
Právě zde je zmínka o králících příhodná. Legenda o Fibonacciho číslech vypráví, že k jejich objevu došlo při výzkumu rozmnožování králíků.

Leonardo Pisánský (známý též jako Fibonacci) totiž pěstoval králíky. První dva měsíce měl 1 pár, další měsíc měl 2 páry, pak 3, pak 5, ...

Bude nám k tomu stačit jednoduché pole P o n prvcích, na počátku inicializované nulami. Kdykoliv budeme chtít spočítat některý člen, nejdříve se podíváme do pole, zda jsme ho již jednou nespočetli. A naopak jakmile hodnotu spočítáme, hned si ji do pole poznamenáme:

```
var P: array[1..MaxN] of integer;
function Fibonacci(n: integer): integer;
begin
  if P[n] = 0 then
  begin
    if n <= 2 then
      P[n] := 1
    else
      P[n] := Fibonacci(n-1) + Fibonacci(n-2)
  end;
  Fibonacci := P[n]
end;
```

Podívejme se, jak vypadá strom volání nyní:



Na každý člen posloupnosti se tentokrát ptáme maximálně dvakrát – k výpočtu ho potřebují dva následující členy. To ale znamená, že funkci `Fibonacci` zavoláme maximálně $2n$-krát, čili jsme touto jednoduchou úpravou zlepšili exponenciální složitost na lineární.

Zdálo by se, že abychom získali čas, museli jsme obětovat paměť, ale to není tak úplně pravda. V prvním příkladu sice nepoužíváme žádné pole, ale při volání funkce si musíme zapamatovat některé údaje, jako je třeba návratová adresa, parametry funkce a její lokální proměnné, a na to samotné potřebujeme určitě paměť lineární s hloubkou vnoření, v našem případě tedy lineární s n.

Určitě vás už také napadlo, že n-té Fibonacciho číslo se dá snadno spočítat i bez rekurze. Stačí prvky našeho pole P plnit od začátku – kdykoli známe $P[1 \ldots k] = F_{1 \ldots k}$ (všechny prvky pole na pozicích od 1 do k), dokážeme snadno spočítat i $P[k+1] = F_{k+1}$:

```
function Fibonacci(n: integer): integer;
var
  P: array[1..MaxN] of integer;
  i: integer;
begin
  P[1] := 1;
  P[2] := 1;
  for i := 3 to n do
    P[i] := P[i-1] + P[i-2];
  Fibonacci := P[n]
end;
```

Zopakujme si, co jsme postupně udělali – nejprve jsme vymysleli pomalou rekurzivní funkci, kterou jsme zrychlili zapamatováváním si mezivýsledků.

Nakonec jsme ale celou rekurzi „obrátili naruby“ a mezivýsledky počítali od nejmenšího k největšímu, aniž bychom se starali o to, jak se na ně původní rekurze ptala.

V případě Fibonacciho čísel je samozřejmě snadné přijít rovnou na nerekurzivní řešení a dokonce si všimnout, že si stačí pamatovat jen poslední dvě hodnoty, a paměťovou složitost tak zredukovat na konstantní.

Zmíněný obecný postup zrychlování rekurze nebo řešení úlohy od nejmenších podproblémů k těm největším funguje i pro řadu složitějších úloh. Obvykle se mu říká *dynamické programování*.

Problém batohu

Je dáno N předmětů o hmotnostech $m_1, \ldots, m_N$ (celočíselných) a také číslo M (nosnost batohu). Úkolem je vybrat některé z předmětů tak, aby součet jejich hmotností byl co největší, ale přitom nepřekročil M. Předvedeme si algoritmus, který tento problém řeší v čase $\mathcal{O}(MN)$.

Náš algoritmus bude používat pomocné pole $A[0 \ldots M]$ a jeho činnost bude rozdělena do N kroků. Na konci k-tého kroku bude prvek $A[i]$ nenulový právě tehdy, jestliže z prvních k předmětů lze vybrat předměty, jejichž součet hmotností je přesně i.

Před prvním krokem (po nultém kroku) jsou všechny hodnoty $A[i]$ pro $i > 0$ nulové a $A[0]$ má nějakou nenulovou hodnotu, řekněme -1.

Všimněme si, jak kroky algoritmu odpovídají podúlohám, které řešíme – v prvním kroku vyřešíme podúlohu tvořenou jen prvním předmětem, ve druhém kroku prvními dvěma předměty, pak prvními třemi předměty atd.

Popišme si nyní k-tý krok algoritmu. Pole A budeme procházet od konce, tj. od $i = M$. Pokud je hodnota $A[i]$ stále nulová, ale hodnota $A[i - m_k]$ je nenulová, změníme hodnotu uloženou v $A[i]$ na k (později si vysvětlíme, proč zrovna na k).

Nyní si rozmyslíme, že po provedení k-tého kroku odpovídají nenulové hodnoty v poli A hmotnostem podmnožin z prvních k předmětů (*podmnožina* je v podstatě jen výběr nějaké části předmětů).

Pokud je hodnota $A[i]$ nenulová, pak buď byla nenulová před k-tým krokem (a v tom případě odpovídá hmotnosti nějaké podmnožiny prvních $k-1$ předmětů) anebo se stala nenulovou v k-tém kroku.

Potom ale hodnota $A[i - m_k]$ byla před k-tým krokem nenulová, a tedy existuje podmnožina prvních $k-1$ předmětů, jejíž hmotnost je $i - m_k$. Přidáním k-tého předmětu k této podmnožině vytvoříme podmnožinu předmětů hmotnosti přesně i.

Naopak, pokud lze vytvořit podmnožinu X hmotnosti i z prvních k předmětů, pak takovou podmnožinu X lze buď vytvořit jen z prvních $k-1$ předmětů, a tedy hodnota $A[i]$ je nenulová již před k-tým krokem, anebo k-tý předmět je obsažen v takové množině X.

Potom ale hodnota $A[i - m_k]$ je nenulová před k-tým krokem (hmotnost podmnožiny X bez k-tého prvku je $i - m_k$) a hodnota $A[i]$ se stane nenulovou v k-tém kroku.

Po provedení všech N kroků odpovídají nenulové hodnoty $A[i]$ přesně hmotnostem podmnožin ze všech předmětů, co máme k dispozici. Speciálně největší index i_0 takový, že hodnota $A[i_0]$ je nenulová, odpovídá hmotnosti nejtěžší podmnožiny předmětů, která nepřekročí hmotnost M.

Nalézt jednu množinu této hmotnosti také není obtížné: V k-tém kroku jsme měnili nulové hodnoty v poli A na hodnotu k, takže v $A[i_0]$ je uloženo číslo jednoho z předmětů nějaké takové množiny, v $A[i_0 - m_{A[i_0]}]$ číslo dalšího předmětu atd. Zdrojový kód tohoto algoritmu lze nalézt na další straně.

Časová složitost algoritmu je $\mathcal{O}(NM)$, neboť se skládá z N kroků, z nichž každý vyžaduje čas $\mathcal{O}(M)$. Paměťová složitost činí $\mathcal{O}(N + M)$, což představuje paměť potřebnou pro uložení pomocného pole A a hmotností daných předmětů.

Cvičení a poznámky

- Proč pole A procházíme pozadu a ne popředu?
- Složitost algoritmu vypadá jako polynomiální, ale to je trochu podvod. Závisí totiž na hodnotě M. Pokud tuto hodnotu na vstupu zapíšeme obvyklým způsobem, tedy v desítkové nebo dvojkové soustavě, použijeme řádově $\log M$ cifer.

  Naše M proto bude vzhledem k délce vstupu až exponenciálně velké. To je typický příklad takzvaného *pseudopolynomiálního* algoritmu – tedy takového, jenž je vzhledem k hodnotám na vstupu polynomiální, ale k délce vstupu exponenciální. Podrobnosti si můžete přečíst v kuchařce o těžkých úlohách.[25]

```
var N: integer; {   počet předmětů   }
    M: integer; { hmotnostní omezení }
    hmotnost: array[1..N] of integer;
                { hmotnosti daných předmětů }
    A: array[0..M] of integer;
    i, k: integer;
begin
  A[0]:=-1;
  for i:=1 to M do A[i]:=0;
  for k:=1 to N do
    for i:=M downto hmotnost[k] do
      if (A[i-hmotnost[k]]<>0) and (A[i]=0) then
        A[i]:=k;
  i:=M; while A[i]=0 do i:=i-1;
  writeln('Maximální hmotnost: ',i);
```

[25] http://ksp.mff.cuni.cz/tasks/23/cook5.html

```
    write('Předměty v množině:');
    while A[i]<>-1 do begin
      write(' ',A[i]);
      i:=i-hmotnost[A[i]];
    end;
    writeln;
  end.
```

Nejkratší cesty a Floydův-Warshallův algoritmus

Náš další příklad bude z oblasti grafových algoritmů, ale zkusíme si jej nejdříve říci bez grafů:

Bylo-nebylo-je N měst. Mezi některými dvojicemi měst vedou (obousměrné) *silnice,* jejichž (nezáporné) délky jsou dány na vstupu. Předpokládáme, že silnice se jinde než ve městech nepotkávají (pokud se kříží, tak mimoúrovňově).

Úkolem je spočítat nejkratší vzdálenosti mezi všemi dvojicemi měst, tj. délky nejkratších cest mezi všemi dvojicemi měst.

Cestou rozumíme posloupnost měst takovou, že každá dvě po sobě následující města jsou spojené silnicí, a délka cesty je součet délek silnic, které tato města spojují.

V grafové terminologii tedy máme daný ohodnocený neorientovaný graf a chceme zjistit délky nejkratších cest mezi všemi dvojicemi jeho vrcholů.

Půjdeme na to následovně – vzdálenosti mezi městy jsou na začátku algoritmu uloženy ve dvourozměrném poli D, tj. $D[i][j]$ je vzdálenost z města i do města j. Pokud mezi městy i a j nevede žádná silnice, bude $D[i][j] = \infty$ (v programu bude tato hodnota rovna nějakému dostatečně velkému číslu).

V průběhu výpočtu si budeme na pozici $D[i][j]$ udržovat délku nejkratší dosud nalezené cesty mezi městy i a j.

Algoritmus se skládá z N fází. Na konci k-té fáze bude v $D[i][j]$ uložena délka nejkratší cesty mezi městy i a j, která může procházet skrz libovolná z měst $1, \ldots, k$.

V průběhu k-té fáze tedy stačí vyzkoušet, zda je mezi městy i a j kratší stávající cesta přes města $1, \ldots, k-1$, jejíž délka je uložena v $D[i][j]$, nebo nová cesta přes město k.

Pokud nejkratší cesta prochází přes město k, můžeme si ji rozdělit na nejkratší cestu z i do k a nejkratší cestu z k do j. Délka takové cesty je tedy rovna $D[i][k] + D[k][j]$.

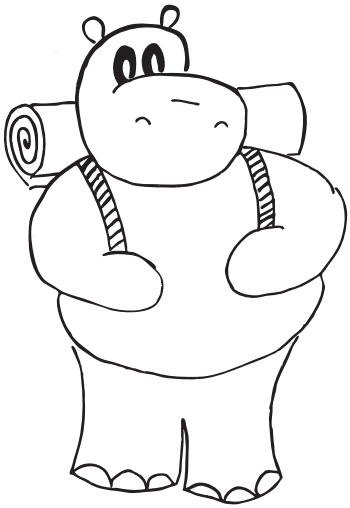
Takže pokud je součet $D[i][k] + D[k][j]$ menší než stávající hodnota $D[i][j]$, nahradíme hodnotu na pozici $D[i][j]$ tímto součtem, jinak ji ponecháme.

Z popisu algoritmu přímo plyne, že po N-té fázi je na pozici $D[i][j]$ uložena délka nejkratší cesty z města i do města j.

Protože v každé z N fází algoritmu musíme vyzkoušet všechny dvojice i a j, vyžaduje každá fáze čas $\mathcal{O}(N^2)$. Celková časová složitost našeho algoritmu tedy je $\mathcal{O}(N^3)$. Co se paměti týče, vystačíme si s polem D a to má velikost $\mathcal{O}(N^2)$.

Program bude vypadat následovně:

```
var N: integer; { počet měst }
    D: array[1..N] of array[1..N] of longint;
      { délky silnic mezi městy, D[i][i]=0,
        místo neexistujících je "nekonečno" }
    i, j, k: integer;
begin
  for k:=1 to N do
    for i:=1 to N do
      for j:=1 to N do
        if D[i][k]+D[k][j] < D[i][j] then
          D[i][j]:=D[i][k] + D[k][j];
end.
```



Popišme si ještě, jak bychom postupovali, kdybychom kromě vzdáleností mezi městy chtěli nalézt i nejkratší cesty mezi nimi.

To lze jednoduše vyřešit například tak, že si navíc budeme udržovat pomocné pole $E[i][j]$ a do něj při změně hodnoty $D[i][j]$ uložíme nejvyšší číslo města na cestě z i do j délky $D[i][j]$ (při změně v k-té fázi je to číslo k).

Máme-li pak vypsat nejkratší cestu z i do j, vypíšeme nejprve cestu z i do $E[i][j]$ a pak cestu z $E[i][j]$ do j. Tyto cesty nalezneme stejným (rekurzivním) postupem.

Poznámky

- Popis algoritmu vysloveně svádí k „rejpnutí“: Jak víme, že spojením dvou cest, které provádíme, vznikne zase cesta (tj. že se na ní nemohou nějaké vrcholy opakovat)?

  To samozřejmě nevíme, ale všimněme si, že kdykoliv by to cesta nebyla, tak si ji nevybereme, protože původní cesta bez vrcholu k bude vždy kratší nebo alespoň stejně dlouhá… tedy alespoň pokud se v naší zemi nevyskytuje cyklus záporné délky. To bychom měli přidat do předpokladů našeho algoritmu, kdybychom byli pedanti.
- Pozor na pořadí cyklů – program vysloveně svádí k tomu, abychom psali cyklus pro `k` jako vnitřní… jenže pak samozřejmě nebude fungovat.

Cvičení

- Jak by algoritmus fungoval, kdyby silnice byly jednosměrné?
- Na první pohled nejpřirozenější hodnota, kterou bychom mohli použít pro ∞, je `maxint`. To ovšem nebude fungovat, protože $\infty + \infty$ přeteče. Stačí `maxint div 2`?
- Hodnoty v poli si přepisujeme pod rukama, takže by se nám mohly poplést hodnoty z předchozí fáze s těmi z fáze současné. Ale zachrání nás to, že čísla, o která jde, vyjdou v obou fázích stejně. Proč?

Nejdelší společná podposloupnost

Poslední příklad dynamického programování, který si předvedeme, se bude týkat posloupností. Mějme dvě posloupnosti čísel A a B.

Chceme najít jejich nejdelší společnou podposloupnost, tedy takovou posloupnost, kterou můžeme získat z A i B odstraněním některých prvků. Například pro posloupnosti

$$A = 2\ 3\ 3\ 1\ 2\ 3\ 2\ 2\ 3\ 1\ 1\ 2$$
$$B = 3\ 2\ 2\ 1\ 3\ 1\ 2\ 2\ 3\ 3\ 1\ 2\ 2\ 3$$

je jednou z nejdelších společných podposloupností tato posloupnost:

$$C = 2\ 3\ 1\ 2\ 2\ 3\ 1\ 2.$$

Jakým způsobem můžeme takovou podposloupnost najít? Nejdříve nás asi napadne vygenerovat všechny podposloupnosti a ty pak porovnat.

Jakmile si ale spočítáme, že všech podposloupností posloupnosti o délce n je 2^n (každý prvek nezávisle na ostatních buď použijeme, nebo ne), najdeme raději nějaké rychlejší řešení.

Zkusme využít následující myšlenku: vyřešíme tento problém pouze pro první prvek posloupnosti A. Pak najdeme řešení pro první dva prvky A, přičemž využijeme předchozích výsledků. Takto pokračujeme pro první tři, čtyři, ... až n prvků.

Nejprve si rozmyslíme, co všechno si musíme v každém kroku pamatovat, abychom z toho dokázali spočíst krok následující. Určitě nám nebude stačit pamatovat si pouze nejdelší podposloupnost, jenže množina všech společných podposloupností je už zase moc velká.

Podívejme se tedy detailněji, jak se změní tato množina při přidání dalšího prvku k A: Všechny podposloupnosti, které v množině byly, tam zůstanou a navíc přibude několik nových, končících právě přidaným prvkem.

Ovšem my si podposloupnosti pamatujeme proto, abychom je časem rozšířili na nejdelší společnou podposloupnost.

Takže pokud známe nějaké dvě stejně dlouhé podposloupnosti P a Q končící nově přidaným prvkem v A a víme, že P končí v B dříve než Q, stačí si z nich pamatovat pouze P.

V libovolném rozšíření Q-čka totiž můžeme Q vyměnit za P a získat tím stejně dlouhou společnou podposloupnost.

Proto si stačí pro již zpracovaných a prvků posloupnosti A pamatovat pro každou délku l tu ze společných podposloupností $A[1 \dots a]$ a B délky l, která v B končí na nejlevějším možném místě.

Dokonce nám bude stačit si místo celé podposloupnosti uložit jen pozici jejího konce v B. K tomu použijeme dvojrozměrné pole $D[a, l]$.

Ještě si dovolíme jedno malé pozorování: Koncové pozice uložené v poli D se zvětšují s rostoucí délkou podposloupnosti, čili $D[a, l] < D[a, l + 1]$, protože posloupnosti délky $l + 1$ nejsou ničím jiným než rozšířeními posloupností délky l o 1 prvek.

Teď již výpočet samotný:

Pokud už známe celý a-tý řádek pole D, můžeme z něj získat $(a + 1)$-ní řádek. Projdeme postupně posloupnost B. Když najdeme v B prvek $A[a + 1]$ (ten právě přidávaný do A), můžeme rozšířit všechny podposloupnosti končící před aktuální pozicí v B.

Nás bude zajímat pouze ta nejdelší z nich, protože rozšířením všech kratších získáme posloupnost, jejíž koncová pozice je větší než koncová pozice některé posloupnosti, kterou již známe. Rozšíříme tedy tu nejdelší podposloupnost a uložíme ji místo původní podposloupnosti.

Toto provedeme pro každý výskyt nového prvku v posloupnosti B. Všimněme si, že nemusíme procházet pole s podposloupnostmi stále od začátku, ale můžeme se v něm posouvat od nejmenší délky k největší.

Ukážeme si, jak vypadá zaplněné pole hodnotami při řešení problému s posloupnostmi z našeho příkladu. Řádky jsou pozice v A, sloupce délky podposloupností.

D	1	2	3	4	5	6	7	8	9	10	11	12
1	2	–	–	–	–	–	–	–	–	–	–	–
2	1	5	–	–	–	–	–	–	–	–	–	–
3	1	5	9	–	–	–	–	–	–	–	–	–
4	1	4	6	11	–	–	–	–	–	–	–	–
5	1	2	5	7	12	–	–	–	–	–	–	–
6	1	2	3	7	9	14	–	–	–	–	–	–
7	1	2	3	7	8	12	–	–	–	–	–	–
8	1	2	3	7	8	12	13	–	–	–	–	–
9	1	2	3	5	8	9	13	14	–	–	–	–
10	1	2	3	4	6	9	11	14	–	–	–	–
11	1	2	3	4	6	9	11	14	–	–	–	–
12	1	2	3	4	6	7	11	12	–	–	–	–

Zbývá popsat, jak z těchto dat zvládneme rekonstruovat hledanou nejdelší společnou podposloupnost (NSP).

Ukážeme si to na našem příkladu – jelikož poslední nenulové číslo na posledním řádku je v 8. sloupci, má hledaná NSP délku 8.

$D[12, 8] = 12$ říká, že poslední písmeno NSP je na pozici 12 v posloupnosti B. Jeho pozici v posloupnosti A určuje nejvyšší řádek, ve kterém se tato hodnota také vyskytuje, v našem případě je to řádek 12. Druhé písmeno tedy budeme určovat z $D[10, 7]$, třetí z $D[9, 6]$, atd.

Jednou z hledaných podposloupností je tedy:

poslupnost:	2	3	1	2	2	3	1	2
indexy v A:	1	2	4	5	7	9	10	12
indexy v B:	2	5	6	7	8	9	11	12

Již zbývá jen odhadnout složitost algoritmu. Časově nejnáročnější byl vlastní výpočet hodnot v poli, který se skládá ze dvou hlavních cyklů o délce $|A|$ a $|B|$, což jsou délky posloupností A a B.

Vnořený cyklus while proběhne celkem maximálně $|A|$-krát a časovou složitost nám nezhorší. Můžeme tedy říct, že časová složitost je $\mathcal{O}(|A| \cdot |B|)$.

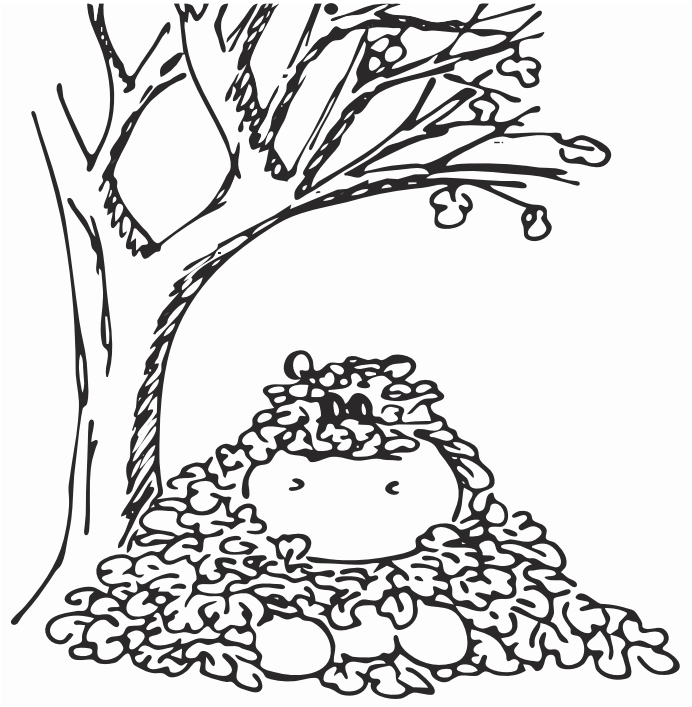
Posloupnosti jsme si prohodili tak, aby první byla ta kratší, protože pak je maximální délka společné podposloupnosti i počet kroků algoritmu roven délce kratší posloupnosti, a tedy i velikost pole s daty je kvadrát této délky.

Paměťovou složitost odhadneme $\mathcal{O}(N^2 + M)$, kde N je délka kratší posloupnosti a M té delší.

```
program Podposloupnost;
var
  A, B, C: array[0..MaxN - 1] of Integer;
  LA, LB, LC: Integer;  { Délky posloupností }
  D: array[0..MaxN, 1..MaxN] of Integer;
  I, J, L, MaxL, T: Integer;
begin
  ...
  if LA > LB then begin { A bude kratší z obou }
    C := A;
    A := B;
    B := C;
    T := LA;
    LA := LB;
    LB := T;
  end;

  for I := 1 to LA do
    D[0, I] := LB;

  L := 0;
  MaxL := 0;
  for I := 1 to LA do begin
    for J := 1 to LA do
      D[I, J] := D[I-1, J];

    L := 0;
    for J := 0 to LB-1 do
      if B[J] = A[I-1] then begin
        while (L = 0) or (D[I-1, L] < J) do
          L:=L+1;
        if D[I, L] >= J then
          D[I, L] := J;
```

```
        end;
        if L > MaxL then MaxL := L;
    end;

    LC := MaxL;
    J := LA;
    for I := LC downto 1 do
    begin
      while D[J-1, I] = D[J, I] do J:=J-1;
      C[I-1] := A[J-1];
      J:=J-1;
    end;
    ...
end.
```

Dnešní menu servírovali
Martin Mareš a Petr Škoda



Kuchařka třetí série – intervalové stromy

Intervalové stromy

Představme si, že máme posloupnost celých čísel

$$p_1, p_2, \ldots, p_N,$$

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 1 až do indexu i.

Tedy `pref[i] = p[1] + ... + p[i]`; z praktických důvodů dodefinujeme `pref[0] = 0`.

Není těžké si rozmyslet, že toto pole dokážeme jednoduše spočítat v čase $\mathcal{O}(N)$.

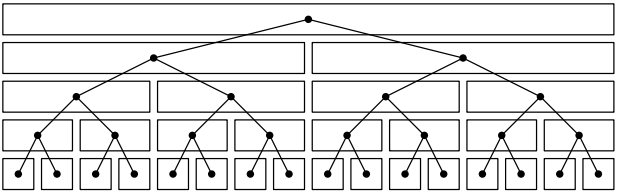
Nyní, když už známe všechny prefixové součty posloupnosti, umíme snadno spočítat součet na libovolném intervalu $[a, b]$:

```
s[a,b] = pref[b] - pref[a-1]
```

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 povolíme i měnit čísla v posloupnosti, pokazíme si časovou složitost. S prefixovými součty stále dokážeme dotaz na součet podposloupnosti provádět v konstantním čase, ale při změně čísla v posloupnosti 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 $\mathcal{O}(NZ + D)$.

S touto složitostí se samozřejmě nespokojíme a budeme se snažit, abychom výsledné intervaly uměli co nejrychleji skládat z předpočítaných hodnot, a tedy 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.

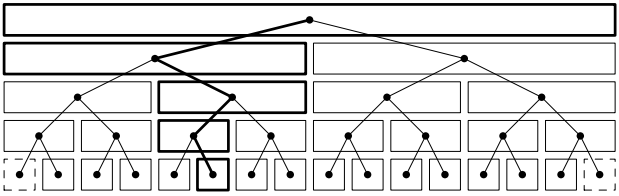
Můžeme ale klidně mít třeba strom, který si pamatuje, jestli celý jeho interval obsahuje jen jednu hodnotu, a pokud ano, tak jakou.

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.

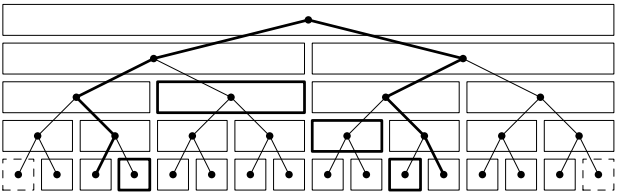


Změna hodnoty na indexu 5 *– musíme změnit vyznačené vrcholy – intervaly* $[5,5]$, $[4,5]$, $[4,7]$, $[1,7]$ *a* $[1,14]$

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. Postupně tak poskládáme celý interval.



Výběr intervalu $[3, 10]$. *Jeho součet spočítáme přes vyznačené intervaly:* $[3, 3]$, $[4, 7]$, $[8, 9]$ *a* $[10, 10]$.

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 $\mathcal{O}(\log N)$, neboť 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 ukládá 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 $N, \ldots, 2N - 1$. 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 samotnou implementaci v jazyce C. Pozor, prvky posloupnosti indexujeme od 1.

```
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);
```

```
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];
    a = a/2; b = b/2; // Přesun na otce
  }
  // 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, kde upravujeme strom od kořene směrem dolů.

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 neexistuje elegantnější či jednodušší řešení – jestli nejdete kanónem na vrabce.

Navíc se některé druhy intervalových stromů implementují velmi obtížně a za tu práci to nestojí.

Intervalový strom obvykle použijeme, pokud potřebujme průběžně zjišťovat informace o intervalech a zároveň je i měnit. Například 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 zvaný *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 však nepřišli o pověstný „aha-efekt“, obrátíme běžný postup vysvětlování. Nejdříve si ukážeme, jak se Fenwickův strom implementuje, a teprve pak dokážeme, že ta magie opravdu 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);   // bitový and
  }
}
```

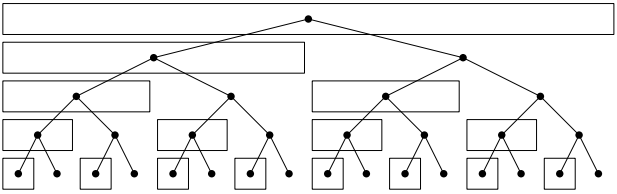
A zde je funkce pro zjištění prefixového součtu:

```
int prefSoucet(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 zprava v binárním zápisu čísla N. Ve stromě máme tedy uloženou takovou pravidelnou strukturu intervalů.



Nyní se podíváme, co dělají naše magické funkce na posouvání ve stromě.

Ve výrazu `index & (index-1)` z funkce `prefSoucet()` se vynuluje nejpravější jedničkový bit v indexu. Tím se dostaneme na první interval, který jsme ještě nepřičetli. Jakmile máme `index == 0`, můžeme ukončit výpočet, neboť již máme interval celý sečtený.

Výraz `index + (index & -index)` dělá to, že se v pomyslném stromě intervalů posune o úroveň výš. 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

Kuchařka čtvrté 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 adresáře (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 `01` pro čísla v binárním zápisu, klasické `A-Za-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 samotná 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 značit dále 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, tak 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 (kromě binárního `0` $<$ `1` se často používá „telefonní“ `A` $=$ `a` $<$ `B` $=$ `b` $<$ $\ldots$ $<$ `Z` $=$ `z`, které je ovšem lineární až na velikost znaků).

Když máme zadané uspořádání na znacích, na všechny řetězce jej rozšíříme následovně: nejkratší je prázdný řetězec a ostatní řetězce třídíme podle znaků od začátku do konce. Zvláštnost je v tom, že řetězec je větší než jeho každá vlastní předpona (neboli *prefix*). Řetězec `A` tedy bude menší než `AUTO`, které samo bude menší než `AUTOBUS`.

Adresář pomocí trie

Typický problém v oblasti textu je, že máme seznam nějakých řetězců (často třeba jmenný adresář), můžeme si jej nějak předzpracovat, a pak bychom rádi efektivně odpovídali na otázku: „Je řetězec S obsažen v adresáři?“ Můžeme také po předzpracování chtít přidávat nové položky i odebírat staré.

Pokud bychom nemuseli odebírat jména, můžeme použít hešování, které je rychlé a účinné. Více o něm najdete v kuchařce o hešování.[26] Má však tu nevýhodu, že při velkém zaplnění se začne chovat pomaleji a mírně nepředvídatelně.

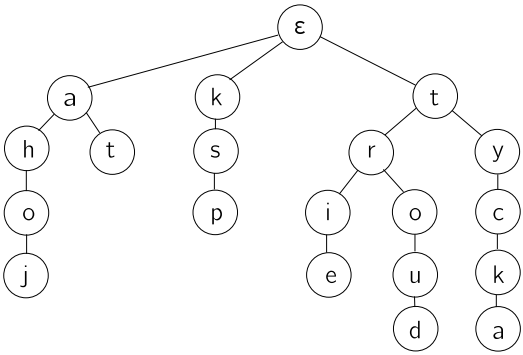
[26] `http://ksp.mff.cuni.cz/viz/kucharky/hesovani`

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, budeme jej stavět pro nějaký adresář A. Kořen bude odpovídat prázdnému slovu ε. Každá hrana, která z něj povede, odpovídá jednomu ze znaků, kterým slovo z adresáře A začíná, a to bez opakování (tedy jsou-li v A čtyři slova začínající na `A`, hranu vedeme jen jednu).

Na koncích těchto hran z kořene nám vznikly vrcholy, které odpovídají všem jednoznakovým prefixům slov z A, a už je celkem jasné, jak struktura dále pokračuje – z každého vrcholu odpovídajícímu prefixu P vede hrana se znakem c právě tehdy, když slovo $P + c$ (za P přilepíme znak c) je také prefixem některého slova z A.

Obrázek vydá za tisíc definic, zde je postavená trie pro slova `AHOJ`, `AT`, `KSP`, `TRIE`, `TROUD`, `TYC`, `TYCKA`:



Jak bychom takovou trii postavili algoritmem? Přesně, jak jsme ji definovali: každé slovo z adresáře 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 z adresáře 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, anebo si rozšíříme abecedu o speciální znak, který se v ní předtím nevyskytoval – třeba `$` – a pak všem slovům z A přilepíme tento `$` na konec.

Budeme-li se později ptát, bylo-li slovo v adresáři, po průchodu trií zkontrolujeme ještě, jestli z konečného vrcholu vede hrana odpovídající znaku `$`.

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) na $\mathcal{O}(D \cdot |\Sigma|)$, kde D značí velikost vstupu, čili součet délek všech slov v adresáři. To je naprosto přijatelné pro malé abecedy, ale už pro `A-Za-z` je tento faktor roven 52 a pro Unicode je už taková alokace nemyslitelná.

Pokud tedy pracujeme s velkou abecedou, může se nám vyplatit oželet konstantní rychlost dotazu a použít v každém vrcholu vlastní binární vyhledávací strom pro znaky, kterými aktuální prefix může pokračovat. To zmírní časovou složitost konstrukce na $\mathcal{O}(D \cdot \log |\Sigma|)$ a zhorší časovou složitost dotazu na slovo délky L na $\mathcal{O}(L \cdot \log |\Sigma|)$.

A jsme hotovi! S trií můžeme v lineárním čase odpovídat na dotazy „Vyskytuje se dané slovo v adresáři?“, 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 v adresáři. (Slovo *prefixové* je však v matematice hodně nadužívané (prefixová notace, prefixové kódy), a tak to může vést ke zmatení.)
- Kdybychom chtěli, mohli bychom pomocí trie vyhledávat v českém textu v lineárním čase. Můžeme přeci postavit adresář ze všech slov v daném textu, a pak procházet tu 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 *Grafové algoritmy*.[27]

Cvičení

- Řekněme, že chceme adresář na vstupu setřídit v lexikografickém pořadí (definovaném v sekci „Jak řetězce chápat“). Můžeme použít nějaký klasický třídicí

[27] `http://mj.ucw.cz/vyuka/ga/`

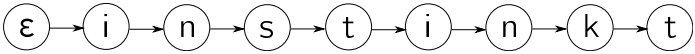
algoritmus, ale bohužel musíme počítat s tím, že porovnání dvou řetězců není konstantně rychlé. Vymyslete způsob, jak setřídit takový adresář pomocí trie.

- *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. Chceme si slovo zpracovat, načež projdeme co nejrychleji text a zahlásíme jeden nebo všechny jeho výskyty. 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“, a tedy se textu přezdívá *seno* a hledanému slovu *jehla*. Délku jehly označíme D a délku textu H.

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 $\mathcal{O}(H \cdot D)$, avšak stačí malá úprava a složitost přejde na lineární $\mathcal{O}(H + D)$. 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*.

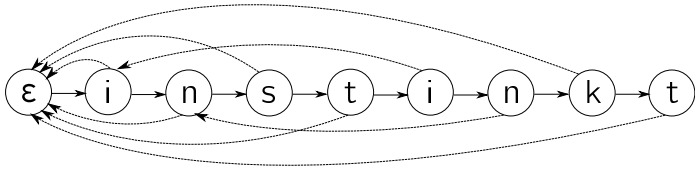
Zatím jsme ale přesně nepopsali, na které políčko přesně bude zpětná funkce ukazovat. Nechť chceme určit zpětné políčko pro druhé `N` ve slově `INSTINKT`. Pracujeme teď s prefixem `INSTIN`. Selsky řečeno, chceme najít „konec slova `INSTIN` takový, že je stejný, jako začátek slova `INSTIN`“.

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? 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 D-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, $\mathcal{O}(H)$.

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 celou jehlu bez prvního znaku a sledovat, jakými stavy bude procházet, a to budou přesně hodnoty zpětné funkce.

Vytvoření zpětné funkce se nám tak nakonec zredukovalo na jediné vyhledávání v textu o délce $D-1$, a proto poběží v čase $\mathcal{O}(D)$. Časová složitost celého algoritmu tedy bude $\mathcal{O}(H+D)$. 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..D] of char;   { jehla }
  Text: array[1..H] of char;    { seno }
  F: array[1..D] of integer; { zpětná fce }
  function Krok(I: integer; C: char): integer;
  begin
    if (I < D) and (Slovo[I+1] = C) then
      Krok := I + 1
    else if I > 0 then
      Krok := Krok(F[I], C)
```

```
    else
      Krok := 0;
  end;
var I, J: integer;  { pomocné proměnné }
begin
  { konstrukce zpětné funkce }
  F[1]:= 0;
  for I:= 2 to D do
    F[I]:= Krok(F[I-1], Slovo[I]);
  { procházení textu }
  J:= 0;
  for I:= 1 to H do begin
    J:= Krok(J, Text[I]);
    if J = D 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.

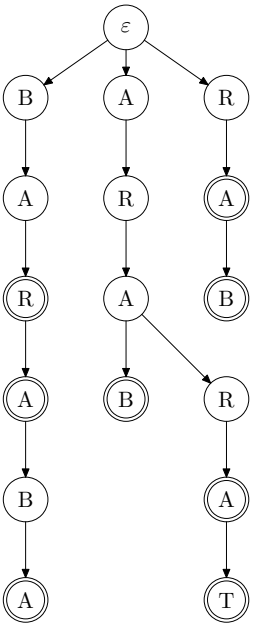
Cvičení

- 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, jako jsme řešili 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.

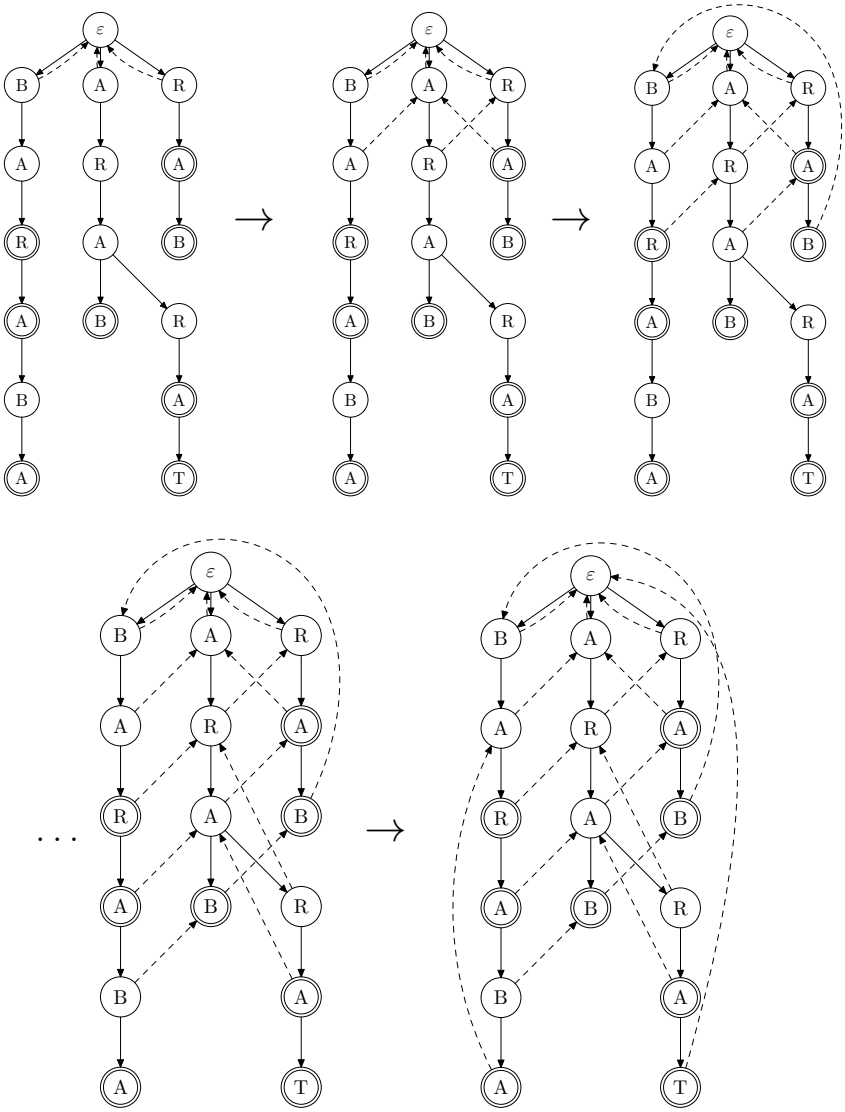
Zpětné hrany totiž nemusí vést do předka. Například pro slovo `BARAB` povede zpětná hrana do slova `ARAB`, z toho do slova `RAB` a z toho do `B`. Kdybychom ale zkonstruovali automat výše popsaným způsobem (a začali slovem `BARAB`), nebude existovat v trii ani `ARAB`, ani `RAB`, takže bychom vedli zpětnou hranu chybně do `B`.

Můžeme se ale opřít o trik z konstrukce KMP – vyhledání svého nejdelšího vlastního suffixu. 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 tedy 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.



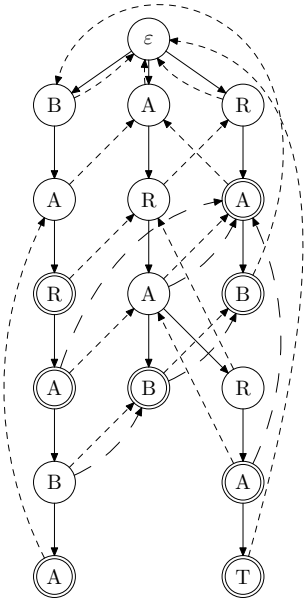
Ještě zbývá otázka, jak konstruovat zpětné hrany efektivně, když je musíme vyrábět po vrstvách. Mohli bychom prostě vzít slovo, pro které hledáme zpětnou hranu, utrhnout mu první znak a vyhledat. Jenže to budeme dělat spoustu práce zbytečně.

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`?

Najdeme tedy akorát, kde jsme minule skončili, a odtamtud pokračujeme dál. Jak to najdeme? Z otce našeho vrcholu tam přece vede zpětná hrana. Takže můžeme postup 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 $\mathcal{O}(D \cdot |\Sigma|)$, resp. $\mathcal{O}(D \cdot \log |\Sigma|)$ (pokud použijeme binární strom ve vrcholech) a z předpočítání zpětných hran. Při předpočítávání uděláme nějaký konstantní počet operací pro každý vrchol (celkem tedy $\mathcal{O}(D)$) a také paralelně vyhledáváme všechny jehly z jehelníčku, jejichž vyhledání nás stojí $\mathcal{O}(D)$, resp. $\mathcal{O}(D \cdot \log |\Sigma|)$.



Tedy konstrukce trvá celkem $\mathcal{O}(D \cdot |\Sigma|)$, resp. $\mathcal{O}(D \cdot \log |\Sigma|)$, paměťová náročnost je stejná jako u trie – $\mathcal{O}(D \cdot |\Sigma|)$, resp. $\mathcal{O}(D)$, přidali jsme jen $\mathcal{O}(D)$ 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 $D-1$). Budeme-li jím vyhledávat v textu `AAAA...A` délky $H > D$, projdeme prakticky pro každý znak až $D-1$ zpětných hran, čímž složitost naroste až na nepoužitelných $\mathcal{O}(H \cdot D)$.

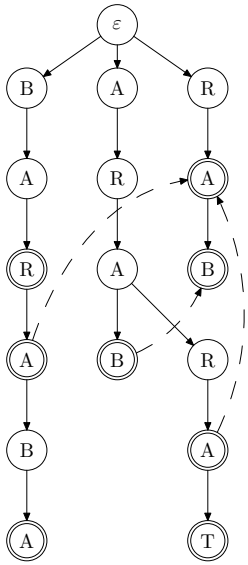
Všimněme si však, že většinou zpětných hran jsme prošli úplně zbytečně. Předpočítáme si tedy *zkratky* – z vrcholu vede zkratka do nejdelšího jeho suffixu, který je jehlou. Na obrázku jsou vyznačeny dlouze čárkovanými šipkami.

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 $\mathcal{O}(H + O)$, resp. $\mathcal{O}(H \cdot \log|\Sigma| + O)$, kde O je velikost výstupu – počet výskytů všech slov.

Celková časová složitost prohledávání včetně stavby automatu tedy bude $\mathcal{O}(O+H+D \cdot |\Sigma|)$, resp. $\mathcal{O}(O+(H+D) \cdot \log|\Sigma|)$.

Jak velký může být výstup? Obecně až $\mathcal{O}(H^2)$). Extrémně velký výstup je možné vygenerovat například slovníkem obsahujícím všechny prefixy slova `AAAA...A` délky H a senem taktéž `AAAA...A` délky H. Automat pak hlásí výskyt pro každé podslovo, kterých je $\mathcal{O}(H^2)$.



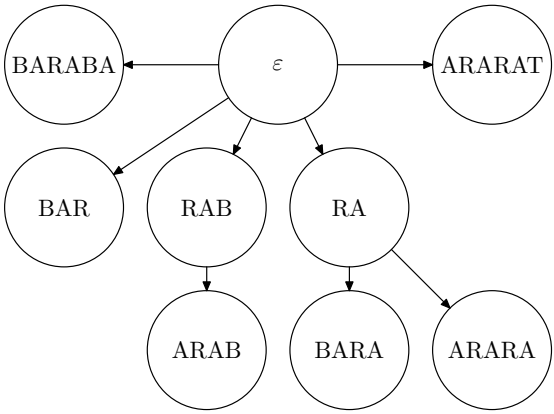
Pokud nám stačí u každého slova jen počet výskytů, nemusíme zoufat – závislost na počtu výskytů umíme odstranit.

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` 3 výskyty a `RAB` 1 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

Kuchařka páté série – geometrie

Geometrické algoritmy

V dnešním díle našeho kuchařkového speciálu se budeme učit vařit geometrické problémy. A co že si představujeme pod pojmem geometrický problém? Trochu analytické geometrie, například zjištění, na které straně orientované přímky bod leží, trocha plotů, neboli konvexních obalů, a obecně mnoho zametání.

V celé kuchařce se omezíme pouze na dvourozměrné problémy, tedy na algoritmy v rovině. Některé postupy se dají zobecnit pro trojrozměrné, a většinou i pro n-rozměrné problémy, ale to je již nad rámec této kuchařky.

Geometrické základy

Nejdříve trocha středoškolské analytické geometrie pro ty, kdo ji ještě neměli. Ostatní mohou tuto sekci přeskočit.

Každý bod v rovině můžeme určit jeho souřadnicemi vůči osám. Nejběžněji se používá takzvaný *kartézský souřadný systém,* tedy dvě na sebe kolmé osy označované jako x-ová osa (vodorovná) a y-ová osa (svislá). Obvykle se uvažuje, že hodnoty na osách rostou směrem doprava (osa x) a směrem nahoru (osa y), my se toho budeme v naší kuchařce držet.

Místo, kde se obě osy protínají, se označuje jako *počátek* soustavy souřadnic. Samotné *souřadnice* bodu zapisujeme jako dvojici čísel, která udávají, o kolik jednotek se musíme posunout ve směru které z os, abychom z počátku dorazili do bodu, kterému souřadnice patří. Počátek má souřadnice $[0, 0]$. Bod se souřadnicemi $[a, b]$ leží na pozici, kterou získáme tak, že se od počátku posuneme o a jednotek ve směru první osy (x-ové) a o b jednotek ve směru druhé osy (y-ové).

Vše ostatní funguje tak, jak jsme se učili při geometrii na základní škole, tedy úsečka je určena dvěma krajními body, obdélník čtyřmi a podobně. Ještě si ale řekneme, co je to vektor, a zavedeme některé další pojmy.

Často potřebujeme popsat vzájemnou polohu dvou bodů. Můžeme například udat jejich vzdálenost a směr (třeba jako úhel vzhledem k ose x). Praktičtější ale bývá říci, o kolik se liší jejich x-ové a y-ové souřadnice. To nám dá dvojici čísel, které říkáme *vektor.*

Pokud například k bodu $[1, 1]$ přičteme vektor $a = (2, -1)$, dostaneme se do bodu $[3, 0]$. Stejně tak, pokud odečteme například bod $[4, 2]$ od bodu $[1, 3]$, tak dostaneme vektor $b = (-3, 1)$ udávající jejich vzájemnou polohu.

Pomocí vektoru a bodu tedy lze určit přímku. Bod nám určí, kam umístit vektor, a vektor nám určí směr přímky z daného bodu. Tomuto vektoru se říká *směrový vektor,* nebo také někdy *směrnice,* dané přímky nebo úsečky.

Samotné vyjádření přímky nebo úsečky poté může být ve dvou tvarech. Prvním z nich je *parametrický tvar.* Základem je nějaký bod $A = [a_x, a_y]$. Od toho se ve směru směrového vektoru $u = (u_x, u_y)$ můžeme pohybovat libovolně a stále budeme na přímce. To nám vede na následující tvar, kde t je libovolný reálný parametr, neboli proměnná, za kterou si můžeme dosadit jakékoliv reálné číslo a vždy nám vyjde bod na přímce. Parametrický tvar vypadá:

$$x = a_x + tu_x$$

$$y = a_y + tu_y$$

To samé můžeme vyjádřit i vektorově, tedy $X = A + tu$.

Pro ilustrování funkce parametru, když bude $t = 0$, tak dostaneme výchozí bod přímky. Pokud poté budeme s parametrem hýbat od $-\infty$ do $+\infty$, dostaneme postupně všechny body na přímce.

Druhým způsobem zápisu je *obecný tvar přímky.* K jeho vyjádření budeme potřebovat kolmý vektor ke směrovému vektoru, tomu se také říká *normálový vektor.* V rovině ho získáme jednoduše. Pokud je $v = (v_x, v_y)$ směrnice přímky, tak vektor na něj kolmý má tvar $n = (v_y, -v_x)$. Jako poznámku pro zvídavé můžeme uvést, že *skalární součin* těchto vektorů, tedy součin po složkách ($v \cdot n = ab + b(-a)$), je roven 0, což je také jedna z definic kolmosti.

A jak tedy vypadá slibovaný obecný tvar přímky? Pokud je $n = (a, b)$ normálový vektor přímky, tak obecný tvar přímky je rovnice $ax + by + c = 0$. Dobře, a a b máme, jak ale zjistit c? Normálový vektor určuje směr, kterým přímka povede, ale stále ji můžeme libovolně posouvat. Potřebujeme ještě znát jeden bod, který na naší přímce leží, aby byla určená jednoznačně.

Když dosadíme souřadnice takového bodu do rovnice přímky s neznámou c, získáme tak rovnici pro c, kterou vyřešíme. A máme hotovo, známe hodnoty všech koeficientů v rovnici. Ještě si můžeme všimnout, že pro $c = 0$ prochází přímka počátkem.

Takovéto tvary se hodí jednak pro nějaké zapsání přímek, ale také pro *zjištění jejich průsečíku.* Když hledáme průsečík, hledáme vlastně místo, kde mají obě přímky navzájem stejné x-ové a y-ové souřadnice. A to vede na jednoduché soustavy lineárních rovnic, které jistě již vyřešit umíte.

Ještě si ale zdůrazněme rozdíl úseček oproti přímkám. V případě parametrického tvaru omezujeme velikost parametru t (například $t \in \langle 0, 1 \rangle$) a v případě obecného tvaru omezujeme rozsah jedné ze souřadnic (například $x \in \langle -2, 2 \rangle$). V případě, že bychom chtěli vyjádřit polopřímku, si parametr nebo souřadnici omezíme pouze z jedné strany.

Nakonec si ukážeme jednu základní aplikaci parametru a parametrického vyjádření úsečky. Jak snadno spočítat střed nějaké úsečky AB? V takovém případě

není nic jednoduššího, než si vzít vektor $B - A$, přenásobit ho parametrem $1/2$ (střed úsečky je v polovině její délky) a přičíst k bodu A. Triviální úpravou pak zjistíme, že střed úsečky můžeme spočítat jako aritmetický průměr jejích krajních bodů:

$$A + \frac{1}{2} \cdot (B - A) = \frac{A + B}{2}$$

Jako příklad na rozkoukání si ukážeme, jak zjistit, na které straně přímky leží bod.

Zjištění polohy bodu vůči přímce

Nejdříve si zavedeme pojem orientovaná přímka. Když budeme mít přímku určenou dvojicí bodů A a B, budeme se na ni dívat, jako kdybychom stáli v prvním bodě (bod A) a dívali se směrem ke druhému (bod B). Pak již máme jasně definovanou pravou a levou stranu a můžeme říci, kde vůči přímce bod leží.

Vezměme si tedy přímku určenou body A a B a bod X. Určíme si vektory $u = X - A$ a $v = B - A$ (s prvky u_x, u_y, respektive v_x, v_y) a porovnáme úhel mezi nimi.

Pokud jste už měli analytickou geometrii, určitě znáte vzoreček na výpočet úhlu mezi dvěma vektory. Vzoreček má tvar:

$$\cos \alpha = \frac{u \cdot v}{|u||v|}$$

Jeho nevýhodou je, že výpočet inverzní funkce $\cos^{-1}$ trvá dlouho. Je proto lepší použít jiný způsob výpočtu, kde si vystačíme pouze s násobením.

Tím jiným způsobem je výpočet determinantu matice určené těmito vektory. *Matice* je pouze tabulka, kde jsou vektory poskládány pod sebe (ta naše tedy bude velká 2 na 2 políčka).

Determinant matice této velikosti nám udává obsah rovnoběžníku určeného zadanými vektory. A navíc znaménko determinantu nám říká, jestli je úhel mezi vektory (měřený v kladném směru, tedy proti směru hodinových ručiček) menší než π, nebo větší než π.

Kdo se ještě s determinanty nesetkal, může brát následující vzorec pro výpočet determinantu matice dva krát dva jako kouzelnou formuli. Kdo přesto chce zdůvodnění, může si zkusit udělat rozbor všech vzájemných poloh dvou přímek (a jejich směrových vektorů), které mohou nastat. Po chvíli dojdete ke vztahu přesně odpovídajícímu následujícímu vzorečku:

$$d = u_x v_y - u_y v_x.$$

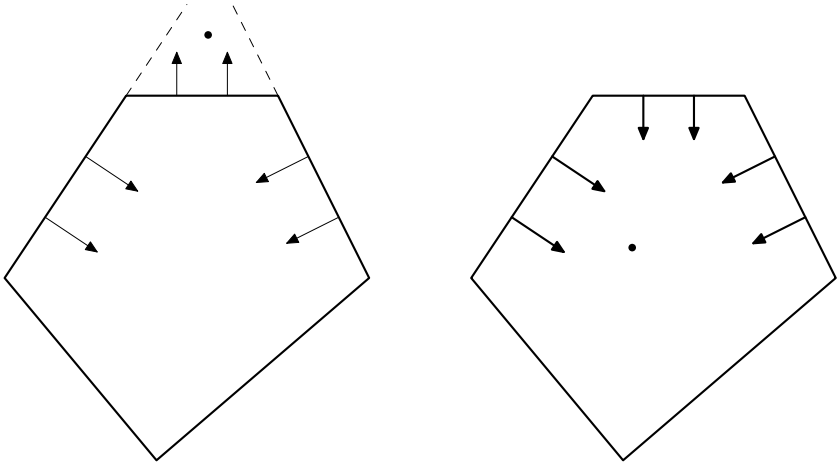
Pokud vyjde d kladné, je bod napravo od přímky, pokud vyjde d záporné, je bod nalevo od přímky, a konečně, pokud vyjde $d = 0$, tak bod leží na přímce.

Bod a konvexní mnohoúhelník

Konvexní mnohoúhelník je takový, který nemá žádný vnitřní úhel větší než 180°. Jinou definicí je, že pokud si zvolíme libovolné dva body v mnohoúhelníku a natáhneme úsečku mezi nimi, nikdy nám nevyleze z mnohoúhelníku ven.

Když už víme, co konvexní mnohoúhelník je, jak zjistíme, jestli nějaký bod leží v něm nebo ne? Využijeme vlastnosti konvexnosti. Stačí nám jít po hranách na obvodu a zjišťovat, jestli hledaný bod leží na stejné straně všech hran (tedy přímek určených koncovými body hran), nebo neleží.

Pokud bod leží na stejné straně všech hran, nachází se uvnitř mnohoúhelníku. Pokud se ale vůči jen jediné hraně nachází na jiné straně než vůči ostatním, leží bod vně mnohoúhelníku. Nejlépe to vysvětlí obrázek:



Tomuto postupu se také někdy říká test polorovinami. Každá kontrola nám zabere konstantně mnoho času. Časová složitost tohoto postupu je tedy lineární vzhledem k počtu hran, neboli $\mathcal{O}(N)$.

Pro nekonvexní útvary je již postup o něco těžší, jednoduše si můžeme všimnout, že postup s kontrolováním polohy bodu vůči všem hranám nebude fungovat. Zkuste si postup pro nekonvexní obrazce rozmyslet sami. Můžete buď využít testu polorovinami, jako v případě konvexního obrazce, nebo využít zajímavé vlastnosti průsečíků hran obrazce s náhodně vedenou polopřímkou.

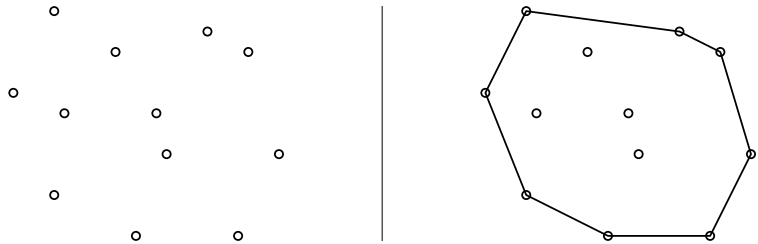
Pokud se vám o tom nechce přemýšlet, můžete se podívat na vzorové řešení úlohy 24-4-2.[28]

[28] `http://ksp.mff.cuni.cz/viz/24-4-2/reseni`

Konvexní obal a zametání roviny

Podíváme se na jeden z nejznámějších geometrických problémů, totiž hledání konvexního obalu množiny bodů v rovině. *Konvexní obal* je nejmenší konvexní mnohoúhelník, který obsahuje všechny zadané body. Můžeme si všimnout, že všechny vrcholy výsledného mnohoúhelníka musí být nějaké body ze zadané množiny, jinak bychom mohli mnohoúhelník ještě zmenšit (a nebyl by to konvexní obal).

Jako motivaci si představte třeba situaci, že máte sad ovocných stromů a chcete je oplotit co nejkratším plotem. Jak takový plot, nebo obecně obal, nalézt?



Vlevo neobalené body, vpravo obalené.

Ukážeme si postup, kterému se říká *zametání roviny.* Je to trik, který najde uplatnění u mnoha různých geometrických problémů a vyplatí se ho umět.

Základní myšlenka spočívá v tom, že nějakou přímkou, říkejme jí *zametací přímka,* přejedeme přes celou rovinu (od minus nekonečna do plus nekonečna, zleva doprava nebo shora dolů) a vždy když zametací přímka protne nějaký pro nás zajímavý bod, zpracujeme příslušnou událost. *Událost* je něco významného, co souvisí s příslušným bodem (průsečík přímek, vrchol mnohoúhelníka apod.)

Ale jak jet přímkou postupně od minus nekonečna do plus nekonečna? To není vůbec nutné. Pohyb přímky můžeme začít v nějakém startovním bodě (většinou první událost v setříděné posloupnosti událostí) a ukončit ho po zpracování všech událostí. Navíc nebudeme přímkou pohybovat plynule, ale budeme jí vždy skákat z události na událost (protože mezi událostmi se nic zajímavého neděje).

Vraťme se k našemu problému s konvexním obalem. Jako události budeme brát všechny body, které dostaneme na vstupu. V tomto případě nám žádné nové události v průběhu výpočtu vznikat nebudou, takže frontu událostí můžeme implementovat jako lineární spojový seznam.

Na začátku si body setřídíme podle jejich x-ové souřadnice (zatím budeme pro jednoduchost předpokládat, že žádné dva body nemají stejnou x-ovou souřadnici), začneme je zametací přímkou postupně procházet zleva doprava a budeme si udržovat konvexní obal bodů, které jsme už zpracovali.

V průběhu výpočtu si budeme konstruovat horní a dolní *obálku.* Obě obálky budou určitě začínat v nejlevějším a končit v nejpravějším bodě (jednoduchým pozorováním lze nahlédnout, že tyto body do obalu určitě patří). A jak už název napovídá, horní obálka půjde vrchem a bude se zatáčet stále doprava, a dolní obálka naopak půjde spodem a bude se stále zatáčet doleva.

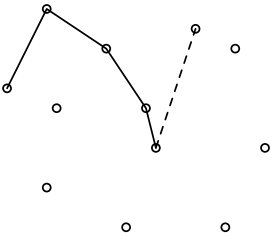
Můžeme se pro zjednodušení dohodnout, že nejlevější i nejpravější bod patří do obou obálek. Když pak horní a dolní obálku spojíme, dostaneme konvexní obal.

Horní (respektive dolní) obálku si budeme udržovat jako lineární seznam vrcholů.

Teď si ukážeme, jak bude probíhat jeden krok zpracování. Výpočet se bude provádět samostatně pro horní a dolní obálku, my si ho ukážeme jen pro horní (pro dolní je až na zrcadlení stejný).

Uvažujme, že už máme nějakou část horní obálky, skočili jsme zametací přímkou na další bod a ten teď chceme přidat. Podíváme se na poslední bod v horní obálce a zkontrolujeme úhel poslední hrany v obálce a úsečky mezi posledním bodem obalu a novým bodem.

K tomu můžeme využít například test polorovinami z úvodu kuchařky (pokud nový bod leží vůči poslední hraně horní obálky napravo, je vnitřní úhel konvexní, pokud nalevo, je úhel konkávní). Jestliže se horní obálka zatáčí doprava, máme vyhráno, přidáme nový bod do obálky a můžeme se posunout na další bod. Zajímavější je ale situace, kdy se nám obálka stočí doleva a vznikne konkávní úhel.



Pokud se podíváme na obrázek výše, jasně vidíme, že je potřeba dosavadní poslední bod obálky odebrat a zkusit spojit nově přidávaný bod s předposledním. Odstraníme tedy poslední bod obálky a budeme test opakovat s předposledním bodem.

Takto budeme pokračovat (a případně vyhazovat další body), než buď bude úhel hran konvexní, nebo dokud nám v obálce nezůstane pouze jeden bod (počáteční). Pak nový bod přidáme do obálky a pokračujeme s dalším.

Výše popsaný postup je nejvýhodnější provádět najednou pro obě dvě obálky. Tedy každý bod se pokusím připojit k horní i dolní obálce a podle toho obě obálky příslušně upravím.

Proč tento postup funguje? Postupně projdeme všechny body a každý z nich se alespoň na chvíli stane posledním bodem obálky. Při změně obálky se obsažená plocha v konvexním obalu vždy pouze zvětší a žádný bod tedy nám tedy nemůže zůstat mimo konvexní obal.

Ještě jsme zapomněli na případ, kdy úhel není ani konvexní, ani konkávní. V takovém případě se rozhodneme, jestli budeme vrchol tohoto úhlu započítávat mezi vrcholy konvexního obalu. Obvykle se takový vrchol z konvexního obalu vyhazuje, ale nakonec vždycky záleží, k čemu ten konvexní obal vlastně potřebujeme.

Skončíme, až zametací přímkou skočíme na poslední bod a zpracujeme ho. V tomto bodě se nám obálky spojí a dostaneme celý konvexní obal. Teď ale přichází otázka, kolik času nám tento postup zabere?

Může se zdát, že hodně, protože při vyhazování bodů z obálky můžeme postupně vyhodit skoro všechny body. Označme si velikost zadané množiny (počet bodů na vstupu programu) N. Musíme si uvědomit, že každý bod do obálky přidáme pouze jednou a vyhodíme ho také maximálně jednou, tedy časová složitost je lineární k velikosti množiny, tedy $\mathcal{O}(N)$, v případě, že již máme setříděný vstup. Pokud ne, musíme ještě přičíst čas potřebný k setřídění bodů, tedy $\mathcal{O}(N \log N)$ při použití nějakého rychlého třídicího algoritmu.[29]

Nakonec ještě zbývá dořešit více bodů se stejnou x-ovou souřadnicí. Pokud to nejsou krajní body, tak nám to v postupu nevadí. Menším problémem je, když to jsou počáteční, nebo koncové body. Problém ale snadno vyřešíme tím, když body seřadíme lexikograficky, tedy nejdříve podle x a pokud je stejné, pak podle y. To nám jednoznačně určí pořadí bodů a počáteční i koncový bod.

Také si to můžeme představit tak, že rovinu nepatrně natočíme. Tím se určitě konvexní obal (až na natočení) nezmění, nikde nebudou dva body nad sebou a z pohledu algoritmu je to vlastně totéž, jako bychom prošli body v lexikografickém pořadí.

Hledání průsečíků úseček

Nakonec si ukážeme ještě jeden typický zametací problém, který principu zametání využívá o trochu více než konvexní obal. Představte si, že máte v rovině N úseček a chcete najít všechny jejich průsečíky.

Hledáme samozřejmě co nejrychlejší algoritmus vzhledem k N a počtu průsečíků P.

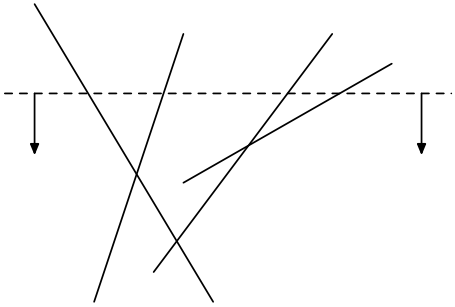
Bystří si jistě již spočítali, že průsečíků může být v extrémním případě až N^2 a tedy nic rychlejšího než zkontrolovat každou úsečku se všemi dalšími v tomto případě není.

Ale takové případy se moc často nestávají, spíše naopak. Uvažujme tedy, že průsečíků je řádově tolik, kolik je úseček a v tom případě je výše popsaný algoritmus již pomalý.

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

Předpokládejme pro zjednodušení, že v žádném bodě se neprotínají tři a více úseček, žádné dvě úsečky nemají více než jeden společný bod (neleží přes sebe) a žádná úsečka není ani přesně svislá, ani přesně vodorovná. Vyřešení takovýchto případů spočívá v jednoduchých úpravách uvedeného řešení.

Použijeme opět zametací přímku (pro lepší představu teď jdoucí shora dolů, obecně ale nemá směr zametání význam), kterou budeme skákat přes události, a na ní si budeme udržovat aktuální stav. Nazvěme ji třeba *průřezem.* Jak už název napovídá, bude udržovat pořadí úseček, které aktuálně protínají zametací přímku. Jelikož se průřez bude po každé události měnit, budeme pro něj potřebovat šikovnou datovou strukturu. Ale na to se podíváme až potom, co si rozebereme události, ať víme, co od průřezu budeme chtít.



Stejně jako v minulém případě budou mezi událostmi všechny body na vstupu (tedy počáteční i koncové body úseček), vyskytnou se tam ale i další. Pojďme si tedy trochu lépe rozebrat události a akce, které se při nich mají stát:

- *Začátek úsečky:* Přidáme úsečku na správné místo do průřezu, spočítáme případné průsečíky s okolními úsečkami a přidáme je do seznamu událostí.
- *Konec úsečky:* Smažeme úsečku z průřezu, a jelikož se nám dvě okolní úsečky dostanou smazáním této k sobě, musíme ještě spočítat jejich případný průsečík a přidat ho do seznamu událostí.
- *Průsečík:* Započítáme a zapíšeme si průsečík úseček, prohodíme pořadí těchto dvou úseček na průřezu, a jelikož se nám k sobě na průřezu dostaly nové úsečky, musíme spočítat, jestli se někde protínají, a případně průsečíky přidat do seznamu událostí.

Spočítání průsečíků úseček je jednoduchá analytická geometrie. Nejdříve porovnáme jejich směrnice. Pokud jdou od sebe, nemusíme se o nic starat, pokud jdou k sobě, spočítáme, ve kterém bodě se protnou. A když máme tento bod, jenom ověříme, jestli leží na obou úsečkách (neboli že úsečky nekončí ještě před spočítaným průsečíkem).

Když se podíváme na požadavky, hodilo by se nám umět v průřezu rychle vyhledávat, přidávat a mazat, k čemuž nám nejlépe poslouží vyhledávací strom. Ale co za informace si budeme o úsečkách ve vrcholech stromu pamatovat? Jejich aktuální x-ovou pozici (tedy přesněji x-ovou souřadnici bodu této úsečky na úrovni zametací přímky)? Tu bychom museli po každé události u všech úseček přepočítat, budeme na to tedy muset jít chytřeji.

Ve vrcholech stromu si budeme ukládat pouze nějaký rovnicový tvar úsečky (například její obecnou rovnici, nebo směrnici a bod) a vždy, když budeme vyhledávat ve stromu, tak si na základě aktuální y-ové pozice zametací přímky spočítáme v konstantním čase aktuální x-ovou pozici úsečky (jednoduchým doplněním do obecné rovnice) a podle toho se budeme ve vyhledávacím stromu pohybovat.

Máme tedy datovou strukturu pro průřez, ale jak dlouho budou trvat operace s ním? Jelikož v každou chvíli bude ve vyhledávacím stromu maximálně N vrcholů (tedy maximálně tolik, kolik je úseček), budou všechny operace se stromem trvat $\mathcal{O}(\log N)$.

Do seznamu událostí budeme potřebovat také přidávat prvky, takže tentokrát se nám mnohem více hodí použití nějaké haldy. Opět si můžeme uvědomit, že v haldě bude najednou pouze $\mathcal{O}(N)$ prvků (za každou úsečku její začátek a konec a průsečíky úseček vedle sebe na průřezu, tedy maximálně $N-1$ průsečíků) a tedy operace s haldou bude trvat také $\mathcal{O}(\log N)$.

Když už máme vybudované datové struktury, podívejme se na to, jak algoritmus poběží. Na začátku přidáme do průřezu první úsečku a do seznamu událostí všechny začátky i konce úseček. Pak již jen postupujeme po událostech, každou událost zpracujeme podle postupu výše a skončíme ve chvíli, kdy nám dojdou všechny události.

Algoritmus funguje správně, jelikož postupně projde přes všechny průsečíky (když jedna úsečka protíná více dalších, tak postupným prohazováním v průřezu se dostanou všechny tyto dvojice vedle sebe a všechny průsečíky přidáme do událostí) a žádný průsečík neprojdeme vícekrát.

Zpracování jakékoliv události nás stojí konstantní množství operací s datovými strukturami, a protože každá z těchto operací stojí maximálně $\mathcal{O}(\log N)$, tak nás zpracování jedné události stojí $\mathcal{O}(\log N)$. Počet událostí je $2N + P$ kde N je počet úseček a P počet průsečíků na výstupu, tedy celková časová složitost je $\mathcal{O}((N + P) \log N)$. Pro pořádek ještě uveďme paměťovou složitost, které je díky použitým datovým strukturám $\mathcal{O}(N)$.

Můžeme si všimnout, že pokud by průsečíků bylo řádově N^2, tak jsme si vlastně pohoršili. Předpokládali jsme ale situaci, kdy je průsečíků řádově stejně jako úseček. V tomto případě je náš algoritmus výrazně rychlejší.

Závěr

Prošli jsme si základní geometrické algoritmy pro rovinnné problémy a ukázali jejich základní myšlenky. Různou aplikací a kombinací těchto postupů můžeme řešit většinu lehčích geometrických problémů v rovině, se kterými se setkáme.

Jen jako ochutnávku si ještě uvedeme například *Voroného diagramy,* což je rozklad roviny na oblasti, které jsou vždy nejblíž danému bodu (motivací může být například přiřazení obcí na mapě k nejbližšímu krajskému městu). Při jejich konstrukci se také uplatní zametání roviny, ale tentokrát již ne přímkou, ale pomocí zametacích parabol.

A jak jsme si uvedli na začátku, mnohé z uvedených postupů lze zobecnit z roviny i do prostoru a podobně. Ale o tom třeba někdy jindy. Pokud však máte zájem o další informace o geometrických algoritmech, tak vás mohu odkázat na studijní text o geometrických algoritmech[30] k přednášce ADS na stránkách Martina Mareše.

Pokud stále nemáte geometrie dost, můžete si ještě zkusit vyhledat pojmy *kombinatorická a výpočetní geometrie.* Dostanete se tak ke spoustě dalších zajímavých materiálů.

Dnešní kuchařkové menu vám servíroval

Jirka Setnička

[30] `http://mj.ucw.cz/vyuka/1112/ads2/6-geom.pdf`

Vzorová řešení

24-1-1 Podvádíme s XORem

Nejdříve je potřeba rozmyslet podmínky, za kterých lze součástky rozdělit na hromádky se stejným XORem.

Operace XOR je komutativní. Můžeme tedy nejdříve spočítat XOR přes všechny součástky první hromádky, poté XOR přes součástky té druhé. Označme tyto výsledky x a y. Z definice operace XOR snadno nahlédneme, že $x \oplus y$ je v případě $x = y$ rovno nule. Naopak, pro $x \neq y$ je $x \oplus y$ vždy různé od nuly.

Navíc kvůli komutativitě a asociativitě též platí, že pro dané součástky bude XOR mezi libovolnými dvěma hromádkami vždy stejný – hromádky oddělíme závorkami a na pořadí operandů v rámci nich nezáleží. Díky tomu dostáváme, že nulový XOR všech součástek je postačující a zároveň nutnou podmínkou pro existenci řešení.

V případě, kdy je XOR všech prvků nulový, musíme pro co největší rozdíl hodnot hromádek dát kolegovi buď nic, nebo nejlevnější prvek. V zadání nebylo řečeno, zda kolegova hromádka musí býti neprázdná, tudíž jsme i taková řešení považovali za správná. Pro nenulový XOR prvků pak švindl na kolegovi provést nelze.

Jan Bok

24-1-2 Rozházené řádky v BASICu

Tato úloha se ukázala jako lehká, většina z Vás došla ke správnému řešení. Častou chybou byla buď absence důkazu, nebo důkaz pouze pro konkrétní případ.

Pro jednoduchost si budeme zpřeházené řádky představovat pouze jako posloupnost čísel řádků, jak jdou na vstupu po sobě. Představují vlastně permutaci. Nyní si můžeme všimnout několika věcí:

Celou permutaci můžeme rozložit na několik samostatných cyklů. Jako cyklus označíme takovou vybranou podposloupnost prvků, ve které stačí prohodit prvky pouze v rámci této podposloupnosti, abychom dostali prvky na správné pozice (na ty, na které patří), a která se zároveň nedá rozložit na žádné menší cykly.

Speciálním případem cyklu je cyklus o jednom prvku, který představuje prvek již správně umístěný na svém místě. Dokažme si, že v jakémkoliv větším cyklu velikosti k nám stačí právě $k - 1$ výměn prvků k tomu, abychom všechny prvky dostali na správné místo.

U cyklu se dvěma prvky platí předpoklad triviálně, zde nám stačí právě jedna výměna k tomu, abychom na správné místo dostali oba dva prvky. U větších cyklů můžeme lehce nahlédnout, že každou výměnou umístíme na správné místo právě

jeden prvek. Nakonec se dostaneme do situace, kdy dojde k prohození posledních dvou prvků, při které umístíme správně oba prvky.

Protože jste ale v odevzdaných řešeních měli problém hlavně se správným důkazem, ukážeme si ještě formálně lepší důkaz pomocí indukce. Cykly s jedním a dvěma vrcholy jsme si rozebrali již výše, takže rovnou přistoupíme k indukčnímu kroku a budeme předpokládat, že pro k prvků potřebujeme právě $k-1$ výměn.

Nyní si vezmeme cyklus s $k+1$ prvky. Pokud prvek A vyměníme s prvkem, který se aktuálně nachází na správné pozici prvku A, rozdělíme náš cyklus na dva. Jeden jednoprvkový cyklus je samostatný prvek A, druhý cyklus s k prvky tvoří všechny zbylé prvky z původního cyklu.

O jednoprvkový cyklus se již zajímat nemusíme a z indukčního předpokladu víme, že na druhý cyklus o k prvcích potřebujeme právě $k-1$ výměn. Tedy na původní cyklus s $k+1$ prvky jsme potřebovali $(k-1)+1$ výměn. Tím jsme dokázali naše tvrzení.

Jak tedy spočítat, kolik výměn potřebujeme k navrácení všech prvků na správná místa? Jednou z možností je projít všechny cykly v posloupnosti na vstupu a v každém spočítat počet nutných prohození (tedy počet prvků v cyklu zmenšený o jedna).

Druhou možností je uvědomit si, že za každý cyklus nám stačí započítat pouze onu „-1“, neboli stačí nám spočítat počet cyklů a odečíst ho od celkového počtu prvků (tedy pro posloupnost délky N rozdělenou do C cyklů bude správná odpověď $N-C$).

Implementačně i časově jsou oba postupy stejně náročné. Přesněji pro variantu počítající počet cyklů je paměťová složitost $\mathcal{O}(N)$, protože si na vstupu musíme načíst informace o každém prvku a pamatovat si, které prvky jsme již v cyklu prošli.

A časová složitost je také $\mathcal{O}(N)$, jelikož na každý prvek sáhneme právě dvakrát – jednou při lineárním procházení, jednou při procházení každého cyklu.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-1-2.c`

Jiří Setnička

24-1-3 Turnaj jazyků

Zadání této úlohy by se na první přečtení lekl asi každý; snad proto mi přišla jen hrstka řešení od pár odvážlivců. Pojďme se tedy podívat, jak by zadání vypadalo napsané stručněji a s menší porcí pohádky.

Mějme soutěž o K kolech s N soutěžícími ($N, K \leq 1000$). V každém kole může být vyřazen libovolný (i nulový) počet soutěžících. Po posledním kole ve

hře musí zůstat právě jeden, BestLang, jehož bodový zisk za celou soutěž máme maximalizovat.

Body v jednom kole se počítají celočíselně podle vzorce $Vyhra(v, h) = v \cdot 100\,000/h$, kde h je počet hráčů na začátku kola, ze kterých je v vyřazeno. Zisk soutěžícího za celou hru je součet získaných bodů ze všech kol.

Výstup programu má být posloupnost, která v k-tém prvku obsahuje počet vyřazených v k-tém kole. Při tom uvažujeme průběh hry, během které BestLang dosáhne maximálního počtu bodů.

V zadání stále máme některé trochu chlupaté části. Na první pohled působí divně, že by mohlo mít smysl nikoho nevyřazovat. Aby situace byla jasnější, uvedeme si několik jednoduchých pozorování:

- Zisk bodů nezávisí na čísle kola. Jde jen o počet jazyků na začátku kola a počet vyřazených.
- V soutěži proběhne nejvýše $N - 1$ vyřazení. Může se stát, že v některém z kol nikdo vyřazen být nemůže – například pro $K > N - 1$.
- Nezáleží na umístění kol bez vyřazení, protože tato nijak nemění stav hry (body, počet zbývajících soutěžících). Bez újmy na obecnosti je můžeme umístit třeba na konec.

Příprava je za námi, o problému už trochu něco víme, pusťme se do něj tedy pořádně. První je po ruce procházení všech možných průběhů her, se svojí složitostí $\mathcal{O}(N^K)$ je však beznadějně pomalé. Kdo už někdy potkal podobnou úlohu, bude se zamýšlet nad použitím dynamického programování. I mně se hodilo při psaní vzorového řešení, hodlám se k němu však dostat malou oklikou.

Nebudeme totiž ze začátku vůbec počítat vyřazené soutěžící, ale bodový zisk. Sestrojíme rekurzivní funkci `Zisk`, která pro zadaný počet kol a hráčů spočítá, jaký nejvyšší počet bodů může BestLang získat.

```
int Zisk(int k, int h){
  if (h == 1)
    // poslední soutěžící už nemá koho vyřadit
    return 0;
  if (k == 1)
    // v posledním kole končí i zbylí soupeři
    return Vyhra(h - 1, h);
  int max = 0;
  // v: počet vyřazených (aspoň jeden)
  for (int v = 1; v <= h - 1; v++) {
    int vyhra
          = Vyhra(v, h) + Zisk(k - 1, h - v);
    if (max < vyhra)
```

```
            max = vyhra;
        }
        return max;
    }
```

Na této funkci je snadno vidět, že skončí a vrátí správný výsledek. Také se objeví jedna důležitá pravidelnost uvnitř úlohy: maximální zisk z posledních k kol je možné spočítat s pomocí maximálního zisku z posledních $k-1$ kol. Nejvýrazněji ovšem stále bije do očí exponenciální časová složitost.

Co naplat, pro zrychlení budeme muset obětovat kousek paměti. Všimneme si, že se rekurzivně ptáme častokrát na stejnou věc – například pokud BestLang nejprve vyřadí jednoho soupeře a potom dva, další rekurzivní volání jsou stejná, jako kdyby nejprve vyřadil dva a potom jednoho.

Dvěma parametry funkce budeme indexovat dvourozměrné pole s tabulkou již spočítaných hodnot zisku. Funkce `Zisk` se při každém volání nejprve podívá, jestli si výsledek nepamatuje. Pokud ano, místo nového počítání vrátí známou hodnotu z tabulky, jinak ji spočítá a před vrácením zapíše.

Nakonec dáme dohromady všechen vtip a postřehy, jež jsem dosud utrousil, opustíme rekurzi a půjdeme na řešení dynamicky. Dosavadní pomocná tabulka se stává tím hlavním, o co nám jde. Od `Zisk(K,N)` k `Zisk[K,N]` tak daleko není, význam sloupců a řádků je tedy zřejmý.

Ke spočítání tabulky vlastně jen použijeme to, co už jsme uměli při rekurzi. Jediný myšlenkový rozdíl je, že musíme postupovat pozpátku, od konce hry, po jednotlivých kolech (řádcích).

Poslední kolo (první řádek) má ve všech svých buňkách výhru pro vyřazení všech soupeřů. Při výpočtu každé buňky předchozího řádku se stejně jako v rekurentní verzi hledá maximum ze součtu budoucího zisku a aktuální výhry. Když výpočet dojde až k `Zisk[K,N]`, máme hledaný výsledek.

Opravdu? Ne tak docela, původní úloha se ptala po posloupnosti počtů vyřazených, o maximálním bodovém zisku vůbec nemluvila. Ale tato posloupnost je jenom popisem, jak tolik bodů získat. Jde snadno zrekonstruovat, pokud si každá buňka tabulky pamatuje počet vyřazených soupeřů, při kterém bylo dosaženo maxima bodů. K tomu bude potřeba druhá, stejně velká tabulka, což paměťovou složitost nezhorší.

Paměti celkem potřebujeme $\mathcal{O}(N \cdot K)$, času $\mathcal{O}(N^2 \cdot K)$, protože na každé buňce tabulky trávíme čas $\mathcal{O}(N)$ výpočtem maxima.

Prostor pro zlepšení je dle mého názoru na úrovni konstant, ne typu složitosti. Dokázat to bohužel neumím. Problém nevypadá na první pohled tak složitě, ale celočíselné dělení se každému chytřejšímu přístupu staví do cesty.

Na to narazili i někteří řešitelé. Překvapil mě program, který vypadal, že by mohl fungovat, běží v čase $\mathcal{O}(N \cdot K)$ a prostoru $\mathcal{O}(K)$. Také dynamické programování, ale tentokrát přes počet soutěžících, ne přes počet kol, jak popisuji výše. Většinou dával správný výsledek, ale pro $N, K \leq 100$ se zhruba 500krát seknul. Rozhodnutí, ve kterém kole vyřadit dalšího soutěžícího, bylo uděláno docela správně, ale to bohužel nestačí, protože někdy je potřeba některému kolu počet vyřazených zmenšit.

Při samotné implementaci je vhodné zamyslet se nad datovými typy. Aby byla překročena v prvcích pole velikost 32bitového integeru, muselo by každé z maximálně tisíce kol přispět víc než milionem bodů; ze vzorce však jasně vyplývá, že největší možná výhra za jedno kolo se pouze blíží statisíci.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-1-3.c`

Tomáš „Palec“ Maleček

24-1-4 Složitá složitost

Na úvod poznamenám, že ste si niektorí správne všimli, že ak premennú `odm` inicializujeme na hodnotu $\lfloor\sqrt{n}\rfloor$, tak potom pre niektoré hodnoty n pole `zač` nebude veľkosťou stačiť.

Algoritmus najprv rozdelí pole dĺžky n na $\sqrt{n}$ vzostupne zoradených úsekov $\sqrt{n}$ dlhých. Kým zoradíme jeden úsek (algoritmus využíva Bubblesort), strávime v najhoršom prípade $\mathcal{O}(n)$ času, a to práve vtedy, keď je úsek zoradený zostupne. Zoradenie každého úseku nám preto bude trvať $n\sqrt{n}$ v najhoršom prípade.

Potom algoritmus označí minimá v jednotlivých úsekoch, ktorých je rovnako ako úsekov, tedy $\sqrt{n}$. Ďalej nasleduje n prechodov, kde v každom prechode bude vybraté jedno minimum (ktorých je $\sqrt{n}$) do výsledného poľa.

Za nové minimum označí algoritmus prvok, ktorý je v rámci úseku bezprostredne za aktuálne vybratým minimom. Vytváranie výsledného poľa má teda časovú zložitosť $\mathcal{O}(n\sqrt{n})$.

Z predchádzajících odstavcov plynie, že výsledná časová zložitosť je $\mathcal{O}(n\sqrt{n} + n\sqrt{n}) = \mathcal{O}(n\sqrt{n})$.

Konečne pár slov k pamäťovej zložitosti. Potrebujeme si pamätať n prvkov postupnosti a pri vytváraní výsledného poľa $\sqrt{n}$ pozíc miním, teda pamäťová zložitosť je $\mathcal{O}(n)$.

Peter Zeman

24-1-5 Razítková grafika

Dříve, než začneme hledat největší možné razítko, jakým lze obrázek vyrazítkovat, podíváme se, jak zjistit, zda obrázek lze vyrazítkovat razítkem velikosti S.

Všimneme si, že bod, který je umístěn nejvíce nahoře a nejvíce vlevo, můžeme vyrazítkovat jen tak, že v něm bude mít razítko levý horní roh. Pokud tedy existuje čtverec velký $S \times S$, který má levý horní roh právě v tomto políčku, tak razítko můžeme použít. V opačném případě víme, že obrázek vyrazítkovat nejde.

Na razítkování razítkem velkým S tedy použijeme následující algoritmus. Obrázek budeme procházet po řádcích a vždy, když najdeme černé políčko, tak se podíváme, jestli existuje čtverec velký $S \times S$ mající levý horní roh v tomto políčku. Pokud ano, tak tento čtverec smažeme, a pokud ne, tak obrázek nelze obarvit.

Pokud tímto způsobem projdeme celý obrázek, tak jsme jej právě vyrazítkovali. Každé políčko maximálně jednou přebarvujeme a maximálně jednou přes něj projdeme. Tento algoritmus tedy běží v čase $\mathcal{O}(W \cdot H)$, kde W je šířka a H výška obrázku.

Nyní, když umíme razítkovat, nám stačí najít největší velikost razítka, se kterým obrázek dokážeme vyrazítkovat. Takový přímočarý postup začneme s razítkem o velikosti $\mathcal{O}(\min(W, H))$ a budeme jej postupně zmenšovat, dokud se nám obrázek nepovede vyrazítkovat.

Tento postup má časovou složitost $\mathcal{O}(\min(W, H) \cdot W \cdot H)$, protože například pro černý obrázek s bílým pravým dolním rohem s každým razítkem projdeme skoro celý obrázek.

Další věc, které si můžeme všimnout, je, že pokud obrázek lze vyrazítkovat razítkem velkým S, tak S musí dělit délky všech vodorovných i svislých souvislých úseků (myšleno v rámci jednoho řádku či sloupce).

Tedy velikost razítka musí dělit největšího společného dělitele délek těchto úseků. Stačí nám tedy zkoušet jen velikosti razítek, které dělí největšího společného dělitele. Zdrojový kód tohoto algoritmu je přiložen k vzorovému řešení.

Určitě nevyzkoušíme více než $2 \cdot \sqrt{\min(W, H)}$ razítek, protože žádné číslo k nemá více než $2 \cdot \sqrt{k}$ dělitelů. Snadno můžeme nahlédnout, že pokud $k = a \cdot b$, tak potom $a \leq \sqrt{k}$ nebo $b \leq \sqrt{k}$.

Na počítání největšího společného dělitele použijeme Euklidův algoritmus, který pracuje v logaritmickém čase. Součet čísel, pro které jej zavoláme, je maximálně $W \cdot H \Rightarrow$ celkem Euklidovým algoritmem ztratíme nejvýše čas $\mathcal{O}(W \cdot H)$.

Časovou složitost nám nejvíce ovlivňuje samotné razítkování, celkem tedy dostáváme $\mathcal{O}(W \cdot H \cdot \sqrt{\min(W, H)})$.

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-1-5.cpp`

Karel Tesař

24-1-6 V bludišti s krumpáčem

Jak jste téměř všichni uhádli, mřížka, ve které se pohybujeme, je jen speciálním případem grafu. Je tedy nasnadě pokusit se aplikovat některé grafové algoritmy, které známe z KSP kuchařek či odjinud.

Na náš problém s bludištěm by se hodil jeden ze dvou algoritmů – prohledávání do šířky nebo Dijkstrův algoritmus. Prohledávání do šířky (BFS) má tu výhodu, že nalezne nejkratší cestu ze začátku do cíle v lineárním čase ($\mathcal{O}(N \cdot M)$).

Ve své základní podobě však neumí pracovat se skutečností, že některé cesty, ač stejně dlouhé na počet políček, jsou různě dlouhé co do vzdáleností. Jinak řečeno, nepracuje s ohodnocenými hranami (či v našem případě vrcholy).

Druhý algoritmus, Dijkstrův, vyhledá nejkratší cestu v grafu ohodnoceném nezápornými reálnými čísly. Používá k tomu datovou strukturu *halda*, proto jej také máme popsaný v kuchařce o haldách. Bohužel, jeho časová složitost je vyšší, zde by byla alespoň $\mathcal{O}(N \cdot M \cdot \log(N \cdot M))$.

Snadné řešení tedy bylo napsat „Dijkstra“ a dostat pár bodů. Na plný počet nezbývá, než se zamyslet nad tím, jestli nejde prohledávání do šířky upravit, aby pomohlo i nám, případně jestli nejde Dijkstra zrychlit.

Jednodušší bude upravit prohledávání do šířky. To je v naší kuchařce implementováno pomocí fronty (pokud nevíte, jak fronta funguje, utíkejte si to přečíst).

Když procházíme jedno políčko, obvykle chceme všechny jeho sousedy přidat dozadu do fronty. To v našem bludišti neplatí, protože chceme souseda přidat buď dozadu, nebo „o K míst dál,“ tedy nejen za všechny ve frontě, ale navíc ještě za všechny sousedy, kteří jsou blíž.

Využijeme toho, že máme jen dva typy políček a můžeme si udělat dvě fronty. Jednu pro *rychlá* políčka, tedy ta, kterými procházíme za jeden krok, a jednu pro *pomalá* políčka, tedy pro zdi. Políčka budeme dávat do front podle toho, kterého typu jsou.

Musíme ještě zajistit, abychom nezapomněli vybírat z fronty pro pomalá políčka, když už je čas (tedy poté, co jsou všechny kratší cesty vybrány). Stačí si ke každému políčku připsat, v kterém čase jej máme z fronty vyzvednout. Například pro $K = 5$ a pomalé políčko, které je sousedem políčka s hodnotou 15, připíšeme při uložení do fronty 20. Pro rychlá políčka vždy jen zvýšíme hodnotu o jedna.

Když pak vybíráme z front, jen porovnáváme, jestli má nižší hodnotu rychlé políčko, nebo pomalé políčko, a podle toho volíme nové políčko na prohledání. V obou frontách budou políčka setříděná podle poznamenané hodnoty (stejně tak, jako by byla setříděna v BFS s jednou frontou). Náš algoritmus se tedy nesplete.

Asymptotická časová složitost je stejná jako pro BFS, neboť přidáním nové fronty nám vzniklo jen konstantní zpomalení – při vybírání dalšího políčka pro průchod jen porovnáváme, ze které fronty ho máme vzít, a jinak se chováme stejně, jako kuchařkové BFS.

Martin Böhm

24-1-7 Distribuované výpočty

Základní myšlenka rešení je jednoduchá – odpojit všechny počítače, které jsou napojeny na aktuální, dřív než sebe.

Budeme procházet graf do hloubky, dokud nenarazíme na vrchol s hranami vedoucími jen k počítačům již odpojeným či navštíveným během rekurze. Ten následně odpojíme (vše, co je k němu připojeno, bude po odpojení stále v síti) a podobně postupujeme dál.

Časová složitost je lineární vzhledem k hranám i vrcholům, protože každý počítač navštívíme právě jednou a na každou hranu se podíváme maximálně dvakrát (z každého konce). Paměťová taktéž.

Program (Pascal):
`http://ksp.mff.cuni.cz/viz/24-1-7.pas`

Pavel Čížek

24-1-8 Pojďte pane, budeme si hrát

Z úkolu v matematické části určitě nebude nikdo smutný, protože byl vcelku lehký, což se projevilo i na veselém bodovém zisku – až na detaily byla řešení správně.

Pro hru s odebíráním žetonů v případě maximálně tří hromádek bylo potřeba ověřit, že smutné stavy jsou právě ty, v nichž je XOR všech velikostí hromádek nulový. Smutný stav si lze představit jako stav předem prohraný – pokud jste ve smutném stavu, soupeř vás může porazit. Je-li pozice mimo smutný stav, hráč na tahu má výherní strategii.

V zadání se tvrdí, že strategie funguje pro libovolný konečný počet hromádek. Abyste nám věřili, vrhněme se na obecnější důkaz!

Bude se nám hodit asociativita a komutativita XORu, tedy že můžeme velikosti hromádek XORovat v libovolném pořadí. Ověření těchto vlastností je mechanickou záležitostí spočívající v rozboru případů. Také je dobré uvědomit si, že i-tý bit v XORu velikostí hromádek může být roven jedné právě tehdy, když má lichý počet hromádek i-tý bit jedničkový.

Začněme nejlehčím požadavkem: prohraný stav se všemi hromádkami prázdnými je smutný. Velikosti hromádek jsou shodně nuly, jejich XOR je nula, tedy stav je smutný.

Proč všechny tahy ze smutných pozic vedou do pozic, které smutné nejsou? To už tak zřejmé není. Mějme tah ze smutné pozice, který odebere k žetonů z hromádky H. XOR všech hromádek byl dosud nula, tedy XOR hromádek mimo H je přesně velikost H.

Po odebrání z H se XOR hromádek mimo H nezměnil, ale H ano. Tedy XORujeme dvě různá čísla, což nikdy nedá nulu, jelikož jejich binární zápis se musí lišit alespoň na jednom místě.

Zbývá poslední požadavek: není-li hráč ve smutném stavu, má tah vedoucí do smutného stavu (takže je vlastně ve „veselém“ stavu, protože má jistou výhru). Dalo by se říci, že z celé úlohy jde o nejzajímavější část, přičemž Vaše řešení se někdy lišila.

XOR velikostí hromádek je nenulový (označme ho X), my chceme po odebrání z jedné hromádky mít smutný stav. Označme i pozici nejlevějšího jedničkového bitu v binárním zápise X. Jedna z hromádek (označme ji H) musí mít velikost alespoň 2^{i-1} a i-tý bit roven jedné – máme lichý počet hromádek, jež mají v binárním zápise na pozici i jedničku.

Z hromádky H odeberu žetony v počtu menším nebo rovném 2^{i-1} tak, aby se vynuloval i-tý bit její velikosti a výsledný XOR všech hromádek byl nulový. Jak se přijde na to, kolik mám odečíst? Jednodušší je přemýšlet, jak velká má být hromádka H, aby výsledný XOR byl nula.

Řešení není těžké: vezmeme velikost hromádky H a překlopíme bit na místech, kde je v X jednička (tedy H vyXORujeme s X). Tím se v XORu všech hromádek změní parita počtu jedniček pouze na bitech, kde byl původně lichý počet jedniček, což dává nulový XOR všech hromádek. Číslo H se navíc muselo zmenšit, jelikož nejlevější změněný bit se překlopil z jedné na nulu.

Q. E. D. (Quite Easily Done nebo Quod Erat Demonstrandum, vyberte si.) Jak je vidět, nebylo potřeba nikde použít počet hromádek, i když jsme předpokládali, že jsou alespoň dvě.

Na závěr řešení tohoto úkolu dodejme, že popsaná hra se jmenuje Nim. Lze ji hrát i s modifikací, kde prohrává ten, kdo vezme poslední žeton z poslední hromádky. Definice smutné (předem prohrané) pozice se pak liší jen v určitých aspektech – můžete si jako cvičení rozmyslet v jakých.

Druhé části seriálové úlohy se zhostili jen nemnozí, ačkoliv šlo o kreativnější úkol. Bylo třeba v Pythonu napsat pro šestvorky ohodnocovací funkci a funkci generující tahy z dané pozice.

Pokud se zdá, že funkce generující tahy měla být jen otrocká práce spočívající ve vygenerování všech dvojic volných políček, není tomu tak. Předně je těch dvojic opravdu hodně (po prvním tahu $\binom{224}{2}$, tedy 24 976) a většina z tahů postrádá smysl, protože jsou třeba na kraji desky, kde se nikde v okolí nehraje.

Dobrou heuristikou mohlo být hledání linie svých značek, kterou je možno v jednom tahu vyhrát (tj. například 4 značky v řadě s oběma volnými konci). Pokud neexistuje, tak hledání soupeřovy linie, s níž by mohl vyhrát dalším tahem, a jinak generování všech dvojic z políček sousedících s nějakou značkou.

Ještě zajímavější je ohodnocování pozice. Určitě je dobré při něm zkoumat, jestli už není pozice vyhraná nebo prohraná. Jinak se hodí třeba hledat souvislé linie jednoho hráče, které mají alespoň na jednom konci volné políčko, a ty ohodnocovat podle délky (např. exponenciálně), přičemž je dobré zohlednit, jestli má linie oba konce volné, nebo jen jeden.

Ohodnocení je pak součet ohodnocení mých linií minus součet soupeřových linií. Vše pak záleží na dobrém nastavení konstant. Je to však jen jeden z mnoha možných přístupů a určitě půjde vymyslet lepší :-)

Pavel „Paulie“ Veselý

24-2-1 Požární poplach

Uvedieme tri postupne zlepšujúce sa riešenia. Prakticky všetky riešenia, ktoré prišli a boli správne, popisovali jeden z nižšie uvedených algoritmov.

Aby sme nezapísali viac, ako je treba, je dobré si všimnúť čo majú všetky algoritmy riešiace túto úlohu spoločné. A síce je nutné zistiť, kedy jednotlivé stromy (políčka 1×1 označené bodkou) začnú horieť. Keby sme túto informáciu nemali, tak o ľubovoľnej ceste lesom zľava doprava nie sme schopní rozhodnúť, ako dlho bude priechodná. Potrebné zistíme prehľadávaním do šírky od políčok, ktoré sú označené ako ohne. Časová zložitosť je lineárna k veľkosti lesa, teda $\mathcal{O}(R \cdot S)$.

Drevorubačský algoritmus

Myšlienka je pokúsiť sa po označení nejakého políčka (viď predchádzajúce odstavce) nájsť cestu lesom zľava doprava (napr. opäť prehľadávaním do šírky).

Predstavme si, že sme nejaké políčko označili číslom i a nevieme nájsť cestu (po neoznačených políčkach – tie ešte nehoria) zľava doprava. To teda znamená, že les je priechodný najneskôr v čase $i - 1$.

Algoritmus funguje, avšak má nepeknú časovú zložitosť. Označených políčok je $R \cdot S$ a pre každé takéto políčko raz prehľadáme do šírky celý les. Teda celková časová zložitosť tohoto algoritmu je $\mathcal{O}(R \cdot S \cdot R \cdot S) = \mathcal{O}(R^2 \cdot S^2)$, teda kvadratická k veľkosti lesa.

Zlepšujeme binárnym vyhľadávaním

Predpokladajme, že les dohorel v čase n (políčka máme označené číslami $0 \ldots n$).

Jednoduchým pozorovaním je, že ak je les v čase k priechodný, tak je určite priechodný aj v čase menšom ako k. Podobne ak je les už v čase k nepriechodný, tak neskôr priechodný určite nebude. Keď teda zvolíme nejaké k a skúsime nájsť cestu po políčkach označených číslom väčším ako k, tak podľa výsledku (buď sme našli cestu alebo nenašli) nás nemusia viac zaujímať políčka označené číslom väčším (ak sme cestu nenašli) alebo menším (ak sme cestu našli).

Z tohoto pozorovania nás môže napadnúť použiť binárne vyhľadávanie. Nech $d = 0$ a $h = n + 1$. Opakujeme nasledovné:

- pozrieme sa či vieme prejsť lesom v čase $\lfloor (d + h)/2 \rfloor$
- ak vieme, tak položíme $d = (d + h)/2$
- inak $h = (d + h)/2$
- ak $h - d = 1$, tak skončíme, výsledok je d

Nahliadnime ešte, že d je hľadané číslo. Predtým, než sme skončili, sme buď znížili h na $d+1$, alebo zvýšili d na $h-1$. V prvom prípade to znamená, že v čase

$d+1$ sa prejsť nepodarilo, ale zároveň vieme, že v čase d a menšom prejsť vieme, teda správna odpoveď je d. Druhý prípad si analogicky rozmyslíte sami.

Zostáva už len rozobrať časovú zložitosť. Pre binárne vyhľadávanie je potreba $\mathcal{O}(\log n)$ krokov, kde $n = \mathcal{O}(R \cdot S)$. Každý krok má časovú zložitosť $\mathcal{O}(R \cdot S)$, preto výsledná časová zložitosť druhého algoritmu je $\mathcal{O}(R \cdot S \cdot \log n)$.

A konečne lineárne riešenie

Opäť predpokladajme, že máme označené políčka číslami $0 \ldots n$. Chceme odpozorovať, či je les priechodný v čase k, ale nie len tak bez rozmyslu, pretože to by sme sa dostali časovú zložitosť $\mathcal{O}(R \cdot S \cdot n)$ a boli by sme niekde medzi dvoma vyššie uvedenými algoritmami. Budeme chcieť na každé políčko stúpiť $\mathcal{O}(1)$-krát a tým pádom dosiahnuť časovú zložitosť $\mathcal{O}(R \cdot S)$.

Môžeme to spraviť napríklad tak, že budeme postupovat v čase dozadu a prepočítavať, z ktorých políčok je dosiahnutelný cieľ. Pre čas k pridáme políčka, ktoré začali horieť v čase k. Ak z nejakého pridaného políčka existuje cesta na pravý okraj, tak z tohoto políčka prehľadáme do šírky celý les ale len po pridaných políčkach. Všetky políčka, na ktoré pri prehľadávaní stúpime uzavrieme (pri ďalšom prehľadávaní sa už na ne nedostaneme). Ak sme uzavreli aj nejaké políčko na ľavom okraji, tak môžeme skončiť.

Políčka budeme označovať `U` – uzavreté, `O` – otvorené a `X` budú ostatné. Pre $i = n, \ldots, 1$ budeme opakovať:

- všetky políčka označené číslom i označ ako `O`
- ak existuje políčko P, ktoré je označené `O` a susedí s políčkom označeným `U` alebo je napravo, tak P označ `U` a spusti z P prehľadávanie do šírky po políčkach označených `O` a všetky označ `U`
- skonči hneď, ako je nejaké políčko naľavo označené `U` – výsledok je i

Nech k je číslo, ktoré hľadáme. Potom v k-tom sú políčka s číslom k a väčším označené `U` alebo `O` a nutne musí existovať cesta po týchto políčkach zľava doprava a nájdeme ju prehľadávaním do šírky. V čase $k+1$ ešte nemohla, inak by sme ju našli už vtedy.

Každé políčko najprv označíme `O` a ak sa k nemu dostaneme znovu, tak ho označíme `U` a potom ho už viac neuvidíme. Celková časová zložitosť teda je $\mathcal{O}(R \cdot S)$.

Pamäťová zložitosť všetkých troch popísaných algoritmov je $\mathcal{O}(R \cdot S)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-2-1.c`

Martin „Medvěd“ Mareš & Peter Zeman

24-2-2 Centrální sklad

Centrální sklad byl tak jednoduchý, jak jen vypadal. Kdo se nebál řešit praktickou úlohu, tak měl k plnému počtu bodů velice blízko.

Nejprve se zamyslíme nad definicí průměrné vzdálenosti. Ta praví, že průměrná vzdálenost je součet vzdáleností ke všem výrobcům vydělená jejich počtem. Počet výrobců je stále stejný, dělení počtem výrobců tedy nemění výsledek a můžeme ho zanedbat.

Nyní tedy už počítáme jen součet vzdáleností. Představme si nyní, že bychom chtěli centrální sklad postavit na některém místě tak, že nalevo od něj by byli dva výrobci a napravo od něj tři výrobci. Vyplatí se to? Nevyplatí. Představme si, že ho posuneme o malé číslo c doprava. Vzdálenost od výrobců nalevo bude o c větší, čímž součet zvětšíme o $2c$, ale zároveň ho o $3c$ snížíme díky menší vzdálenosti od výrobců napravo. Tedy jsme si polepšili.

Jak dlouho budeme moci takto zlepšovat? Dokud stále ještě na jedné straně bude víc výrobců, než na straně druhé. Pro lichý počet výrobců je tedy řešení unikátní – postavit sklad přesně na umístění prostředního výrobce (mediánu) – a pro sudý počet výrobců si můžeme vybrat libovolné místo mezi $\lfloor n/2 \rfloor$-tým výrobcem a $\lceil n/2 \rceil$-tým výrobcem.

Neměli jste tedy vymyslet nic jiného, než jak najít medián v nesetříděné posloupnosti. Optimální algoritmus pracuje v lineárním čase a je popsán v naší kuchařce Rozděl a panuj;[31] my jsme však dovolili i nalezení mediánu pomocí setřídění posloupnosti některým rychlým algoritmem, jako například QuickSortem.

Program (Python):
`http://ksp.mff.cuni.cz/viz/24-2-2.py`

Martin Böhm

24-2-3 Odčítání

Program funguje korektně v případě, kdy je menšitel menší nebo roven menšenci a první platná cifra menšence je uložena v `prvni[0]`. Jak funkce odčítá? Nejdříve si uvědomme, co se děje v první části algoritmu. Na i-tou pozici v poli `vysledek` ukládáme rozdíl hodnot prvního a druhého čísla zvětšený o devět. Nakonec přičítáme k poslednímu prvku jedničku. Skutečný rozdíl je tedy zvětšený o číslo 10^d (kde d je délka pole `vysledek`). Navíc máme tento výsledek uložený v upravené desítkové soustavě, v níž mají jednotlivé řády váhy 10^i, ale číslice mohou být libovolné nezáporné (ne jen 0–9).

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

V druhé části algoritmu pak čísla v poli `vysledek` normalizujeme do klasického desítkového zápisu a zároveň se zbavujeme přebytečného 10^d. To se děje tak, že kontrolujeme, zda je na i-té pozici v poli číslo větší než 10. Pokud ano, odečteme desítku a předáme ji jako jedničku do vyššího řádu, čímž upravíme prvky v poli `vysledek` tak, abychom měli v každém prvku pole vždy jen jednociferné číslo. Všimněme si dále, že při předání jedničky doleva u prvku pole s indexem 0 odčítáme od výsledku přebytečné 10^d.

Nyní ke složitosti. Paměťová je zjevně lineární (třikrát pole velikosti d a konstantní počet proměnných k tomu). Časová je taktéž lineární. V první fázi algoritmu provádíme d operací. V druhé části při každém kroku doleva klesne součet všech číslic, zatímco při kroku doprava se nezmění. Proto je celkový počet kroků doleva nejvýš lineární; kroků doprava pak je nejvýše o d víc než kroků doleva, takže také lineárně.

Jan Bok

24-2-4 Odbočení vlevo

Kdo už slyšel o teorii grafů, tak si jistě pamatuje, že síť křižovatek a cest je nejlépe modelovaná právě pomocí grafu – a pro hledání nejkratší cesty v grafech máme lineární algoritmus procházení do šířky, také zvaný BFS.

Kdo o BFS nebo grafech[32] ještě nic neví, ten si může doplnit znalosti v našich kuchařkách.

Nejsme ale zcela hotovi – musíme totiž postavit ze zadání vhodný graf, abychom mohli spustit BFS. Kdybychom jen prohlásili křižovatky za vrcholy a ulice za hrany, narazili bychom, protože podmínka v zadání (můžeme jet jen rovně nebo doprava) nám říká, že některé ulice nesmí být průjezdné z některých křižovatek.

Můžeme tedy z neorientovaných grafů přejít na grafy orientované – takové, kde každá hrana má i směr, kterým ji lze procestovat. BFS funguje stejně dobře i v takovýchto grafech.

Nicméně ani teď ještě nejsme hotovi, protože ono docela hodně záleží na tom, ze kterého směru přijíždíme. Když přijíždíme na křižovatku z jihu, můžeme jet rovně na sever nebo doprava na východ – jenže když jedeme z východu, můžeme pokračovat rovně na západ nebo doprava na sever.

Jinými slovy, kdybychom měli vrcholy jako křižovatky, tak by se hrany musely měnit podle toho, jak do křižovatky přijedeme. Grafy se ale takto měnit nesmí. Zkusme vytvořit graf jinak.

My si uvědomíme, že když jedeme ulicí v jednom směru, už je naprosto jednoznačné, jaké máme možnosti na další křižovatce. Mohli bychom tedy vytvořit

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

graf tak, že vrcholy tohoto grafu jsou ulice z našeho zadání a orientované hrany povedou mezi ulicemi U a V, pokud z ulice U na další křižovatce jde odbočit podle pravidel do ulice V.

Na další křižovatce… ale která je další? Ulice U je ohraničena dvěma křižovatkami, ale pro každou z nich budou hrany jiné – proto budeme mít dva vrcholy pro každou ulici, podle toho, jedeme-li jedním směrem nebo tím opačným.

Jakmile máme danou ulici a směr, už je jasné, která je další křižovatka a tedy i kam dál povedou hrany.

Na tento „graf orientovaných ulic“ už můžeme pustit procházení do šířky a najít nejkratší cestu v lineárním čase. Jen ještě musíme poznamenat, že nyní umíme vyhledat jen nejkratší cestu mezi orientovanými ulicemi, ne křižovatkami – ale protože z každé křižovatky vedou jen čtyři ulice, můžeme prostě náš algoritmus zavolat pro každou možnou ulici vedoucí ze startu a pro každou možnou ulici vedoucí do cíle. Nejvýše ho tedy můžeme pustit 16krát, a jak známo, konstanta nám složitost algoritmu nijak podstatně nezhorší. (Jen konstantně.)

Na závěr ověříme, že jsme naší konstrukcí nevytvořili příliš velký graf. Na začátku máme N křižovatek, mezi nimi je nataženo nejvýše $4 \cdot N$ ulic. My jsme vytvořili méně než $8 \cdot N$ vrcholů, za každou ulici a směr jeden. Kolik náš graf má hran? No, na každé křižovatce máme dokonce už jen dvě možnosti, jak jet dál – tedy jich bude mít nejvýše $16 \cdot N$, a to je stále jen lineární zvětšení.

Převést vstupní mapu na náš graf lze také udělat v lineárním čase (pokud dostaneme vstupní data v rozumném formátu, což jste mohli předpokládat).

Sečteno a podtrženo – vstup zvětšíme jen konstanta-krát, pak na něj zavoláme lineární kuchařkový algoritmus (nejvýš 16krát) a naše časová i paměťová složitost tedy bude lineární.

Martin Böhm

24-2-5 Logická formule

Držme se zásad známého hesla rozděl a panuj. Budeme výraz postupně rozkládat na stále menší podvýrazy, dokud je nedokážeme triviálně spočítat.

Představme si, že již máme dva podvýrazy, u kterých známe počty pravdivých uzávorkování a celkové počty korektních uzávorkování. Počty pravdivých uzávorkování si označme P_a a P_b a celkové počty uzávorkování C_a a C_b. Celkový počet uzávorkování výrazu spočteme jednoduše jako $C_a \cdot C_b$, protože můžeme skombinovat jakákoliv dvě korektní uzávorkování podvýrazů.

Pokud je mezi podvýrazy operátor AND, je počet pravdivých uzávorkování celého výrazu roven $P_a \cdot P_b$ (protože celý výraz bude pravdivý v případech, kdy budou pravdivé oba jeho podvýrazy).

Pokud je mezi podvýrazy operátor OR, je to již zajímavější. Celý výraz bude pravdivý v případech, kdy je pravdivý pouze levý podvýraz, pouze pravý podvýraz nebo když jsou pravdivé oba.

Když je pravdivý levý podvýraz, může být pravý podvýraz jakkoliv korektně uzávorkovaný, tedy dostáváme $P_a \cdot C_b$ pravdivých uzávorkování. Obdobně pro případ, kdy je pravdivý pravý podvýraz.

Tím jsme ale dvakrát započítali i případy, kdy jsou pravdivé oba podvýrazy, musíme proto ještě odečíst $P_a \cdot P_b$ (podle principu inkluze a exkluze). Celý vztah pro operátor OR je tedy $P_a \cdot C_b + P_b \cdot C_a - P_a \cdot P_b$.

Teď, když už máme definované skládání podvýrazů, můžeme přistoupit k samotnému počítání. Základním přístupem je rozdělit celý výraz na podvýrazy a podle postupu popsaného výše spočítat počet pravdivých uzávorkování.

Vždy vezmeme celý výraz a postupně ho rozdělíme ve všech logických operátorech. Pro každý operátor rekurzivně spočítáme počty pravdivých a všech korektních uzávorkování příslušných podvýrazů a podle vztahů pro AND a OR určíme počty uzávorkování při rozdělení v tomto logickém operátoru.

Rekurze se zastaví v okamžiku, kdy dojde k jednoprvkovým podvýrazům (v tom okamžiku vrátí jejich hodnotu). Po spočtení hodnot ve všech operátorech výrazu jenom posčítáme tyto hodnoty a získáme počty uzávorkování celého výrazu.

Lehce si ale všimneme, že spoustu věcí počítáme v rekurzi stále dokola. Nebylo by lepší si je pamatovat? Pro tento přístup ve stylu dynamického programování si tedy založíme dvourozměrné pole, ve kterém budeme ukládat vypočtené hodnoty pro podvýrazy začínající a končící na daných prvcích.

Vždy, když budeme chtít znát počet pravdivých uzávorkování daného výrazu, tak se nejdříve podíváme do tohoto pole a teprve poté případně rekurzivně spočítáme. A naopak, vždy, když vypočteme počet pravdivých a všech korektních uzávorkování u nějakého podvýrazu, uložíme si tyto hodnoty do pole.

Tím jsme si časově hodně pomohli. Podvýrazů je N^2 s tím, že výpočet podvýrazů na jedné hladině (podvýrazů o stejné délce) nám v případě znalosti všech kratších podvýrazů trvá $\mathcal{O}(N)$. Díky tomu, že každý podvýraz počítáme pouze jednou, je celková časová složitost $\mathcal{O}(N^3)$.

Paměťová složitost je kvůli použití dvourozměrného pole $\mathcal{O}(N^2)$.

Ještě poznámka na konec: Počet korektních uzávorkování nějakého výrazu je n-té Catalanovo číslo. To je definováno jako

$$C_n = \frac{1}{n+1}\binom{2n}{n} \qquad \forall n \geq 0$$

Neudává pouze počet korektních uzávorkování, ale uplatnění najde i ve spoustě jiných úloh z kombinatoriky. Více informací lze najít například na Wikipedii.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-2-5.c`

Jiří Setnička

24-2-6 Závorky

Většina z vás si správně uvědomila, že aby byla posloupnost správně uzávorkovaná, musí být počet levých závorek větší nebo roven počtu pravých ve všech prefixech a celkově počet levých a pravých závorek musí být stejný.

Zavedeme si pro posloupnost závorek a dvě proměnné: `a.potrebujeme` bude udávat počet levých závorek, který je potřeba napsat před posloupnost, aby nikdy nebylo více pravých než levých závorek. V `a.navic` je napsáno o kolik je v a více levých závorek než pravých. Např. `"())(".potrebujeme` $= 1$,`"())()".navic` $= -1$

Posloupnost závorek a je správně uzávorkovaná, právě když `a.potrebujeme` $= 0$ a zároveň `a.navic` $= 0$. Problém se tedy redukuje na zjištění těchto dvou proměnných. Ale jak na ně přijít? Pokud bychom znali dvě poloviny posloupnosti a a b, tak vypočítat proměnné pro jejich spojení c už je rychlé.

Jaký je v c rozdíl počtů levých a pravých závorek, se spočítá jednoduše: `c.navic` $=$ `a.navic` $+$ `b.navic`.

Kolik je potřeba doplnit závorek před c (`c.potrebujeme`), nemusí být stejné jako `a.potrebujeme`, například když je a správně uzávorkované a `b.potrebujeme` > 0.

Začátek posloupnosti b ovlivňuje hodnota `a.navic` a závorky, jež se mají přidat před c, dokáží ovlivnit obě části c, takže platí, že `c.potrebujeme` je maximum z `a.potrebujeme` a `b.potrebujeme` $-$ `a.navic`.

Pro triviální řetězce je `"(".potrebujeme` $= 0$, `"(".navic` $= 1$, `")".potrebujeme` $= 1$, `")".navic` $= -1$. Můžeme tedy tyto proměnné určit pro triviální řetězce a spojováním se dostat až k řetězci délky N.

Teď přijde trik: většina proměnných je po otočení stejná jako před ním, a tak není potřeba počítat vždy všechny. Když si nad řetězcem postavíme binární strom, tak bude stačit změnit jen vrcholy nad pozicí, kde se otáčela závorka. Čili bude potřeba $\mathcal{O}(\log(N))$ přepočítání proměnných a poté se podívat do kořene, jestli je posloupnost správně uzávorkovaná.

Na začátku potřebujeme $\mathcal{O}(N)$ času na vytvoření toho binárního stromu, poté nám na dotaz stačí $\mathcal{O}(\log N)$ času. Paměťová složitost je $\mathcal{O}(N)$ kvůli uložení stromu.

Jitka Novotná

24-2-7 Štětcování

Tato úloha se projevila jako dosti zrádná. Většinou jste ji řešili tak, že jste si našli všechny vodorovné a svislé vybarvené úseky, z jejich délek vybrali minimum a to prohlásili za výsledek. Bohužel toto řešení nefunguje například na tomto vstupu:

```
11000
11100
00111
00011
```

Minimální souvislý vodorovný/svislý úsek má velikost dva, zatímco obrázek dokážeme nakreslit pouze štětcem velkým jedna.

Když jsem mluvil o zrádnosti úlohy, tak jsem to myslel vážně. Nikdo úlohu nevyřešil úplně správně a i já jsem měl ve svém původním řešení chybu. Jak to tedy mělo být?

Ukážeme si jedno řešení pomocí binárního vyhledávání a jedno lineární řešení.

Všimneme si, že pokud obrázek umíme vybarvit štětcem velkým K, tak jej umíme vybarvit i libovolným menším štětcem. Když tedy zvládneme v rozumném čase ověřit, zda lze obrázek vybarvit daným štětcem, můžeme správnou velikost binárně vyhledat.

Nejdříve si pro každé políčko spočítáme, kolik je ve sloupci pod ním černých políček. To zvládneme v čase $\mathcal{O}(R \cdot S)$. Dále si během výpočtu budeme pro každé políčko udržovat hodnotu H, jestli jsme toto políčko již vybarvili. Nyní pojedeme postupně po řádcích (budeme přikládat horní stranu štětce) a vždy, když budeme na místě, kde můžeme barvit, tak obarvíme všechna zatím neobarvená políčka a označíme je v H. Detaily výpočtu a jak postupovat při barvení, abychom si nepokazili časovou složitost, si můžete rozmyslet sami jako cvičení.

Každé políčko jsme obarvili maximálně jednou a zkusili jsme $\mathcal{O}(R \cdot S)$ pozic štětce, tedy tento krok zvládneme dohromady v $\mathcal{O}(R \cdot S)$. Společně s binárním vyhledáváním dostaneme časovou složitost $\mathcal{O}(R \cdot S \cdot \log \min(R, S))$.

Nyní k lineárnímu řešení. My vlastně pro každé políčko chceme zjistit, v jakém největším čtverci leží. Pak z těchto hodnot vybereme minimum a dostaneme správné řešení. Jak na to? Nejdříve předvedu algoritmus, který úlohu řeší, a pak dokážu, že odpovídá správně.

Odteď budeme černá políčka nazývat jedničkami a bílá políčka nulami. Spočítáme, v jakém největším čtverci jedniček se jedničky nachází. Ovšem maximální čtverce budeme hledat jen pro ty jedničky, které mají vedle sebe alespoň jednu

nulu. (Kraj považujeme za nulu.) U takových jedniček totiž dokážeme jednoduše určit, kterým směrem jejich čtverec povede.

Nyní bychom u každé jedničky chtěli znát, jak dlouhý souvislý úsek jedniček z ní vede směrem doprava, doleva, nahoru i dolů. To vše si dokážeme předpočítat v čase $\mathcal{O}(R \cdot S)$. Pak pro danou krajní jedničku použijeme následující postup:

Jednička má alespoň z jedné strany nulu, BÚNO* vlevo. Nyní se od této jedničky vydáme směrem doprava, budeme se koukat na délky horních a spodních úseků jedniček a udržovat jejich minima H_{min} a D_{min}. Půjdeme směrem doprava, dokud nenarazíme na nulu a dokud $H_{min} + D_{min} - 1 \geq K$, kde K je počet kroků, které jsme udělali. Jinými slovy zvětšujeme náš čtverec tak dlouho, dokud to jde.

Není těžké nahlédnout, že jsme našli největší čtverec, ve kterém naše jednička leží. Pokud tento postup uděláme pro všechny krajní jedničky a z výsledků vybereme minimum, tak dostaneme maximální velikost štětce. Tento postup má časovou složitost $\mathcal{O}(R \cdot S)$, protože jsme na každou jedničku obrázku přišli maximálně ze čtyř směrů.

Zbývá jen nahlédnout, že nám výsledek nemůže pokazit žádná jednička, která má za sousedy jen jedničky.

Pro takovou jedničku j uvažme největší čtverec, ve kterém leží. Takový čtverec určitě musí dvěma protějšími stranami sousedit s nulou, protože jinak bychom jej mohli zvětšit. Nyní se podíváme, jak se algoritmus choval u jedniček, které jsou ve čtverci vedle jedné z těchto dvou nul.

Pokud alespoň u jedné z nich najdeme právě takto velký čtverec, tak jsme vyhráli. Pokud ne, tak buď v jednom z těchto čtverců leží jednička j, nebo jeden z nich můžeme pošoupnout tak, aby v něm jednička j ležela. V obou případech dostáváme spor s tím, že jsme na začátku měli největší možný čtverec pro jedničku j.

Tím je řešení hotové. Vzorová implementace:

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-2-7.cpp`

Karel Tesař

24-2-8 Alfa-beta ořezávání a piškvorky

Úkol 1: čtyři v řadě

Nebylo těžké si tipnout, že v piškvorkách, kde vyhrává řada čtyř značek, na neomezeném hracím plánu vyhrává začínající hráč (křížek). Důkaz rozborem případů není ani moc dlouhý, bylo však třeba dát si pozor a nezkrátit ho moc.

* BÚNO = bez újmy na obecnosti

Největším chytákem úkolu bylo, že druhý hráč při obraně může vytvořit vlastní řadu dvou značek (tzv. *dvojici*) a pak se bude muset i první bránit, což někteří řešitelé opomněli zohlednit.

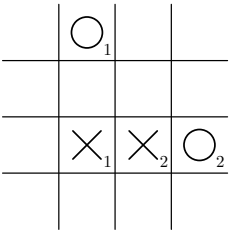
Pojďme tedy na rozbor případů, který se pokusíme co nejvíce zkrátit, aniž bychom něco zanedbali.

Klasickým trikem, jak v teorii her zredukovat počet probíraných případů, jsou symetrie. Když lze nějakou pozici získat z jiné pozice například otočením herního plánu nebo zrcadlením přes libovolnou osu a otočení či zrcadlení nemá v dané hře vliv na strategii hráčů, můžeme zkoumat obě pozice současně.

Jelikož hra je vyhraná pro prvního hráče, pro ověření jeho výhry si stačí pro tohoto hráče vždy vybrat nějaký tah, díky čemuž lze strom hry opět zmenšit (v úrovních prvního hráče). Je však třeba prozkoumat všechny protitahy soupeře.

Snadno si lze všimnout, že kdo udělá tři piškvorky v řadě s volnými políčky na obou koncích, vyhrál, nemá-li zrovna soupeř podobnou řadu. Taktéž vyhraje ten, kdo udělá najednou dvě dvojice, které obě mají volná políčka na koncích – za předpokladu, že soupeř nemůže dalším tahem udělat trojici s volnými konci.

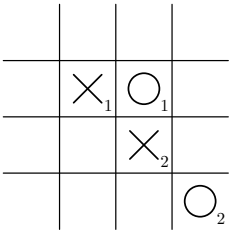
Křížek táhne libovolně a kolečko má následně až na otočení herní plochy tři možnosti: zahrát vpravo od křížku (dotýká se ho hranou), nebo vpravo nahoře (dotýká se rohem) anebo někam mimo (tak, aby se kolečko nedotýkalo křížku).



Zahraje-li kolečko mimo křížek, udělá první hráč dvojici nedotýkající se kolečka. Druhý pak musí dvojici blokovat a v některých případech ještě může zahrozit, že vytvoří trojici s volnými konci (viz obr. vlevo).

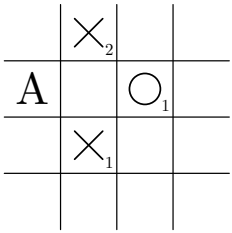
Křížek však takovou hrozbu dokáže odvrátit (což si není těžké rozmyslet) a navíc vždy udělá najednou dvě dvojice s volnými konci, čímž vyhrává.

Zahraje-li kolečko vpravo od křížku, první hráč umístí další značku pod kolečko. Nyní druhý musí blokovat dvojici, aniž by si mohl vytvořit vlastní dvojici (viz obrázek vpravo). Křížek dalším tahem vytvoří dvě dvojice s volnými konci a opět vyhrává.



Poslední případ, v němž první kolečko sousedí rohem s křížkem, je zajímavý tím, že pro křížek je pak výhodnější zahrát tah nesousedící s prvním křížkem do situace na obrázku na následující straně:

Kolečko musí nějak blokovat vznik trojice s volnými konci, přičemž má tři možnosti: nad ní, doprostřed (tím vytvoří vlastní dvojici) a pod ní.



V každém případě může křížek zahrát na políčko A, udělat dvě dvojice s volnými konci (případně ještě blokovat dvojici koleček), díky čemuž následně vyhraje.

Rozbor případů je hotov, takže nyní víte, jak za prvního hráče vyhrát, ať už bude soupeř vyvádět cokoliv (samozřejmě v rámci pravidel).

Úkol 2: devět v řadě

Zdůvodnění, proč má druhý neprohrávající strategii ve hře devět v řadě na neomezeném plánu, si skutečně zasloužilo svých 9 bodů i s nápovědou – správně ho měl jen jeden řešitel. Ukažme si tedy, jak bylo možné se s úlohou poprat.

Nápověda byla rozdělení párů (dvojic), čemuž se říká *párování*. Správným řešením bylo vytvořit je tak, aby každá možná řada devíti značek obsahovala nějakou dvojici a každé políčko bylo maximálně v jedné dvojici (v našem řešení bude každé políčko přesně v jedné dvojici).

Než si ukážeme vytvoření oněch dvojic, popíšeme tzv. *párovací strategii* pro druhého hráče, díky níž neprohraje. Druhý vždy reaguje na předchozí tah prvního hráče tak, že zahraje do druhého políčka dvojice, do níž zahrál první hráč.

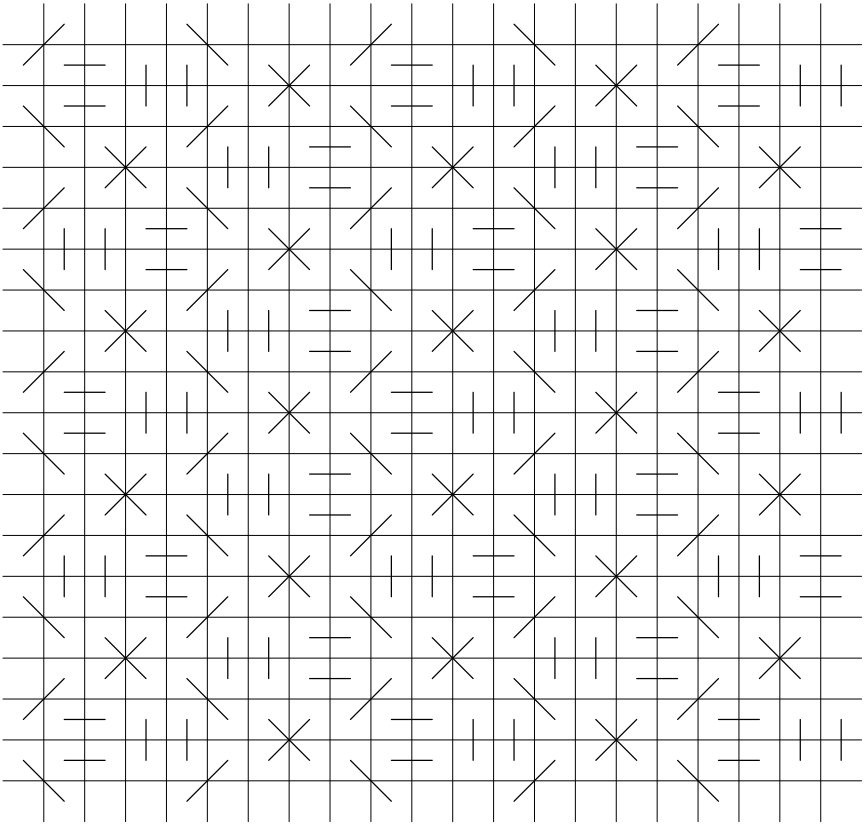
Tak je zajištěno, že po tahu druhého hráče bude každá dvojice z párování buď prázdná, nebo v ní bude kolečko a křížek. Díky tomu se nikdy nestane, že by v nějaké dvojici byly dvě stejné značky, takže nikdo nemůže dosáhnout řady devíti značek, jelikož každá taková řada obsahuje nějakou dvojici.

Jako rozdělení do dvojic si předvedeme *Hales-Jewettovo párování*. Obrázek vydá za tisíc slov, což v tomto případě platí dvojnásob – viz následující stranu.

Tento vzor se pořád opakuje, takže každé políčko herního plánu je v nějaké dvojici. Snadno lze ověřit, že každá možná řada devíti políček obsahuje nějaký pár.

Mimochodem, bylo možné si všimnout, že pokud je remízová varianta piškvorek, v níž vyhrává osm v řadě, musí být remíza i devět v řadě. Jenže o osmi v řadě jsme se jen letmo zmínili, bylo tedy třeba dokázat, že osm v řadě je remíza, což není vůbec, ale vůbec jednoduché.

Pavel „Paulie“ Veselý



24-3-1 Intervalové duplicity

Kuchařka na intervalové stromy, intervaly v názvu úlohy, dokonce nám i chodí dotazy na intervaly. „To prostě musí být intervalové stromy!“ Ale nejsou. Tato shoda náhod je jeden velký chyták. Je možné, že na tuto úlohu nějakým způsobem jdou napasovat intervalové stromy, ale rozhodně to nepatří k těm jednodušším řešením. Jaký tedy byl vzorový postup?

Celé řešení této úlohy je vlastně jen jeden velký trik. Nejdříve si všimneme, že pro dvojici čísel se stejnou hodnotou nás zajímá především interval, který má na krajích čísla z této dvojice... Pak o libovolném intervalu $[X, Y]$ řekneme, že je špatný, pokud v sobě obsahuje některý z těchto minimálních intervalů.

My dostaneme dotaz na interval $[L, P]$ a vše, co nás zajímá, je, jestli se v něm vyskytuje některý z minimálních špatných intervalů. Co kdybychom si pro každý možný pravý kraj intervalu $[L, P]$ předpočítali pozici A začátku nejbližšího levého minimálního intervalu $[A, B]$, který zároveň splňuje $B \leq P$? Pak bychom jen porovnali A a L. Pokud by $A < L$, tak by interval byl dobrý a v opačném případě by byl špatný. To bychom měli vyhráno!

Předpočítat si tyto hodnoty ale vůbec není těžké. Posloupnost projdeme zleva doprava a pro každou hodnotu si budeme pamatovat, kdy naposled jsme ji viděli. To si můžeme pamatovat například pomocí hešovací tabulky, nebo binárního vyhledávacího stromu. Ať už to bude cokoliv, říkejme tomu *mapa*. Dále si chceme pamatovat *poslední* minimální špatný interval nalevo od nás. Nyní pro všechny pozice k v posloupnosti zavoláme tyto příkazy:

```
poslední[k] = max(poslední[k-1], mapa[pole[k]])
mapa[pole[k]] = k
```

A to už vlastně máme hodnoty předpočítané. Teď jen stačí odpovědět na všechny dotazy.

Při použití binárního vyhledávacího stromu potřebujeme čas $\mathcal{O}(N \log N)$ na předpočítání a čas $O(Q)$ na zodpovězení dotazů, kde N je délka posloupnosti a Q je počet dotazů. Celkem tedy $\mathcal{O}(N \log N + Q)$. Při použití hešovací tabulky časová složitost závisí na použité hešovací funkci a průměrném množství vzniklých kolizí v tabulce. Tento rozbor složitosti zde vynecháme.

Paměťová složitost řešení je $\mathcal{O}(N)$. Potřebujeme si pamatovat konstantní množství informací ke každé hodnotě posloupnosti.

Ve vzorovém zdrojovém kódu je použita mapa z C++ knihovny STL, se kterou se pracuje stejně jako s hešovací tabulkou a vlastně stejně jako s asociativním polem, které používáte například v PHP.

Závěrem bych chtěl poznamenat, že na našich testovacích vstupech prošla na plný počet bodů i některá řešení s kvadratickou časovou složitostí s dostateč-

ným množstvím heuristik. Tohoto faktu si všiml Ondra Hübsch a my mu tímto děkujeme za upozornění.

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-3-1.cpp`

Karel Tesař

24-3-2 Nemnoho počítačů

Nejdříve si uvědomíme, že rychleji než v $\mathcal{O}(N)$ čísla počítačů nesetřídíme, protože je potřebujeme alespoň načíst a vypsat, což rychleji než lineárně nejde. Současně určitě umíme čísla počítačů setřídit v $\mathcal{O}(N \log N)$, protože v tomto čase umíme setřídit obecnou posloupnost čísel pomocí třídících algoritmů, jako jsou Quicksort nebo Mergesort. Nejde to však rychleji?

V došlých řešeních se objevovaly dva možné přístupy, popíšeme si tedy oba. Prvním z nich je použití *asociativního pole*, neboli hešování.

Řešení pomocí hešování

V rychlosti tu popíšeme pouze základní principy, koho by to zaujalo, může se podívat do odpovídající kuchařky o hešování.

Základem fungování heše je *hešovací funkce*. Ta přiřazuje *klíčům* (textových řetězcům nebo velkým číslům podle toho, jakými hodnotami chceme heš indexovat) nějaká malá čísla z rozsahu 0 až $K - 1$, pomocí nichž se již dá indexovat normální pole. Dobrým příkladem hešovací funkce pro velká čísla je například zbytek po dělení číslem K.

Když ale hešovací funkce přiřadí dvěma klíčům stejnou hodnotu, nastává *kolize*. V takovém případě je někdy nutné projít až celé pole a to trvá lineárně dlouho. Pro větší detaily se opět podívejte do kuchařky o hešování.[33]

Pokud se rozhodneme použít hešování, vytvoříme si asociativní pole o velikosti K, sloužící pro ukládání počtů jednotlivých typů počítačů. Číslo K zvolíme jako nějaké prvočíslo mezi $2 \log N$ a $4 \log N$ (takové prvočíslo mezi číslem a jeho dvojnásobkem určitě existuje, ale to si zde nebudeme dokazovat).

Proč právě takhle? Rozsah zhruba dvojnásobku počtu klíčů je rozumná volba z hlediska minimalizování počtu kolizí, ale současně pole ještě není příliš velké. A volba prvočísla je šikovná z hlediska hešovací funkce, která bude vracet zbytek po dělení K.

Poté již postupně procházíme vstupní posloupnost. Pokud se klíč odpovídající typu počítače v heši ještě nenachází, založíme ho, jinak ke stávajícímu počtu počítačů tohoto typu pouze přičteme jedničku.

[33] `http://ksp.mff.cuni.cz/viz/kucharky/hesovani`

Po načtení celého vstupu pak pole setřídíme, což nám vzhledem k jeho velikosti $\mathcal{O}(\log N)$ bude s použitím například Mergesortu trvat $\mathcal{O}(\log N \log\log N)$, což je méně než $\mathcal{O}(N)$. Poté již stačí jenom setříděné pole projít a u každého typu ho vypsat tolikrát, kolikrát byl na vstupu. To nám zabere lineární čas vzhledem k velikosti vstupu.

Paměťová složitost je úměrná velikosti pole, tedy $\mathcal{O}(\log N)$. Je ale časová složitost skutečně $\mathcal{O}(N)$? Vše záleží na volbě hešovací funkce a na číslech, která se vyskytnou na vstupu. V nejhorším případě může nastat u všech prvků heše kolize a zpracování kolize může stát až lineárně vzhledem k velikosti heše, tedy $\mathcal{O}(\log N)$ Časová složitost by tedy byla až $\mathcal{O}(N \log N)$.

Řešení pomocí vyhledávacích stromů

Druhým přístupem, který nám zajistí dobrou časovou složitost ve všech případech (i když to nebude tak dobré, jako $\mathcal{O}(N)$ u hešování v nejlepším případě), je použití vyhledávacích stromů. Vyvážený vyhledávací strom nám zaručuje přístup ke všem jeho prvkům v logaritmickém čase vzhledem k jeho velikosti.

Vyhledávací strom je strom s nějakou hodnotou v každém vrcholu. Pro každý vrchol platí, že všechny hodnoty v jeho levém podstromu jsou menší, než hodnota v daném vrcholu, a všechny hodnoty v pravém podstromu jsou zase větší, než hodnota v daném vrcholu.

Budeme potřebovat pouze přidávání do stromu, proto si popíšeme pouze to. Pro více detailů se podívejte do kuchařky o vyhledávacích stromech.[34] Přidávání je stejné jako vyhledávání. Začneme v kořeni a postupně se zanořujeme do levého nebo pravého podstromu (podle toho, jestli je hledaná hodnota menší nebo větší, než hodnota ve vrcholu), dokud nenarazíme na vrchol s hledanou hodnotou.

Nebo, pokud takový vrchol neexistuje a my ho chceme přidat, vytvoříme ho na daném místě jako levého nebo pravého syna vrcholu, kde jsme skončili. Při takovém přidání nám ale může strom degenerovat a může nám vzniknout až strom tvaru dlouhé lineární cesty. Pro zajištění podmínky přístupu ke všem prvkům v logaritmickém čase je nutné strom vyvažovat.

Vyvažování se provádí pomocí *rotací*. Prostě překořeníme nevyvážený podstrom za nějaký jeho vrchol tak, aby se hloubka levého a pravého podstromu vždy lišila maximálně o jedna. Zároveň samozřejmě nesmíme porušit uspořádání hodnot ve vrcholech – výsledkem rotace je opět binární vyhledávací strom.

Takovým stromům se říká *AVL stromy*, koho rotace zajímají více, nechť se opět začte do kuchařky o vyhledávacích stromech.

Nám stačí vědět pouze to, že jedna rotace trvá konstantně dlouho a při jednom vyvažování provedeme maximálně tolik rotací, kolik je hloubka stromu, tedy logaritmicky vzhledem k jeho velikosti.

[34] `http://ksp.mff.cuni.cz/viz/kucharky/stromy`

Nyní tedy máme datovou strukturu, do které můžeme v logaritmickém čase přidávat libovolné hodnoty. A navíc, pokud poté budeme strom procházet zleva (v každém vrcholu nejdříve zpracujeme levý podstrom, pak vrchol a nakonec pravý podstrom), dostaneme rovnou setříděnou posloupnost těchto hodnot. Procházení stromu zleva trvá lineárně vzhledem k jeho velikosti.

Základní idea programu tedy bude stejná jako v předchozím případě. Budeme načítat posloupnost na vstupu a každý prvek se pokusíme vložit do stromu. Pokud se už ve stromu tento typ počítače nachází, jenom ke stávajícímu počtu počítačů tohoto typu přičteme jedničku, jinak tento typ počítače založíme.

Velikost stromu bude stejná, jako je počet typů počítačů, tedy $\mathcal{O}(\log N)$ a přidání prvku do stromu včetně následného vyvážení bude stát $\mathcal{O}(\log \log N)$. Zpracování vstupu nám tedy zabere $\mathcal{O}(N \log \log N)$. Nakonec už pouze projdeme strom zleva a vypíšeme každý typ tolikrát, kolikrát se objevil na vstupu.

Paměťová složitost je v tomto případě také $\mathcal{O}(\log N)$. Nejvíce času nám zabere zpracování celého vstupu do stromu, tedy celková časová složitost je $\mathcal{O}(N \log \log N)$.

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-3-2.cpp`

Jirka Setnička

24-3-3 Párování znalců

Našou úlohou je zistiť počet hrán v maximálnom párovaní v strome.

Hranu, ktorá obsahuje vrchol, ktorý má len jedného suseda (a to druhý vrchol, ktorý patrí tej istej hrane) nazvyme *listová hrana*.

Algoritmus je jednoduchý. Vezmeme si ľubovoľnú listovú hranu a odoberieme zo stromu oba vrcholy patriace tejto hrane (odobrať vrchol znamená aj odobrať všetky hrany ktorým patrí). Za takýto krok si započítáme jednu hranu do párovania (tú listovú), tie čo sme odobrali spolu s vrcholom, ktorý v nájdenej listovej hrane nebol list, do párovania samozrejme nepočítame. Skončíme, keď už nemáme čo odobrať. To, že nám pri odoberaní listových hrán môže vzniknúť les, ničomu nevadí.

Prečo to funguje? Tak, predpokladajme, že e je listová hrana a M je nejaké maximálne párovanie v strome. Takže platí, že ak do M pridáme ľubovoľnú hranu, ktorá v M ešte nie je, tak M už nebude párovanie. Nech M neobsahuje listovú hranu e. Ak hranu e do M pridáme, tak práve jeden vrchol bude obsiahnutý v dvoch hranách párovania (lebo e je listová). Takže môžme odobrať nejakú z hrán v M a dostať maximálne párovanie, ktoré obsahuje e.

Môžeme si to predstaviť takto: vždy, keď nájdeme nejakú listovú hranu, tak na základe predchádzajúceho odstavca vieme, že existuje maximálne párovanie

(v tom, čo nám zo stromu na vstupe ešte zostalo) také, že obsahuje nájdenú hranu a preto môžeme vrcholy tejto hrany odobrať. A teda správnosť algoritmu je dokázaná, pretože vieme, že v každom kroku nič nepokazíme.

Algoritmus môžeme implementovať ako prehľadávanie stromu do hĺbky metódou postorder (s vrcholom niečo vykonáme až potom, čo sme dokončili prácu s jeho synmi): vždy, keď sa pozrieme na vrchol (to je už po tom, čo sme sa pozreli na všetkých jeho synov), tak zistíme, či nemá nejakého nespárovaného syna. Ak áno, tak vrchol spárujeme s ľubovoľným nespárovaným synom.

Čo sa časovej zložitosti týká, tak tá je lineárna vzhľadom na počet vrcholov, čiže $\mathcal{O}(n)$, kde n je počet vrcholov. Pamäťová zložitosť je na tom rovnako.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-3-3.c`

Peter Zeman

24-3-4 Návrat do podposloupnosti

Po přečtení zadání není těžké si uvědomit, že naším úkolem je vyškrtnout souvislou část posloupnosti tak, abychom dostali co nejdelší souvislou rostoucí podposloupnost.

Než začneme cokoliv vymýšlet, tak si uvědomíme, že by se nám pro každý prvek mohlo hodit znát, jak dlouhá souvislá rostoucí podposloupnost v něm končí a jak dlouhá souvislá rostoucí podposloupnost v něm začíná. Těmto hodnotám budeme říkat prefixy a sufixy prvků. Prefixy spočítáme tak, že posloupnost projdeme zleva doprava a budeme si průběžně pamatovat, jak dlouhá je poslední rostoucí část. Obdobně, při průchodu zprava doleva, spočítáme sufixy.

Teď teprve nad úlohou začneme přemýšlet. Pokud bychom nic neškrtli, tak odpovědí bude nejdelší prefix. Pokud vyškrtneme nějaký úsek $[a, b]$, tak se nám situace může zlepšit pouze v místě škrtání, pokud spolu můžeme spojit prefix končící v bodě $a - 1$ se sufixem začínajícím v bodě $b + 1$. Nikde jinde se nám situace nezmění.

Pro kvadratické řešení nám tedy stačí jen vyzkoušet všechny možné úseky a pro každý se podívat, jak dlouhá posloupnost vznikne po jeho vyškrtnutí. Pak jen vezmeme maximum ze všech hodnot, které jsme takto našli, a všech prefixů a řešení vypíšeme. Toto řešení má časovou složitost $\mathcal{O}(n^2)$. Zkoušíme $\mathcal{O}(n^2)$ úseků a z každého získáme zlepšení v konstantním čase.

Jde to ale řešit i lépe. Pokud si určíme, kde škrtaný úsek bude končit, tak budeme chtít efektivně zjistit, kde má škrtaný úsek začínat, abychom vytvořili co nejdelší souvislý rostoucí úsek. Jinými slovy nás zajímá, k jakému nejdelšímu prefixu, umístěnému směrem nalevo, jsme schopní tento sufix napojit.

K tomu využijeme datovou strukturu jménem intervalový strom, který je popsán v kuchařce ke třetí sérii. Konkrétně se nám bude hodit intervalový strom pro maxima, na začátku inicializovaný na samé 0. Jak přesně nám pomůže?

Hodnoty v posloupnosti si seřadíme podle velikosti od nejmenších po největší. Nyní postupně od nejmenších prvků budeme provádět tyto operace (pozor, první operace bez druhé nedává smysl):

1) Zeptáme se stromu na maximum na intervalu jedna až původní pozice prvku.
2) Do intervalového stromu na původní pozici prvku uložíme velikost rostoucího prefixu končícího tímto prvkem.

Vždy, když se intervalového stromu ptáme na maximum nějakého intervalu, tak v něm máme uložené prefixy všech menších prvků, tedy jediných prvků, na které má smysl se ptát. Tedy dostáváme správné odpovědi. A to je vlastně celé.

Toto řešení má časovou složitost $\mathcal{O}(n \log n)$. Prvky třídíme a pokládáme $\mathcal{O}(n)$ dotazů intervalovému stromu, kde každý zabere čas $\mathcal{O}(\log n)$.

Řešení pomocí intervalového stromu můžete najít ve zdrojovém kódu. Pro jednoduchost zápisu program jen zjišťuje, jak dlouhou posloupnost umíme vytvořit. Konkrétní posloupnost dostaneme tak, že intervalový strom upravíme, aby si pamatoval i to, odkud maximum pochází.

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-3-4.cpp`

Karel Tesař

24-3-5 Součin zlomků

Ukážeme si celkem tři postupně se zlepšující řešení. Stojí možná za poznámku, že jen pár řešitelů přišlo na první z nich a nikdo na druhé ani na třetí. Navíc se objevil netriviální počet řešitelů, kteří vzali příklad s jednobajtovými čísly za součást zadání, což je pochopitelně stálo nemalou část bodů. A nyní už k věci.

První varianta

Lze snadno nahlédnout, že rozložením čitatelů a jmenovatelů na prvočinitele a následným pokrácením dostaneme zlomek v základním tvaru. K rozkladu (neboli faktorizaci) použijeme pole prvočísel předpočítané známým algoritmem Eratosthenova síta. Ten ostatně budeme potřebovat i v následujících dvou řešeních.

Nejdříve tedy přečteme celý vstup a vybereme maximum M ze všech čitatelů a jmenovatelů. M nastavíme jako horní mez pro Eratosthenovo síto. Sítem získáme pole prvočísel, kde si následně u každého prvočísla budeme udržovat hodnotu jeho exponentu ve výsledku. Tu zjistíme tak, že znovu procházíme vstup

a každého z čitatelů, resp. jmenovatelů rozkládáme na prvočinitele a podle toho zvyšujeme, resp. snižujeme příslušný exponent o jedničku.

Tento algoritmus běží v čase $\mathcal{O}(M \log\log M + N \cdot K)$. Z toho $\mathcal{O}(M \log\log M)$ nás stojí síto (viz dodatek), $\mathcal{O}(N \cdot K)$ trvá rozklad na prvočinitele (K označíme počet prvočísel menších než M a pro každé z $2N$ čísel na vstupu projdeme v nejhorším všechna prvočísla). Pokud jako výstup chceme skutečného čitatele a jmenovatele, nejen jeho faktorizaci, musíme ještě započíst čas na umocňování. Ten se dá snadno shora odhadnout logaritmem maximální hodnoty D datového typu, tedy $\mathcal{O}(\log D)$.

Časová složitost je tedy $\mathcal{O}(M \log\log M + N \cdot K + \log D)$.

Druhá varianta

Eratosthenova síta se nejspíš nezbavíme, takže se zaměříme na druhou část algoritmu, a to na prvočíselný rozklad. Upravíme síto tak, aby si u složených čísel pamatovalo nejen to, že jsou vyškrtnutá, ale také některé z prvočísel, jimiž jsme je škrtli. Přesněji řečeno, když v sítu vyškrtáváme k-tý násobek prvočísla p, poznamenáme si do prvku pole $P[k \cdot p]$ číslo p.

K čemu je nám to dobré? Ve chvíli, kdy potřebujeme faktorizovat nějaké číslo i, podíváme se do $P[i]$ a tam najdeme jeden z faktorů. Tím i vydělíme a proces opakujeme tak dlouho, až v $P[i]$ bude 0. Faktorizace každého čísla nám nyní zabere $\mathcal{O}(\log M)$ (zkuste si rozmyslet, proč). Celková časová složitost tudíž klesla na $\mathcal{O}(M \log\log M + N \log M + \log D)$.

Třetí, nejlepší varianta

Opět využijeme vhodného předpočítání. Než spustíme síto, spočítáme si pro každé číslo od 1 do M hodnotu $C[i]$, která se rovná rozdílu počtu výskytů i v čitatelích a jmenovatelích. Kdykoliv pak v sítu vyškrtáváme násobky nějakého prvočísla p, posčítáme $C[i]$ všech vyškrtaných čísel a hned víme, kolikrát se prvočíslo vyskytuje ve výsledku. Jen přitom musíme dávat pozor na ta i, která jsou dělitelná vyšší mocninou p. Ta musíme započítat vícekrát.

Kdybychom neošetřili vyšší mocniny, trval by celý algoritmus $\mathcal{O}(N)$ pro výpočet pole C a $\mathcal{O}(M \log\log M)$ pro síto. Vyšší mocniny nám ale ve skutečnosti algoritmus nezpomalí. Pokud vyškrtáváme násobky prvočísla p, budou mě první mocniny stát n/p, druhé n/p^2, atd., což není nic jiného, než geometrická řada se součtem $\mathcal{O}(n/p)$. Celkově tedy dostáváme časovou složitost $\mathcal{O}(M \log\log M + N + \log D)$.

Paměťová složitost všech tří řešení je $\mathcal{O}(M + N)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-3-5.c`

Jan Bok

Složitost Eratosthenova síta

Eratosthenovo prvočíselné síto je jedním z vůbec nejstarších známých algoritmů (Eratosthenés z Kyrény žil ve 3. století př. n. l. a objevil ledacos zajímavého, například docela přesně spočítal velikost Země). Ovšem teprve v historicky nedávné době se matematici naučili spočítat, jakou má toto síto časovou složitost. Pojďme to také zkusit.

Uvažujme následující přímočarou implementaci síta:

```
for (int p=2; p<=n; p++)
   if (!sito[p])
      for (int j=2*p; j<=n; j+=p)
         sito[j] = 1;
```

Většinu času jistě trávíme ve vnitřním cyklu. Pokud zrovna vyškrtáváme násobky prvočísla p, projdeme jich $\lfloor n/p \rfloor$. Označíme-li všechna nalezená prvočísla $p_1 < p_2 < \ldots < p_k$, můžeme složitost celého síta zapsat jako $\mathcal{O}(n/p_1 + \ldots + n/p_k) = \mathcal{O}(n \cdot s)$, kde $s = 1/p_1 + \ldots + 1/p_k$, tedy součet převrácených hodnot všech prvočísel od 1 do n.

Přesný vzorec pro s není znám, ale ukážeme, jak hodnotu s omezit shora.

Nejprve to zkusíme poměrně hrubě: doplníme do součtu i převrácené hodnoty ostatních čísel. Tedy $s \leq 1/1 + 1/2 + 1/3 + \ldots + 1/n$. Tomuto součtu se říká n-té harmonické číslo a značí se H_n. Za chvíli dokážeme, že $H_n = \mathcal{O}(\log n)$, takže síto doběhne v čase $\mathcal{O}(n \log n)$.

Aby se nám H_n počítalo snáz, budeme předpokládat, že n je mocnina dvojky. (Pokud by nebylo, prostě ho zaokrouhlíme na nejbližší vyšší mocninu dvojky n', čímž nevzroste víc než dvojnásobně. Dostaneme $H_n \leq H_{n'} = \mathcal{O}(\log 2n) = \mathcal{O}(\log n)$.)

Zlomky v harmonickém součtu rozdělíme na bloky po mocninách dvojky:

$$H_n = \left(\frac{1}{1}\right) + \left(\frac{1}{2}\right) + \left(\frac{1}{3} + \frac{1}{4}\right) + \left(\frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8}\right) + \ldots.$$

V i-tém bloku se tedy nacházejí čísla od $1/(2^{i-1}+1)$ do $1/2^i$. Blok tudíž obsahuje 2^{i-1} čísel a všechna jsou menší než $1/2^{i-1}$, takže součet bloku je nejvýše 1. (To platí i pro nultý blok $1/1$, který jinak z pravidelné struktury vybočuje.)

Jelikož každý blok přispěje nejvýše jedničkou a bloků je $\mathcal{O}(\log n)$, platí $H_n = \mathcal{O}(\log n)$.

Mimochodem, podobně můžeme dokázat, že každý blok přispěje aspoň $1/2$, takže H_n můžeme logaritmem omezit i zespoda.

Logaritmický odhad součtu s je sice pěkný, ale ještě jsme vůbec nevyužili toho, že součet obsahuje jen prvočíselné členy. Podobně jako předtím

budeme předpokládat, že n je mocnina dvojky, součet rozdělíme na bloky a omezíme shora součet jednoho bloku, řekněme toho mezi $n/2$ a n.

Nejprve spočítáme, kolik mezi $n/2$ a n leží prvočísel. Označme P množinu všech těchto prvočísel, tedy:

$$P = \{p \mid p \text{ je prvočíslo } \wedge\ n/2 < p \leq n\}.$$

Bude se nám hodit následující kombinační číslo:

$$C = \binom{n}{n/2} = \frac{n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n/2+1)}{(n/2) \cdot (n/2-1) \cdot \ldots \cdot 2 \cdot 1}.$$

Dokážeme následující nerovnosti:

$$(n/2)^{|P|} \leq \prod_{p \in P} p \leq C \leq 2^n.$$

Třetí nerovnost platí, jelikož libovolná n-prvková množina má celkem 2^n podmnožin a číslo C udává počet jejích $(n/2)$-prvkových podmnožin, takže musí být menší.

Druhou nerovnost dostaneme z toho, že každé prvočíslo $p \in P$ je dělitelem našeho C: v prvočíselném rozkladu čitatele se p vyskytuje právě jednou a ve jmenovateli ani jednou. A jelikož je C dělitelné všemi prvočísly z P, musí být dělitelné i jejich součinem, takže C je aspoň tak velké, jako tento součin.

První nerovnost je nejsnazší: všechna $p \in P$ jsou větší nebo rovna $n/2$.

Nyní nerovnosti složíme:

$$(n/2)^{|P|} \leq 2^n$$

a zlogaritmováním získáme:

$$(\log_2 n - 1) \cdot |P| \leq n,$$

z čehož vyjádříme počet prvočísel v množině P:

$$|P| \leq n/(\log_2 n - 1) = \mathcal{O}(n/\log n).$$

Dokázali jsme tedy, že mezi $n/2$ a n leží nejvýše $\mathcal{O}(n/\log n)$ prvočísel. Součet převrácených hodnot těchto prvočísel už omezíme snadno:

$$\sum_{p \in P} \frac{1}{p} \leq \sum_{p \in P} \frac{2}{n} \leq \mathcal{O}(n/\log n) \cdot \frac{2}{n} = \mathcal{O}(1/\log n).$$

Vraťme se k původní otázce, totiž k součtu převrácených hodnot všech prvočísel mezi 1 a n. Ta mezi $n/2$ a n, čili v posledním bloku, jsme už započítali, teď stejným způsobem započteme i bloky předcházející:

$$\begin{aligned} s &= \mathcal{O}\left(\frac{1}{\log n} + \frac{1}{\log n/2} + \frac{1}{\log n/4} + \ldots + \frac{1}{\log 2}\right) \\ &= \mathcal{O}\left(\frac{1}{\log n} + \frac{1}{(\log n) - 1} + \frac{1}{(\log n) - 2} + \ldots + \frac{1}{1}\right). \end{aligned}$$

To je ovšem až na konstantu skrytou v $\mathcal{O}$ rovno $(\log n)$-tému harmonickému číslu, čili $\mathcal{O}(\log \log n)$.

Dokázali jsme tedy, že Eratosthenovo síto doběhne v čase $\mathcal{O}(n \log \log n)$.

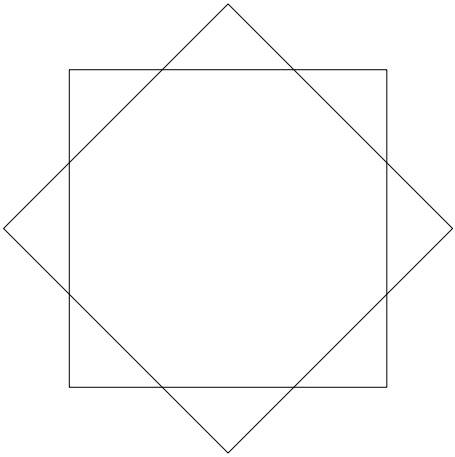
Martin „Medvěd“ Mareš

24-3-6 Průnik plánů

Úloha nebyla tak těžká, jak se na první pohled zdála. Stačilo nebát se a nenechat se ukolébat jednoduchostí vzorového obrázku.

Nejpřímočařejším řešením je uvědomit si, které vrcholy budou ohraničovat hledaný průnik. Vrchol jednoho mnohoúhelníka, který je uvnitř nebo na hranici druhého, bude určitě vrcholem průniku. Stejně tak průsečík stěn mnohoúhelníků bude vrcholem jejich průniku. Žádný jiný bod jistě nebude vrcholem průniku.

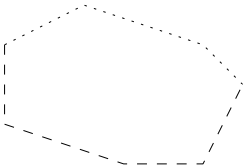
Na tomto místě většina z řešitelů zajásala a řekla, že maximální počet průsečíků stěn bude nějaká konstanta, nejčastěji čtyři. To ale není pravda. Obou typů vrcholů průniku může být $\mathcal{O}(n)$, kde n je počet vrcholů na vstupu. Lineární počet vrcholů uvnitř jednoho mnohoúhelníku si lze představit snadno – druhý mnohoúhelník bude celý uvnitř prvního. Lineární počet průsečíků stěn mají například dva soustředné pravidelné n-úhelníky. Pro šest průsečíků je to známá Davidova hvězda, pro osm dva pootočené čtverce.



I na základě této myšlenky by šel vymyslet hezký program. My si však ukážeme daleko jednodušší algoritmus.

Napřed nastiňme jeho myšlenku. Horní hranice průniku nebude výš než minimum z horních hranic obou mnohoúhelníků. Obdobně dolní hranice nebude níž než maximum.

Pro snazší popis si rozdělme konvexní obal na *horní* a *dolní obálku.* To jsou části, které vedou od nejlevějšího k nejpravějšímu vrcholu „horem“ a „spodem“. Pokud by byly dva vrcholy se stejnou x-ovou souřadnicí, berme vždy ten z nich, který má větší y-ovou souřadnici. Obálky si pamatujme v poli jako vrcholy seřazené podle x-ové souřadnice.



Rozdělení na horní a dolní obálky zvládneme snadno v lineárním čase, pokud máme vrcholy zadané už seřazené podle x-ové souřadnice nebo po obvodu konvexního obalu. Kdybychom měli vrcholy zadané jako nesetříděnou množinu, potřebovali bychom ještě třídit. Tento čas nebudeme počítat do výsledného času.

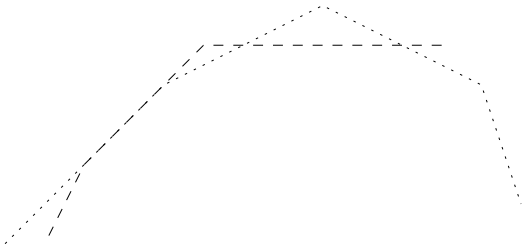
Kolmý průmět množiny bodů M na osu x je množina bodů na ose x takových, že když jimi vedeme kolmici, tak tato kolmice má neprázdný průnik s množinou M.

Pomocí horních obálek sestrojme horní lomenou čáru, která bude jejich minimem. Její kolmý průmět na osu x bude průnikem kolmých průmětů horních obálek. Na postup tvorby horní lomené čáry se mohou zkušení řešitelé geometrických úloh a znalci košťat dívat jako na zametání roviny.

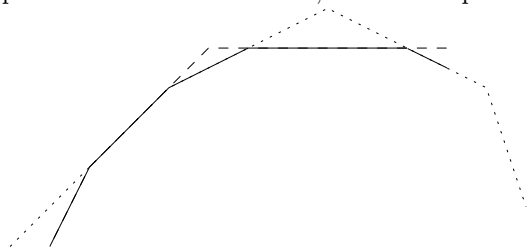
Z obou horních obálek si udržujme jednu úsečku, se kterou budeme pracovat. Na začátku to budou první úsečky z obálek. Dokud mají prázdný průnik kolmých průmětů na osu x (tedy dokud neexistuje přímka kolmá na osu x, která má s oběma úsečkami společný aspoň jeden bod), nahradíme úsečku s menší x-ovou souřadnicí za následující v její obálce.

Dokud je průnik kolmých průmětů pracovních úseček neprázdný, přidáváme do horní lomené čáry hraniční body části jedné úsečky, která je pod druhou, nebo s ní splývá. Jakmile dojdeme na konec některé z našich pracovních úseček, vezmeme z její obálky další. Zjišťování, která část jedné úsečky je pod druhou, nebo s ní splývá, zabere konstantní čas.

Snadným rozborem případů nahlédneme, že do horní lomené čáry přidáme nejvýš dvě úsečky v každém pásu kolmém na osu x a vyhraničeném průnikem jejich kolmých průmětů. Takových úseků je lineárně s počtem úseček v obou obálkách.



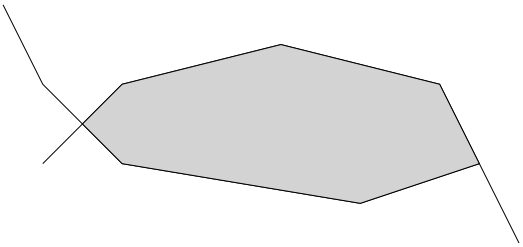
Lomenou čáru si ještě „uklidíme“ – odebereme z ní vrcholy, které jsou na spojnici dvou sousedních vrcholů. Jak výrobu, tak uklízení lomené čáry stihneme v čase $\mathcal{O}(n)$. Obdobně vyrobíme i dolní lomenou čáru, ale nesmíme zapomenout, že v tomto případě hledáme lomenou čáru, která vede po maximu z obálek.



Obdobným zametením, jako když jsme tvořili lomené čáry, určíme hranice oblasti, kde je horní lomená čára nad dolní. Můžeme si všimnout, že to bude souvislá oblast. Průnik konvexních množin je totiž konvexní množina. Případný důkaz plyne z definice – množina bodů M je konvexní, právě když pro každé dva body $x, y \in M$ leží i celá úsečka xy v M. Pokud x, y náleží průniku konvexních množin, náleží tam i jimi daná úsečka, protože x, y leží v průniku, takže musely ležet ve všech konvexních množinách, stejně jako jimi daná úsečka.

Obě lomené čáry musí mít stejnou x-ovou souřadnici začátku (a symetricky i konce). Je to dáno tím, že jejich kolmý průmět na osu x je roven průniku kolmých průmětů zadaných konvexních mnohoúhelníků na osu x. Lomené čáry se musí dvakrát protnout nebo dotknout. Pokud by se nedotkly, znamenalo by to, že minimum z horních obálek je větší než maximum z dolních. Ale horní obálka se v konvexním mnohoúhelníku vždy dotýká dolní.

Postupujeme po lomených čarách a část, kde horní je nad spodní, si zapamatujeme a vypíšeme. Musíme vypsat i případné průsečíky úseček tvořících lomené čáry. Opět stihneme v lineárním čase.



Dokažme ještě korektnost. Pokud leží bod v námi vypsané oblasti, jeho x-ová souřadnice je z průniku kolmých průmětů zadaných konvexních mnohoúhelníků na osu x. Navíc leží pod minimem z horních obálek a nad maximem z dolních

obálek. Tedy leží v průniku oněch konvexních mnohoúhelníků. Naopak, pokud bod leží v průniku, musí ležet i ve vypsané oblasti.

Celkem tedy časová složitost algoritmu je $\mathcal{O}(n)$ (bez třídění, které není potřeba, pokud jsou vstupem body v jejich pořadí na konvexním obalu). Paměťová složitost je také lineární, protože si nepamatujeme víc než konstantně mnoho lineárně velkých polí.

Karel Král

24-3-7 Mazání závorek

Predpokladajme, že N je párne, pretože v opačnom prípade nemá význam uvažovať o správnosti uzátvorkovania a podobne musí platiť, že $K \leq N/2$.

Základnou myšlienkou pri riešení tejto úlohy je použiť zásobník k overeniu správnosti uzátvorkovania. Otváracie zátvorky postupne ukladáme do zásobníka. Ak narazíme na uzavieraciu zátvorku, tak ak je zásobník prázdny (momentálne v ňom nie sú otváracie zátvorky) alebo typ otváracej zátvorky na vrchu zásobníka sa nezhoduje s typom uzavieracej zátvorky, potom uzátvorkovanie určite nie je správne.

V prípade, že nám v zásobníku po vyčerpaní zátvoriek ostanú ešte nejaké otváracie, je uzátvorkovanie nesprávne. Inak ho môžeme prehlásiť za správne.

Budeme teda postupne čítať vstup. Ak je zátvorka na vstupe otváracia, tak ju vložíme na vrch zásobníka. Ak je uzavieracia, tak rozlíšime nasledujúce možnosti:

- Zásobník je prázdný.
- Na vrchu zásobníka je príslušná otváracia zátvorka.
- Na vrchu zásobníka je otváracia zátvorka iného typu.

V prvom prípade je nutné skontrolovať správnosť uzátvorkovania, v ktorom odignorujeme zátvorky typu práve spracovávanej uzavieracej zátvorky (je jednoduché si rozmyslieť, že stačí odignorovať len tento jeden typ). Podobne spravíme v treťom prípade, avšak musíme navyše skontrolovať správnosť uzátvorkovania, v ktorom odignorujeme zátvorky typu otváracej zátvorky na vrchu zásobníka. (opäť je jednoduché si rozmyslieť, že stačí kontrolovať tieto dva typy). V druhom prípade odoberieme otváraciu zátvorku z vrchu zásobníka.

Ak nakoniec ostane zásobnik prázdny, vieme, že je všetko v poriadku a môžeme prehlásiť uzátvorkovanie za správne. Inak môžeme skúsiť skontrolovať správnosť uzátvorkovania, v ktorom odignorujeme zátvorky typu otváracej zátvorky na vrchu zásobníka. Časová zložitosť je lineárna vzhľadom na dĺžku vstupu.

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-3-7.cpp`

Peter Zeman

24-3-8 Sčítáme hry s panem Conwayem

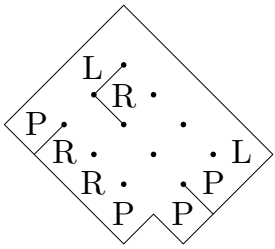
Úkol 1: Maze

Jednoduchá hra Maze spočívající v posouvání žetonu po plánu byla prostým cvičením na definice her prohraných, vyhraných, her levého a pravého (tedy tříd P, V, L a R). Řešení úkolu potěšila, chyb nebylo mnoho a většinou zřejmě z nepozornosti.

Aby nějaká počáteční pozice žetonu mohla být označena správnou třídou, je třeba určit, jak dopadnou pozice, kam z ní lze táhnout (ne vždy nutně všechny, ale hodí se to). Proto bylo vhodným postupem označovat políčka odspodu.

Jako první bylo možné zařadit do třídy P políčka, v nichž nemá žádný hráč žádný tah. Dále pozice žetonu, v nichž jeden hráč nemá tah a druhý může zahrát do prohrané pozice, patří jasně do třídy L nebo R (podle toho, kdo má tah).

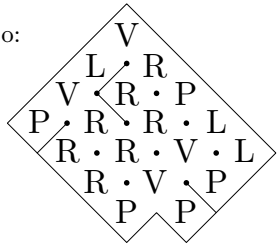
Dostaneme se tak k tomuto částečnému mezivýsledku (písmena tříd vkládáme pro jednoduchost přímo na políčka ve hře, jak to ostatně dělala většina řešitelů):



Pak se odspodu určují políčka tak, že na každém se pro oba hráče zjistí, jestli mohou z této pozice vyhrát tahem do prohrané pozice nebo do jejich pozice (tj. pro levého do pozice L). Podle toho se určí třída, do níž náleží políčko.

Například tedy políčko prohrané pro začínajícího je to, z něhož vedou všechny tahy levého do pozic pravého nebo do pozic vyhraných a všechny tahy pravého do pozic levého nebo vyhraných. Na pozici levého má levý tah, kterým vyhraje, a pravý ne.

Výsledný plán se zařazenými políčky vypadá takto:

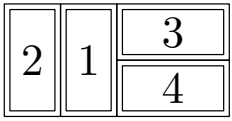


Úkol 2: dlouhé dominování

Prázdná mřížka o rozměrech $2 \times 4k$ (pro každé přirozené k) je vždy vyhraná pro pravého hráče pokládajícího vodorovná domina. (V tomto řešení se uvažuje mřížka se 2 políčky na výšku a se $4k$ na šířku. Pokud jste ve svém řešení měli mřížku otočenou, nevadilo to, jen je třeba prohodit L a R, levého a pravého, tedy prostě celou hru obrátit.)

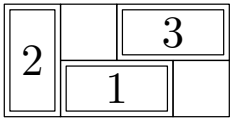
Jednou z možností, jak to ukázat (či vůbec zjistit výsledek), bylo najít pro pravého vyhrávající strategii, když začíná i když nezačíná. My si ukážeme jednodušší argument založený na sčítání her, který také dává pravému strategii vedoucí k výhře.

Nejprve rozebereme nejmenší případ, mřížku 2×4. Začne-li levý, může zahrát do sloupečku u kraje, nebo ve prostředku (ostatní dvě možnosti jsou symetrické k těmto). V obou případech pravý položí někam své domino, nyní má levý jen jeden tah a pravý také, jenže levý je na tahu, takže prohraje.



Na obrázku je jeden z případů, druhý si lze snadno domyslet:

Pokud začne pravý, zahraje doprostřed (je jedno, zda nahoru nebo dolu). Levý položí domino doleva, nebo doprava (opět symetrické případy), načež pravý mu druhou možnost sebere položením posledního volného domina přes poslední volný sloupec (viz obrázek), čímž vyhraje.



Jelikož pravý vyhrál, není třeba zkoumat další možnosti jeho prvního tahu.

Mřížka 2×4 tedy náleží do třídy R. Mřížky $2 \times 4k$ pro $k > 1$ vyřešíme sčítáním tak, že je rozdělíme na k nepřekrývajících se bloků 2×4. Všimneme si, že levý nemůže zahrát do obou bloků současně, pravý však ano.

My ovšem chceme dokázat, že pravý vždy vyhraje, takže si můžeme dovolit ho omezit (pokud i s omezením stále vyhraje). Zakážeme mu tahy do dvou bloků současně, díky čemuž se bloky 2×4 stávají nezávislými hrami. Celá mřížka $2 \times 4k$ je pak jejich součtem.

Všechny bloky má vyhrané pravý, takže i jejich součet má vyhraný pravý (formálně použijeme indukci dle k, přičemž indukční krokem je sečtení mřížek $2 \times 4(k-1)$ a 2×4, jež obě náleží do R). A je to dokázáno.

Navíc doplníme strategii pro druhého na mřížce $2 \times 4k$. Začne-li levý, hraje pravý vždy do stejného bloku jako levý dle strategie pro mřížku 2×4. Pokud začne pravý, táhne doprostřed nějakého bloku (opět dle své vyhrávající strategie pro jeden blok) a pak hraje do stejného bloku jako předtím levý.

Úkol 3: rovnající se hry

Tento úkol se nakonec ukázal býti nejtěžším, soudě dle počtu správných řešení. Kdo nepřišel na následující vcelku jednoduchý důkaz, pustil se do rozboru případů podle toho, do jaké třídy náleží hry G a H. Jenže ten obsahuje spoustu skrytých záludností kvůli tomu, že G a H mohou vypadat o dost jinak, proto se mu budeme stručně věnovat.

Celkem zřejmě náleží G i H do stejné třídy (je to vidět z definice rovnosti, když přičteme prohranou hru, v níž nikdo nemá tah). Pokud je H ve třídě V nebo P, je $-H$ ve stejné třídě. Je-li H hra levého, je $-H$ hra pravého (a opačně pro hru pravého).

Pokud je G prohraná nebo vyhraná hra, pak máme součet dvou prohraných her, respektive dvou vyhraných (z jedné lze tahem udělat buď prohranou hru, nebo hru hráče, co táhl) a lze použít následující část (součet her z L a R) nebo důkaz ze seriálu (přičtení prohrané hry nemění výsledek).

Důkaz v seriálu však obsahoval chybu, za níž se hluboce omlouvám – vyhranou hru lze totiž tahem změnit nejen na prohranou hru, ale i na hru toho hráče, co táhl (je to vidět například v dominování na mřížce 2×2). Toto jsem tedy v řešeních toleroval a v seriálu opravil.

Nejtěžší byl rozbor, když G je hra levého (a analogicky pravého). Asi nejlepší bylo argumentovat stejnou převahou levého či pravého v G a H, neboli stejným počtem tahů pro levého i pravého, což však není lehké obecně spočítat (k úvahám těžších případů se místo dominování hodí spíše abstraktní zápis her).

Tolik v krátkosti k řešením rozborem případů. Obecně se nad ním bylo potřeba pořádně zamyslet, jestli je opravdu v pořádku.

Nyní o poznání jednodušší řešení. Z definice rovnosti G a H dopadnou hry $G+X$ a $H+X$ pro libovolnou hru X stejně. Speciálně to platí pro hru $-H$, tedy $G-H$ dopadne stejně jako $H-H$.

Rozbor, jak dopadne $H-H$ je už o dost jednodušší než rozbor $G-H$. Druhý hráč použije tzv. *zrcadlící* strategii. Táhne-li první do H, zahraje druhý do $-H$ ten samý tah, který tam z definice obrácené hry musí být. Obdobně, po tahu prvního do $-H$ hraje druhý do H.

Takto se druhý po prvním pořád opičí. Zároveň prvnímu musí dojít tahy dříve než druhému, díky čemuž druhý vyhrává. Tedy $H-H$ je prohraná hra a $G-H$ také.

Tím je hotovo. Nezbývá nic jiného než vám popřát hodně štěstí do dalšího řešení.

Pavel „Paulie“ Veselý

24-4-1 Iniciály předků

Nejdříve si přeformulujeme zadání. Naším úkolem je pro daný řetězec zjistit jeho nejkratší periodu. Tedy takový podřetězec, jehož opakováním dostaneme celý řetězec.

Pro řešení využijeme algoritmus KMP, který je popsán v kuchařce ke čtvrté sérii. V zadaném řetězci si vytvoříme zpětné hrany, jako bychom jej chtěli vyhledávat v textu a všimneme si, že o něm platí následující tvrzení: Buď s periodický řetězec délky n s periodou o velikosti $p < n$. Potom je p rovno délce nejdelší zpětné hrany řetězce s spočítané algoritmem KMP.

Stačí se tedy kouknout, jaká je nejdelší zpětná hrana a ověřit, zda takto dlouhý počáteční úsek nám složí celý řetězec. Pokud ano, tak máme délku periody a pokud ne, tak nejkratší periodou je celý řetězec.

Zbývá nám jen dokázat naše malé tvrzení. Zpětnou hranu delší než je perioda řetězce jednoduše mít nemůžeme, protože by pak řetězec nebyl periodický (celá perioda by zpětnou hranou byla přeskočena). Může se nám tedy stát, že by nejdelší zpětná hrana byla menší?

Nechť je délka nejdelší zpětné hrany d menší než délka periody p. O zpětných hranách víme, že každá další je buď stejně dlouhá jako předchozí, nebo delší. Z toho vyplývá, že od jistého místa řetězce budou všechny zpětné hrany stejně dlouhé.

My se podíváme na dva po sobě jdoucí úseky periody na místě, kde už se vyskytují jen nejdelší zpětné hrany. Pokud je takové místo moc na konci, tak pár period přidáme. Nyní ze zpětných hran, které jsou dlouhé d víme, že

$$s_p = s_{p-d}$$
$$s_{p+1} = s_{p-d+1}$$
$$s_{2p-1} = s_{2p-d-1}$$

Z toho dostaneme, že některé znaky v rámci periody musí být stejné. Například pro $p = 5$ a $d = 3$ dostaneme, že všechny znaky jsou stejné a tedy, že perioda je vlastně 1. Obecně pro dané p a d dostaneme z vynucených shodných znaků menší periodu, která bude rovna přesně $\mathrm{nsd}(p, d)$, což je spor s tím, že řetězec má periodu p. Délka nejdelší zpětné hrany nemůže být ani větší, ani menší než délka periody. Tedy musí nastat rovnost.

Časová složitost algoritmu je lineární. Řetězec projdeme jen jednou při stavbě zpětných hran a jednou pro ověření, že řetězec má periodu rovnu délce nejdelší zpětné hrany. Paměťová složitost je taktéž lineární, uchováváme jen řetězec a jeho zpětné hrany.

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-4-1.cpp`

Karel Tesař

24-4-2 Sledování exponátů

Niektorí z vás sa pokúšali úlohy vyriešiť príliš mocnými nástrojmi. Zpravila týmto vzniklo správne, avšak pomalé riešenie.

Jednoduchý algoritmus je viesť polpriamku s počiatočným bodom X, kde X je bod, o ktorom chceme rozhodnúť, či je v mnohouholníku. Ak táto polpriamka pretne mnohouholník párny počet-krát, tak sme vonku, inak vnútri.

Pre každú hranu mnohouholníka zistíme, či ju náhodne zvolená polpriamka pretína. Priesečník polpriamky a úsečky nájdeme v čase $\mathcal{O}(1)$ – ak neviete ako, pozrite sa do geometrickej kuchárky.[35] Ak má mnohouholník N vrcholov, má taktiež N hrán; všetky priesečníky nájdeme v čase $\mathcal{O}(N)$.

Ak by sme náhodou polpriamkou trafili do nejakého vrcholu, zvolíme inú polpriamku. Môžeme očakávať, že mimo vrchol sa trafíme na $\mathcal{O}(1)$ pokusov, takže si časovú zložitosť nepokazíme.

Správnosť odargumentujeme tým, že ak sme vonku, tak za každé preťatie mnohouholníka, keď doň vchádzame, musíme mnohouholník preťať aj keď z neho vychádzame, teda preťatí musí byť párny počet. Ak sme naopak vnútri, tak situáciu prevedieme na prvý prípad tým, že z neho vyjdeme, čo nám dá jedno preťatie a teda celkovo máme nepárny počet preťatí.

Poznamenám na záver zaujímavosť, že toto funguje vďaka tomu, že platí Jordanova veta o kružnici,[36] ktorá vlastne hovorí to, že každá spojitá, uzavretá krivka nepretínajúca samu seba rozdeľuje rovinu na dve disjunktné časti. Formálny dôkaz tohoto (zdanlivo) zrejmého tvrdenia dá v matematike prekvapivo veľa práce.

Peter „pizet“ Zeman

24-4-3 Cinkání skleničkami

Nejdříve ukažme dolní odhad počtu taktů a potom teprve hledejme kýžený způsob, jak to provést.

Snadno nahlédneme, že pokud si mají cinknout všichni se všemi, je počet cinknutí roven počtu hran úplného grafu o N vrcholech, tedy $\binom{N}{2} = \frac{N(N-1)}{2}$. Dále rozlišujme situaci dle parity N.

Je-li N liché, je maximální počet cinknutí v jednom taktu roven $\frac{N-1}{2}$. Tedy v každém taktu si jeden necinká, protože nemá nikoho do páru (proto $N-1$) a každé cinknutí počítáme jen jednou (proto dělíme dvojkou). Tedy minimální počet taktů je roven N.

[35] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`

[36] `http://en.wikipedia.org/wiki/Jordan_curve_theorem`

Obdobně postupujeme pro sudá N. Dostáváme tak dolní odhad $N-1$. Ten ale můžeme vylepšit. Uvažme situaci, kdy si má v rámci jednoho z taktů cinknout např. první se třetím (účastníky číslujeme postupně po směru hodinových ručiček). V tu chvíli si druhý nemůže cinknout s nikým, neboť by musel zkřížit ruce s prvním a třetím. Pro sudá N teď máme odhad také N. Výjimku tvoří N rovno dvěma. V takovém případě lze cinknutí provést na jeden takt.

Víme tedy, že potřebujeme alespoň N taktů. Nyní již ukážeme, že dokážeme najít způsob, jak cinkání na N taktů provést. Představme si přípitek jako kruh, na jehož obvodu je rovnoměrně rozestavěno N účastníků přípitku. Dále všechny lidi očíslujme po směru hodinových ručiček 1 až N. Opět rozdělíme situaci dle parity N.

Nejdříve uvažme N liché. Člověk s číslem 1 si v tomto taktu necinkne. Pro j od 1 do $\frac{N-1}{2}$ si cinkne $(j+1)$-tý s $(N-j+1)$-tým. Každé cinknutí si představme jako úsečku mezi příslušnými lidmi. Snadno nahlédneme, že každá z úseček je rovnoběžná s ostatními, tedy podmínka nekřížení je splněna.

V dalším taktu pootočíme číslování o 1 proti směru hodinových ručiček a pokračujeme stejným způsobem. Pokud otočení opakujeme $(N-1)$-krát, dostali jsme i s počátečním rozložením N různých situací, v nichž každý z lidí necinkal právě jednou. Tedy si musel nutně cinknout se všemi ostatními.

Nyní pro sudé N. Pro prvních $\frac{N}{2}$ taktů použijme následující způsob. V prvním taktu si j-tý cinkne s $(N-j+1)$-tým pro j od 1 do $\frac{N}{2}$. Nyní i po každém z následujících $\frac{N}{2}-1$ taktů provedeme přečíslování stejným způsobem jako v případě lichého N. Úsečky reprezentující cinknutí jsou opět rovnoběžné. Vidíme, že takto si cinknou všechny páry ve tvaru lichý a sudý, resp. sudý a lichý. Situace totiž jsou opět různé a jejich počet je stejný jako počet lidí označených sudým, resp. lichým číslem.

Pro dalších $\frac{N}{2}$ taktů zvolíme cinkání následovně. V $(\frac{N}{2}+1)$-tém taktu první a $(\frac{N}{2}+1)$-tý stojí a pro j od 2 do $\frac{N}{2}$ si cinkne j-tý s $(N-j+2)$-tým. Pro každý další takt opět přečíslujeme. Jelikož je parita obou cinknuvších stejná, situace jsou opět různé; je jich $\frac{N}{2}$ a každý z účastníků stojí právě jednou. Dostáváme tady konečně způsob cinkání na N taktů i pro sudá N.

Jan Bok

24-4-4 Vozíky ve skladu

„Nebýt těch silnoproudých vodičů, šlo by to řešit jednodušeji.“ Takto si určitě povzdechla velká část z vás a měli jste částečně pravdu. Nebýt proměnlivé délky uliček, tak si celé skladiště můžeme představit jako graf a jednoduše na něj pustit Dijkstrův algoritmus.[37]

[37] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`

Ten nám vyhledá nejkratší cestu od jednoho vrcholu ke všem ostatním v grafu a to s časovou složitostí $\mathcal{O}((M+N)\log N)$, kde N je počet vrcholů (křižovatek) a M je počet hran (uliček mezi nimi).

Pracuje ve zkratce tak, že vždy vezme vrchol s nejmenší vzdáleností, který doposud není označený za finální, označí ho za finální a aktualizuje vzdálenosti ke všem jeho sousedům. Takto zpracuje všechny vrcholy grafu a postupně buduje cesty z nejkratších vzdáleností ke zpracovávaným vrcholům.

Podívejme se na zadání znovu. Vždyť se na grafu zase tolik nezměnilo, nestačilo by jen přidat nějaké hrany nebo upravit Dijkstrův algoritmus? Existují v podstatě dva možné přístupy.

Prvním z nich je si celý graf zdvojit pro lichou a sudou délku cesty. Vždy natáhneme hrany mezi odpovídajícími lichými a sudými vrcholy s tím, že lichým hranám se silnoproudými vodiči nastavíme dvojnásobnou délku. A pak na tento upravený graf pustíme klasický Dijkstrův algoritmus.

Tento postup je lehčí z hlediska toho, že si nemusíme upravovat samotný algoritmus, ale je těžší na přípravu grafu a na vypsání správného výstupu na konci (musíme si více hlídat, po kterých hranách jsme přišli).

Druhým postupem je neměnit si graf, ale upravit Dijkstrův algoritmus tak, aby zpracovával odděleně sudé a liché průchody. Každý vrchol tedy nebude mít jednu vlastnost finality, ale bude mít samostatnou finalitu pro lichý a sudý průchod.

V takovém případě si ale musíme zdvojit haldu vrcholů v algoritmu a při zpracování vlastně každý vrchol projdeme dvakrát. Zastavíme se ve chvíli, kdy dojdeme do cílového vrcholu po sudé i liché hraně. V ukázkovém programu použijeme tuto variantu.

Oběma postupy projdeme maximálně dvojnásobek hran, respektive dvojnásobek vrcholů, než v klasické implementaci Dijkstrova algoritmu, a paměti spotřebujeme také zhruba dvojnásobek. Tato konstanta se nám schová do $\mathcal{O}$, tedy časová složitost je stále $\mathcal{O}((M+N)\log N)$ a paměťová $\mathcal{O}(N)$.

Jirka Setnička

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-4-4.cpp`

24-4-5 Holografické projektory

Převedeme úlohu na obarvení vrcholů grafu… aha, ale to je poměrně známý NP-úplný problém, to by nám orgové neudělali, ne?

Neudělali, alespoň protentokrát ne. Zadání totiž není obecný graf, ale jen speciální druh, takže úloha není NP-úplná. Stačilo položit si oblíbenou otázku: „A nejde to hladově?“ Jde to hladově. Nuže, jak na to?

Na vstupu máme pořadí zobrazených obrazů. Příklad ze zadání `3 1 2 5 4` říká, že o první obraz se stará projektor číslo 3, o druhý obraz projektor číslo 1 atd. Budeme si pro jednoduchost značit dvojici projektor-obraz (kterou můžeme chápat také jako paprsek) jako $x \to y$, kde x je pozice projektoru a y je pozice obrazu.

Algoritmus bude jednoduchý – prvnímu obrazu přiřadíme frekvenci 1. Nejbližšímu dalšímu obrazu, jehož paprsek se nekříží s předchozím, přiřadíme také frekvenci 1. Takto pokračujeme, dokud neprojdeme všechny obrazy.

Pak projdeme znova obrazy, jimž jsme nepřiřadili žádnou frekvenci. Prvnímu z nich dáme frekvenci 2, nejbližšímu dalšímu, jehož paprsek se nekříží s předchozím, dáme také 2, ... a takto procházíme obrazy, dokud máme nějaké bezfrekvenční.

Proč to funguje? Všimneme si, že pokud má nějaký paprsek $x \to y$ frekvenci $f > 1$, tak určitě existuje paprsek $x' \to y'$ s frekvencí $f - 1$, pro který platí, že $y' < y$ (obraz je víc vlevo) a $x' > x$ (projektor je víc vpravo), takže se kříží.

Totéž ale platí pro nový paprsek. Opakováním až do $f = 1$ zjistíme, že pokud existuje paprsek $x_f \to y_f$ s frekvencí $f > 1$, tak existují paprsky $x_1 \to y_1$, $x_2 \to y_2$, ..., $x_f \to y_f$, pro které platí $x_1 > x_2 > x_3 > \cdots > x_f$ a zároveň $y_1 < y_2 < y_3 < \cdots < y_f$, neboli které se všechny navzájem protínají. A na takový chrchel potřebujeme jistě f barev.

Naše metoda přiřazování frekvencí tedy jistě nepřidělí zbytečně mnoho frekvencí... a je zjevné, že se žádné paprsky se stejnou frekvencí neprotínají. Tedy je náš algoritmus správně.

Jaká je jeho složitost? Označme si počet obrazů N. Času na každý průchod spotřebujeme $\mathcal{O}(N)$, průchodů bude $\mathcal{O}(N)$ (všechny paprsky se vzájemně protínají), takže celkem $\mathcal{O}(N^2)$. Paměti potřebujeme $\mathcal{O}(N)$ na uložení vstupu a výstupu.

Tady bychom mohli skončit s argumentem, že průsečíků zadaných paprsků může být také $\mathcal{O}(N^2)$ a všechny je musíme probrat. Ale chyba lávky, jde to zrychlit. Při jednotlivých průchodech se totiž dost flákáme a určitě se nemusíme podívat na každý průsečík zvlášť.

Během prvního průchodu se podíváme na všechny projektory, ale určíme frekvenci jen u některých. Při dalším průchodu musíme zase projít všechny zbylé ve stejném pořadí se stejnou činností. Zkusíme tedy vyřídit všechno potřebné jedním průchodem.

Projdeme obrazy od začátku do konce jen jednou. Budeme si pro každou frekvenci udržovat, na jaké pozici je poslední známý projektor, který tuto frekvenci má. Tento seznam bude jistě setříděný sestupně, rozmyslete si, proč. Díky tomu v něm můžeme vyhledávat půlením intervalu.

Vždy, když budeme zpracovávat další obraz $x \to y$, vyhledáme nejmenší takovou frekvenci f, jejíž poslední projektor $p(f)$ je víc vlevo než x ($p(f) < x$). Tuto frekvenci přiřadíme, upravíme seznam a jdeme na další obraz. Pokud zjistíme, že taková frekvence zatím není přiřazena (vytekli jsme ze seznamu), tak ji přiřadíme a zvětšíme seznam.

Proč to je ekvivalentní algoritmus? Jednoduše provádíme všechny fáze najednou podle původního plánu. Když zrovna přidělujeme frekvenci f, tak si můžeme představit, že jsme skočili zrovna do f-té fáze...

Časová složitost je nyní výrazně lepší. Půlení intervalu nám zabere nejhůř $\mathcal{O}(\log N)$ a provádíme jej N-krát, takže celkem máme $\mathcal{O}(N \log N)$. Paměťově jsme pořád na $\mathcal{O}(N)$ (musíme si uložit celé původní pole). A pak že to nejde rychleji.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-4-5.c`

Jan „Moskyto“ Matějka

24-4-6 Starý kód

Složitě (respektive spíše ošklivě) zapsaný kód je jen hledáním mostů v grafu (viz grafovou kuchařku).[38] Jeho srozumitelnost značně stoupne, pokud zjistíme, co znamená která proměnná.

`MAX_H` Maximální počet hran grafu.
`MAX_V` Maximální počet vrcholů grafu.
`N` Počet vrcholů grafu.
`M` Počet hran grafu.
`h` Hrany grafu (s konci `.x` a `.y`).
`v` Seznam hran vedoucích z daného vrcholu.
`p` Počet hran vedoucích z daného vrcholu.
`f` Fronta vrcholů, které jsme navštívili.
`b` „Byli jsme tu“ – vrcholy, kam jsme se již dostali.

A podrobněji jaká je funkce programu? Na začátku načte hranu grafu do polí `h` a `v`. Pak prochází všechny hrany grafu `for (int k= ...` a u každé otestuje, jestli je sama mostem (tj. jestli se po zbylých hranách dá dojít z vrcholu `h[k].x` do `h[k].y`). To dělá procházením do šířky z vrcholu `h[k].x`. Do fronty zařazuje jen vrcholy, kam jsme se ještě nepodívali. Pokud jsme po vyprázdnění fronty (tj. po průchodu všech vrcholů, kam se z počátečního vrcholu lze dostat po hranách různých od `h[k]`) nenavštívíli `h[k].y`, tj. druhý konec zkoumané hrany, je daná hrana mostem a tedy jí vypíšeme.

Paměťová složitost tohoto kódu je zřejmě $\mathcal{O}(M+N)$, časová $\mathcal{O}(M(M+N))$ (v každé iteraci for-cyklu můžeme projít až všechny hrany).

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

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_H 1000000
#define MAX_V 1001

typedef struct {int x, y;} H;
int N, M;
H h[MAX_H];
int v[MAX_V][MAX_V];
int p[MAX_V];
int f[2*MAX_V];
short b[MAX_V];

int main() {
  // načteme počet vrcholů a hran
  scanf("%d%d", &N, &M);
  if (N>MAX_V || M>MAX_H) {
    printf("Chybny vstup.\n");
    return 1;
  }
  // a následně i jednotlivé hrany
  for (int i=0; i<M; i++) {
    int x, y;
    scanf("%d%d", &x, &y);
    if (x>N || x<1 || y>N || y<1) {
      printf("Chybny vstup.\n");
      return 1;
    }
    h[i] = (H){x, y};
    v[x][p[x]++] = y;
    v[y][p[y]++] = x;
  }
  printf("Vysledny seznam:\n");
  // budeme postupně testovat všechny hrany
  for (int k=0; k<M; k++) {
    int a = 0;
    int z = 0;
    for (int i=1; i<=N; i++)
      b[i] = 0;
    // zatím jsme navštívili jen
    // počáteční vrchol
    b[h[k].x] = 1;
    f[z++] = h[k].x;
```

```
      // dokud není prázdná fronta, tj. je ještě
      // nezpracovaný vrchol...
      while (a<z && b[h[k].y]==0) {
        int q = f[a++];
        // projdeme hrany vedoucí
        // z tohoto vrcholu
        for (int i=0; i<p[q]; i++) {
          if (b[v[q][i]]==0 && !(q==h[k].x
               && v[q][i]==h[k].y)) {
            // a pokud to není testovaná hrana
            // a končí někde, kde jsme nebyli
            f[z++] = v[q][i];
            // přidáme koncový vrchol do fronty
            // na zpracování a označíme si ho
            b[v[q][i]] = 1;
          }
        }
      }
      // pokud jsme na konec této hrany nedošli
      // vypíšeme jí, neboť je to most
      if (b[h[k].y]==0)
        printf("%d %d\n", h[k].x, h[k].y);
    }
    return 0;
  }
```

A jak program zrychlit? Vcelku jednoduše. Stačí si uvědomit, že most není v grafu součástí žádného cyklu. Takže budeme procházet graf do hloubky a pokud narazíme na hranu, která vede do vrcholu, který už jsme navštívíli, jsme našli cyklus. Budeme si pamatovat, jak hluboko sahal (tj. kam až se můžeme dostat cyklem) a při návratu z rekurze víme, že všechny hrany až do této hloubky nejsou mosty. Zbytek vypíšeme. Každou hranu projdeme maximálně dvakrát takže časová i paměťová náročnost tohoto řešení je $\mathcal{O}(M + N)$.

Tímto samozřejmě dostaneme jiné pořadí výstupních hran jež z původního starého kódu. Tato nedokonalost se dá poměrně snadno napravit (aniž by to asymptoticky zpomalovalo program), zkuste si rozmyslet jak.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-4-6.c`

Pavel Čížek

24-4-7 Čtvercové bombardování

Zopár poznámok na úvod

Aj napriek tomu, že úloha je označená ako ťažká, niektorí z vás poslali riešenie, ktoré bolo očividne príliš pomalé, malo extrémne nároky na pamäť alebo dokonca oboje. Úplne najjednoduchšie riešenie úlohy má časovú zložitosť $\mathcal{O}(n)$, kde n je počet budov. Stačí si uložiť všetky body do poľa a vždy, keď príde dotaz, pole prejsť a o každom bode rozhodnúť, či do štvorca spadá. Za toto riešenie ste mohli získať jeden bod.

S jedným bodom sa neuspokojíme

Prvé, čo si možno všimnúť je, že sa vlastne pýtame na niečo, čomu by sa dalo hovoriť *dvojrozmerné intervaly.* Poďme si úlohu kvapku zjednodušiť a pozrieť sa na to, ako by sme riešili jednorozmernú verziu. Dostaneme teda (celočíselné) body $x_1, \ldots, x_n$ na x-ovej osi a chceme vedieť, že koľko ich patrí do nejakého intervalu $[x, x']$.

V tejto chvíli si spomenieme, že sme nedávno (v minulej sérii) čítali kuchárku o intervalových stromoch, a že asi bude stáť za to, pokúsiť sa tieto pozoruhodné štruktúry využiť.

Predtým, než sa pustíme do samotného rozprávania o intervalových stromoch, treba ošetriť ešte jednu nepríjemnosť. Hodilo by sa nám, aby sme nemali dva body, ktoré by mali rovnakú x-ovú alebo y-ovú súradnicu (neskôr si budete môcť rozmyslieť prečo). Označme si body zadané na vstupe $b_1, \ldots, b_n$, kde $b_i = (x_i, y_i)$. Položme $b'_i := nb_i + i$ (zmeníme obe zložky). Je zrejmé, že ak mali dva body b_i a b_j rovnakú x-ovú alebo y-ovú súradnicu, tak potom body b'_i a b'_j ju majú rôznu. Ešte nahliadnime, že ju nebudú mať rovnakú, ak ju mali rôznu. Nech $x_i > x_j$. Potom je určite $nx_i > nx_j$ a keďže $|i - j| < n$, tak aj $nx_i + i > nx_j + j$. Analogicky pre y-ovú súradnicu. Dotaz $[x, x'] \times [y, y']$ musíme však upraviť na $[nx, nx' + n - 1] \times [ny, ny' + n - 1]$. V ďalšom texte predpokladáme body s rôznymi súradnicami.

Vráťme sa teraz k jednorozmernej verzii problému. Na tu nám v skutočnosti bude stačiť obyčajné pole, v ktorom sú body zoradené vzostupne. Pri dotaze typu $[x, x']$ stačí v poli dvakrát binárne vyhľadať. Najprv hodnotu x a potom x'. Potom je už jednoduché zistiť počet bodov, ktoré vyhovujú dotazu. Odpoveď zvádneme v čase $\mathcal{O}(\log n)$ a pole pripravíme na dotazy v čase $\mathcal{O}(n \log n)$.

Ďalšou možnosťou je použiť istý druh intervalových stromov. Intervalový strom pre body $x_i, \ldots, x_j$ (nech sú zoradené vzostupne) definujeme rekurzívne:

- Koreň bude uchovávať informáciu o bodoch $x_i, \ldots, x_j$.
- Ak $i < j$, tak položme $mid = \lfloor (i + j)/2 \rfloor$. Ľavý syn uchová informáciu o bodoch $x_i, \ldots, x_{mid}$ a potom pravý o bodoch $x_{mid+1}, \ldots, x_j$.

Môžete si všimnúť, že každý vrchol v intervalového stromu S vybudovaného nad bodmi $x_1, \ldots, x_n$ reprezentuje nejaký úsek $[x_i, x_j]$ na x-ovej osi, jeho ľavý syn $\mathrm{l}(v)$ reprezentuje interval $[x_i, x_{mid}]$ a pravý syn $\mathrm{p}(v)$ reprezentuje interval $[x_{mid+1}, x_j]$, kde $mid = (i + j)/2$.

Takýto strom bude mať hĺbku $\mathcal{O}(\log n)$ a jeho definícia nám vlastne hovorí, ako ho budeme budovať. Budovanie má časovú zložitosť $\mathcal{O}(n \log n)$, keďže si body potrebujeme vzostupne zoradiť. Pamäťová zložitosť je $\mathcal{O}(n)$.

Čo dotaz $[x, x']$? Chceme vlastne vybrať také vrcholy stromu S, aby spolu reprezentovali interval $[x, x']$ a zároveň chceme, aby týchto vrcholov bolo čo najmenej. Vyhľadáme si v strome x a x'. Budeme vyhľadávať ako v binárnom vyhľadávacom strome až na to, že z vrcholu v sa do $\mathrm{l}(v)$ presunieme ak vyhľadávaná hodnota je menšia alebo rovná x_{mid} a v opačnom prípade do $\mathrm{p}(v)$). Vyhľadávanie skončíme v liste.

Označme v_x a $v_{x'}$ listy, v ktorých skončí vyhľadávanie x a x' a v_p ich najbližšieho spoločného predka. Skontrolujeme, či do hľadaných bodov máme započítať aj bod v v_x a $v_{x'}$. Teraz budeme postupovať z v_x do v_p a vždy, keď do nejakého vrcholu prídeme z ľavého syna, tak do výsledku pridáme interval z pravého syna. Rovnako budeme postupovať z $v_{x'}$ do v_p a ak prídeme do vrcholu z pravého syna, tak pridáme interval z ľavého syna.

Na dotaz vieme teda odpovedať v čase $\mathcal{O}(\log n)$. Neskôr sa presvedčíme, že rovnako rýchlo sme schopný odpovedať aj na dvojrozmerný dotaz.

Dvojrozmerné intervalové stromy

Skúsme teraz zostrojiť štruktúru, v ktorej budeme schopní odpovedať na dvojrozmerný dotaz.

Použijeme intervalový strom z predchádzajúcej časti, aby sme rozdelili jeden dvojrozmerný dotaz na niekoľko jednorozmerných poddotazov. Vybudujeme intervalový strom S, ktorý ignoruje y-ové súradnice bodov. V každom vrchole v intervalového stromu, ktorý reprezentuje interval $[a_v, b_v]$ na x-ovej, vybudujeme druhoúrovňovú štruktúru $S_y(v)$, ktorá obsahuje všetky body v intervale $[a_v, b_v]$. Každý vrchol v stromu S nám teda reprezentuje nejaký vertikálny pásik v rovine a $S_y(v)$ uchováva body v ňom. Štruktúra $S_y(v)$ môže byť opäť intervalový strom, ktorý tentoraz ignoruje x-ové súradnice. Môže to byť ale aj pole bodov v príslušnom vertikálnom pásiku, zoradených vzostupne podľa y-ovej súradnice. Prvá varianta sa ľahšie zovšeobecní do viacerých dimenzií, my ale pre jednoduchosť budeme uvažovať, že $S_y(v)$ je pole.

Môžeme si všimnúť, že pre každý vrchol v stromu S platí, že body uložené v $S_y(v)$ sú presne body uložené v $S_y(\mathrm{l}(v))$ a $S_y(\mathrm{p}(v))$. Preto ak $S_y(\mathrm{l}(v))$ a $S_y(\mathrm{p}(v))$ poznáme, tak $S_y(v)$ vybudujeme jednoducho zliatím $S_y(\mathrm{l}(v))$ a $S_y(\mathrm{p}(v))$. Celé budovanie si môžeme predstaviť ako merge sort, s rozdielom, že doposiaľ zoradené polia si ukladáme v jednotlivých vrcholoch S a nakoniec v koreni k dostaneme vý-

sledné vzostupne zoradené pole. A síce, pole $S_y(k)$. Vďaka tomu, že máme rôzne súradnice, môžeme body do druhoúrovňovej štruktúry rozmiestniť rovnomerne.

Je vidieť, že budovanie má časovú zložitosť $\mathcal{O}(n \log n)$, rovnako ako merge sort. Hĺbka stromu je $\mathcal{O}(\log n)$. Na každej hladine si v druhoúrovňových štruktúrach pamätáme spolu $\mathcal{O}(n)$ bodov. Pamäťová zložitosť je teda $\mathcal{O}(n \log n)$.

Na dotaz typu $[x, x'] \times [y, y']$ môžeme odpovedať tak, že sa najprv stromu S spýtame na $[x, x']$, čím vymedzíme $\mathcal{O}(\log n)$ vrcholov S, ktoré spolu tvoria horizontálny interval $[x, x']$. Pre každý takýto vrchol v dvakrát vyhľadáme v poli $S_y(v)$. Najprv hodnotu y, potom y'. Jednoducho spočítame, koľko bodov sa nachádza v $[a_v, b_v] \times [y, y']$, a keďže to spočítame pre každý vrchol v, ktorý sme vymedzili, tak dostaneme počet bodov v $[x, x'] \times [y, y']$.

Pri dotaze $\mathcal{O}(\log n)$-krát binárne vyhľadáme. Časová zložitosť je $\mathcal{O}(\log^2 n)$. Za riešenie, ktoré malo rovnakú časovú zložitosť, ste mohli získať plný počet bodov.

Na záver

Počas výkladu riešenia sme nikde nevyužili toho, že dotaz, ktorý príde na vstupe je štvorcový. Sme teda schopní odpovedať aj na ľubovoľný obdĺžnikový dotaz v rovine.

Peter „pizet“ Zeman

Fractional cascading

Existuje moc pěkný trik (říká se mu *fractional cascading*), kterým se dá časová složitost dotazu v dvojrozměrném intervalovém stromu snížit na $\mathcal{O}(\log n)$.

Zopakujme si, jak se vyhodnocuje obdélníkový dotaz: postupujeme stromem shora dolů a podle x-ových souřadnic se rozhodujeme, zda máme jít doleva nebo doprava. V každém vrcholu, který navštívíme, je přitom uložen seznam, v němž potřebujeme vyhledat minimální a maximální y-ovou souřadnici našeho obdélníku. Jelikož seznam je setříděný, můžeme hledat binárně v čase $\mathcal{O}(\log n)$. To provedeme $\mathcal{O}(\log n)$-krát.

Hledání v setříděném seznamu obecně zrychlit nemůžeme, ale pomůže nám, když si uvědomíme, že seznamy, v nichž hledáme, nejsou nezávislé. Pokaždé, když se přesuneme do nějakého syna, najdeme v něm totiž podseznam seznamu uloženého v otci.

Podívejme se na to tedy obecněji: Máme nějaký seznam $A = a_1, \ldots, a_n$ a víme, kde se v něm nachází číslo x – buďto je rovno nějakému a_i, nebo leží mezi a_i a a_{i+1}. Nyní chceme totéž x najít v jeho podseznamu $B = b_1, \ldots, b_m$.

K tomu nám pomůže, když si pro každé a_i předpočítáme, kde se nachází v seznamu B. A pokud se tam nenachází, zapamatujeme si nejbližší větší prvek seznamu B. Kdyby neexistoval (a_i by bylo větší než všechna b_j), ukážeme za

konec seznamu B. Řečeno formálně: $f_i := \min\{j \mid b_j \geq a_i\}$, přičemž dodefinujeme $b_{m+1} := +\infty$, aby minimum vždy existovalo.

Pokud tedy pro hledané x platí $a_i \leq x < a_{i+1}$, najdeme ho v seznamu B mezi pozicemi $f_{i+1} - 1$ a f_{i+1}.

Vraťme se k intervalovému stromu. Do každého jeho vrcholu přidáme pomocné ukazatele, které seznam v tomto vrcholu propojí se seznamy v obou synech. V kořeni tedy budeme stále muset použít binární vyhledávání, ale pokaždé, když se přesuneme do syna, přepočítáme pouze začátek a konec intervalu podle uložených ukazatelů, což stihneme v konstantním čase. Celkem nás tedy celé hledání stojí $\mathcal{O}(\log n)$ v kořeni a $\mathcal{O}(1)$ v $\mathcal{O}(\log n)$ dalších vrcholech, což dohromady dává $\mathcal{O}(\log n)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-4-7.c`

Martin „Medvěd“ Mareš

24-4-8 O hrách a číslech

Úkol 1: Hromádka n sirek

Úkol vyřešíme takto: nejdříve přiřadíme číslo hrám pro malá n, poté si ukážeme, že pro každé n je hra číslo, a nakonec odvodíme vzorec, jak určit toto číslo.

- $n = 0$ – nikdo nemá tah, hra je tedy 0,
- $n = 1$ – levý má tah do 0, pravý táhnout nemůže, což se dá zapsat $\{0 \mid \} = 1$,
- $n = 2$ – levý táhne do 0, pravý do 1, tedy $\{0 \mid 1\} = 1/2$,
- $n = 3$ – $\{1/2 \mid 1\} = 3/4$,
- $n = 4$ – $\{1/2 \mid 3/4\} = 5/8$,
- $n = 4$ – $\{5/8 \mid 3/4\} = 11/16$,

Dále chceme ověřit, že každá hra je číslo. Nejdříve si všimneme, že hry pro sudá n mají menší čísla než hry s $n+1$ sirkami a naopak hry pro lichá n mají větší čísla než hry pro $n+1$ sirek.

Důkaz provedeme indukcí podle n. Pro prvních pár her platí, že sudé hry mají menší číslo než liché (viz výše). Podívejme se na hru s n sirkami, přičemž předpokládáme, že to máme dokázané pro všechny menší hry.

Je-li n sudé, hraje levý do hry s $n-2$ sirkami a pravý do $n-1$. Z indukčního předpokladu víme, že číslo hry $n-2$ je menší než číslo hry $n-1$, tedy hra s n sirkami je číslo, navíc menší, než má hra s $n-1$ sirkami. Analogicky pro liché n je hra číslo větší než hra pro předchozí sudé $n-1$.

Dalším pozorováním je, že hra s n sirkami má po rozšíření zlomku dvěma stejný jmenovatel jako hra s $n+1$ sirkami. Jelikož jejich čitatel se liší jen o 1, hra

s $n+2$ sirkami tak má ve jmenovateli koeficient o jedna větší a je rovna jejich průměru.

Na čísla her se můžeme dívat jako na posloupnost a_n, kde a_n udává číslo hry s n sirkami. Počátek posloupnosti je $a_0 = 0$, $a_1 = 1$. Platí následující vzorec, který stačil k plnému počtu bodů:

$$a_n = (a_{n-1} + a_{n-2})/2.$$

Nyní můžeme chtít explicitní vzorec pro n-tý člen posloupnosti. Ten vyřešíme pomocí kuchařky o lineárních rekurencích.[39]

Charakteristický polynom posloupnosti $x^2 - x/2 - 1/2 = 0$ má řešení $-1/2$ a 0. Vzorec tedy bude tvaru $a_n = A \cdot 1^n + B \cdot (-1/2)^n$ pro nějaké konstanty A a B. Ty zjistíme dosazením za prvních pár členů posloupnosti, čili vyřešením soustavy rovnic $0 = A + B$ a $1 = A - B/2$.

Z první rovnice vyjde, že $A = -B$, druhou upravíme na $1 = A + A/2$. Dostáváme $A = 2/3$ a $B = -2/3$, vzorec pro n-tý člen tudíž je:

$$a_n = 2/3 - 2/3 \cdot (-1/2)^n$$

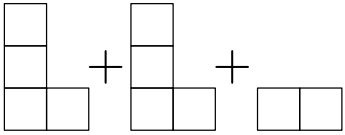
Kdo se setkal s konvergencí posloupností, asi již vidí, že limitou posloupnosti jsou $2/3$.

[Poznámka M.M.: Mimochodem, jde to i bez explicitního vzorce. Zkusme členy posloupnosti zapisovat ve dvojkové soustavě: $a_0 = 0$, $a_1 = 1$, $a_2 = 0.01$, $a_3 = 0.11$, $a_4 = 0.101$, $a_5 = 0.1011$, $a_6 = 0.10101$, … a není těžké ověřit (třeba indukcí), že i další členy mají tento pravidelný tvar. Blíží se proto k $0.\overline{10}$, což jsou desítkově $2/3$.]

Úkol 2: Dominování

Pro pozici v dominování s hodnotou $1/2$, kterou budeme označovat G, chceme dokázat, že $G + G = 1$. Nejsnažším řešení bylo použít poznámku na začátku seriálu: pokud $G - H$ je prohraná hra, pak $G = H$.

Budeme tedy jednoduše zkoumat hru $G + G - 1$, která vypadá takto:



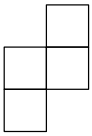
Začne-li levý, položí do jedné z her G své svislé domino buď tak, aby sebral soupeři tah, nebo o políčko výše. Při první možnosti zahraje pravý do druhé hry G a vznikne součet $1 - 1 = 0$. Na tahu je levý, tedy prohrává.

[39] `http://mj.ucw.cz/papers/linrec.pdf`

Druhá možnost (levý položí domino o políčko výše) vede opět na výhru pravého: pravý položí domino do stejné hry jako levý. Ten má v druhé hře G jeden tah, ale pravý má ještě další tah ve hře -1.

Pokud začne pravý, může začít hrát do G nebo -1. Zahraje-li do G, levý potáhne do druhé hry G tak, aby tam pravý neměl tah. Opět dostáváme součet $1-1$, na tahu je však pravý, kvůli čemuž prohraje.

Jestliže pravý položí první domino do hry -1, levý zahraje do G tak, aby tam už nemohl táhnout pravý, jemuž zbude jedna možnost v druhé hře G. Potom však bude mít levý ještě jeden tah, ale pravý ne, takže prohraje.



Druhá část úkolu byla celkem o nápadu, proto jsme nakonec její absenci hodnotili mírně. Řešením je třeba tato pozice H:

Dokážeme jen, že H není číslo, a ověření rovnosti $H+H=1$ necháme jako cvičení (velmi se podobá první části úkolu).

Ve hře H má levý dva tahy, oba vedou do hry 1, pravý může táhnout jen do hry 0. Platí tedy $H=\{1\mid 0\}$, čili H není číslo. (Lze rovněž vyvrátit, že $H=G$, konkrétně přes rozbor hry $H-G$.)

Úkol 3: Padající domino

Úkol byl cvičením na vyškrtáváním tahů, bylo však třeba dávat pozor a ověřovat nerovnosti: např. mezi možnostmi levého můžeme vyškrtnout pozici A jen, pokud může levý zahrát do B a $A\le B$. Jak ověřovat nerovností je popsáno na začátku seriálu.

Nejprve přiřadíme čísla několika jednodušším pozicím:

- všechna domina popadala – číslo 0 (nikdo nemá tah),
- v pozici B má levý dvě možností a pravý žádnou, takže je to 1, pozice BB je 2, BBB je 3... Podobně například $CCCC$ je -4,
- v BC mají oba hráči tah do 0, takže je to *,
- v BBC má levý tah do 0 a 1 (0 je pro levého horší než 1, můžeme ji vyškrtnout), a pravý do 0, jde tedy o $\{1\mid 0\}$. $BBBC$ je z podobného důvodu $\{2\mid 0\}$ a platí $BBBC>BBC$, což se ověří prozkoumáním hry $BBBC-(BBC)=BBBC+CCB$ (je-li to hra levého, nerovnost platí).

Také si všimneme, že otočené pozice dostanou stejná čísla ($BBBBC$ i $CBBBB$ jsou $\{3\mid 0\}$).

Nyní se podívejme na pozici $BCBBBBC$. Levý má možnost hrát do následujících pozic:

- 0 (všechna domina shozena, nikdo nemá tah),
- $BBBC=\{2\mid 0\}$,

- BBC, BC, C jsou všechny horší než $BBBC$ (lze dokázat, že $BBC < BBBC$), takže je můžeme zapomenout,
- $CBBBBC$ – v této pozici má levý tah do $BBBC$ (ostatní možnosti jsou pro něj horší) a pravý do 0 a $BBBBC$ (0 a $BBBBC$ jsou neporovnatelné, neboť $BBBBC + 0$ je hra vyhraná pro začínajícího). Platí tedy: $CBBBBC = \{\{2 \mid 0\} \mid 0, \{3 \mid 0\}\}$. Opět lze dokázat, že tato pozice je horší než $BBBC$,
- BCB, $BCBB$, $BCBBB$ – z těchto her má cenu uvažovat jen $BCBBB$ (ostatní jsou menší, důkaz ponecháme jako cvičení). $BCBBB = \{2 \mid 1\}$, což je větší než $\{2 \mid 0\}$ a 0, takže $BBBC$ a 0 můžeme vyškrtnout.

Pravý může táhnout do pozic:

- všechna domina shozena, tedy 0,
- $B = 1$, ale $1 > 0$, takže tuhle možnost můžeme zapomenout,
- $BBBBC = \{3 \mid 0\}$,
- $BCBBBB = \{3 \mid 1\}$, ale to je větší než $BBBBC$.

Celkově tedy $BCBBBBC = \{\{2 \mid 1\} \mid 0, \{3 \mid 0\}\}$. (To odpovídá intuitivnímu odhadu, že levý zahraje do $BCBBB$.) Možnost $\{3 \mid 0\}$ pro pravého můžeme sice intuitivně vyškrtnout, ale formálně na to nemáme nástroj (hry 0 a $\{3 \mid 0\}$ jsou neporovnatelné).

Podobně, ale stručněji rozebereme hru $BBCCBC$. Levý má tahy do:

- $B = 1$ (což je větší než 0 či $C = -1$, které můžeme vyškrtnout),
- $BBCC = \pm 1$ (což je neporovnatelné s 1)
- $CCBC$ je zjevně horší než 0,
- $BCCBC$ – jelikož hra $BCCBC - B$ je vyhraná pro pravého, platí $B > BCCBC$ a možnost $BCCBC$ není třeba uvádět.

Pravý má možnost táhnout do následujících pozic:

- $CBC = -1/2$
- všechna domina shozena, tedy 0, ale $0 > -1/2$,
- $BC = * > -1/2$,
- $BBC = \{1 \mid 0\} > *$,
- $BB = 2 > -1/2$,
- $BBCCB > -1/2$.

Tedy platí, že $BBCCBC = \{1, \pm 1 \mid -1/2\}$. Možnost ± 1 můžeme intuitivně vyškrtnout, i když je neporovnatelná s 1.

Pavel „Paulie“ Veselý

24-5-1 Holubí centrála

Většina z vás volila jednoduchý algoritmus, který spočíval v procházení grafu do hloubky nebo do šířky od každého z vrcholů grafu. Takové řešení je sice správné, ale jeho kvadratická složitost je příliš veliká. Ukážeme si řešení s lineární složitostí, a to hned dvě. Jedno přímočaré a druhé o kapku složitější.

Řešení prohledáváním do hloubky

Základem je klasické prohledávání do hloubky (DFS). Z libovolného vrcholu (označme ho v) pustíme DFS. Jestliže tímto průchodem objevíme všechny vrcholy, máme jednoho z hledaných členů. Znovu spustíme DFS od v, tentokrát ale po opačně orientovaných hranách. Dostaneme tak všechna další řešení. Proč všechna?

Předpokládejme pro spor, že jsme takto nenalezli nějakého z členů, který patří k řešení. I pro něj musí platit, že se dokáže dostat ke všem ostatním. Speciálně i k členovi v, ze kterého se poprvé DFS spouštělo. V tom případě ale musel být nalezen průchodem do hloubky po opačně orientovaných hranách z v, čímž dostáváme spor.

Zbývá vyřešit situaci, kdy první DFS nenajde člena vyhlašovatele. Pak zjevně ani žádný další člen objevený tímto DFS nemůže být řešením. Označme si každého z nich jako již navštíveného a spusťme DFS na nějakém dosud nenavštíveném. Takto postupujeme až do chvíle, kdy označíme všechny vrcholy.

Jediným kandidátem je nyní člen, ze kterého jsme DFS spustili naposled. Dalším průchodem tedy zjistíme, zda patří k řešení, a v případě, že ano, najdeme zase průchodem po opačně orientovaných hranách celou množinu řešení.

Řešení pomocí komponent silné souvislosti

Základem bude hledání komponent silné souvislosti grafu. Komponenta silné souvislosti je maximální podgraf orientovaného grafu G takový, že pro každé dva různé vrcholy u a v z tohoto podgrafu existuje cesta jak z u do v, tak z v do u.

Dejme tomu, že jsme získali rozklad grafu na KSS. Následuje několik jednoduchých pozorování, která nám už dají řešení úlohy.

Kondenzace grafu je graf, kde vrcholy odpovídají jednotlivým KSS a orientované hrany mezi nimi vedou právě tehdy, pokud mezi nějakým vrcholem jedné komponenty a nějakým vrcholem druhé vede hrana. Taková kondenzace je nutně acyklickým grafem (zkuste si rozmyslet, co by znamenalo, kdyby v kondenzaci cyklus byl).

Dále pokud existuje nějaký hledaný vrchol v komponentě K, od kterého se lze dostat ke všem ostatním, pak i všechny další vrcholy v K jsou řešením.

Nakonec si uvědomme, že taková komponenta K může být právě jedna a musí mít vstupní stupeň roven 0. Kdyby tomu tak nebylo, nedalo by se dostat do

komponent, ze kterých do K vede hrana. To vyplývá z acykličnosti kondenzace. Pokud by takových komponent bylo více, nedalo by se kvůli nulovému vstupnímu stupni dostat z jedné do druhé.

Aby komponenta K byla řešením, musí z ní vést cesta do všech ostatních komponent. To zjistíme triviálně zavoláním DFS na původní graf od libovolného z vrcholů K. Pokud se počet objevených vrcholů rovná počtu vrcholů grafu, je K řešením úlohy.

Zbývá vyřešit, jak rozklad grafu najít. Na to lze použít například Tarjanův algoritmus. Správnost a průběh Tarjanova algoritmu na tomto omezeném místě nemá smysl rozvádět. Počkejte si třeba na jednu z dalších kuchařek. Pro nás je v tuto chvíli důležité, že nám v čase lineárním v počtu hran a vrcholů najde kýžený rozklad.

Časová i paměťová složitost algoritmu je v obou případech lineární ku počtu hran a vrcholů.

Program (C++) – DFS:
`http://ksp.mff.cuni.cz/viz/24-5-1-dfs.cpp`

Program (C++) – komponenty:
`http://ksp.mff.cuni.cz/viz/24-5-1-komponenty.cpp`

Jan Bok

24-5-2 Labutí broadcasting

Na vstupu máme *zprávu* α v podobě řetězce K bitů. Chceme jí přiřadit nějaký *kód,* což bude opět řetězec bitů (jeho délku označíme N) tak, abychom po přijetí libovolné rotace kódu uměli zjistit původní zprávu. Také si to můžeme představovat tak, že kódy jsou *cyklické* posloupnosti a nevíme, kde mají začátek.

Nejjednodušší řešení

Vytvoříme kód tvaru $\mathbf{01}^{K+1}\mathbf{0}\alpha$ – jinými slovy předřadíme zprávě nulový bit, pak $K+1$ jedničkových bitů a opět jeden nulový. Pokud kód čteme cyklicky, narazíme na jednu jedinou $(K+1)$-tici jedniček a ta nám řekne, odkud číst zprávu. Kód měří $N = 2K+2$ bitů (první nulový bit by dokonce šel vynechat, ale tím si moc nepomůžeme). Kódování i dekódování jistě zvládneme v lineárním čase.

Blokový kód

Úspornější způsob kódování spočívá v rozdělení zprávy na bloky velikosti b (konkrétní hodnotu zvolíme vzápětí). Za každý blok připíšeme nulu, čímž zařídíme, že se ve zprávě nevyskytuje víc než b jedniček za sebou. Stačí tedy na začátek přidat synchronizační značku $\mathbf{1}^{b+1}\mathbf{0}$ a jsme hotovi.

Kolik jsme potřebovali bitů? Značka je dlouhá $b+2$, za ní následuje $\lceil K/b \rceil$ bloků délky $b+1$. Celkově tedy $N = b+2+\lceil K/b \rceil \cdot (b+1) \leq b+2+((K/b)+1)(b+1) \leq K+K/b+2b+3$.

Teď využijeme, že jsme mohli b zvolit libovolně, a vybereme si takovou hodnotu, aby N vyšlo co nejmenší. Jelikož s rostoucím b výraz $2b$ roste, zatímco K/b klesá, jejich součet bude (asymptoticky) nejmenší, když se vyrovnají. To odpovídá volbě $b = \lceil \sqrt{K} \rceil$, což vede na $N \leq K + K/\lceil \sqrt{K} \rceil + 2\lceil \sqrt{K} \rceil + 3 \leq K + K/\sqrt{K} + 2\sqrt{K} + 5 = K + 3\sqrt{K} + 5$. Celkem tedy přidáváme $\mathcal{O}(\sqrt{K})$ bitů; kódování i dekódování opět stihneme v lineárním čase.

Abstraktní pohled

Je naše blokové řešení optimální, nebo si lze vystačit s řádově menším počtem bitů? Abychom na tuto otázku odpověděli, zkusme se na úlohu podívat trochu abstraktněji.

Existuje 2^K možných zpráv a každé z nich chceme přiřadit jeden z 2^N možných kódů. Pokud se nějaké dva kódy liší pouze rotací, můžeme použít nejvýše jeden z nich (takovým kódům budeme říkat *ekvivalentní*). Rozdělíme tedy množinu kódů na skupiny tak, že uvnitř každé skupiny budou všechny kódy ekvivalentní a kódy z různých skupin nikdy ekvivalentní nebudou. Každé zprávě pak přiřadíme jednu ze skupin.

(Maličko podvádíme: předpokládáme, že všem 2^K zprávám přiřazujeme kódy téže délky. Nemohlo by pomoci, kdybychom uvažovali i kratší kódy? Asymptoticky nikoliv, protože jak za chvíli uvidíme, počty skupin rostou s N exponenciálně, takže všech kódů délky menší než N je asymptoticky stejně jako kódů délky přesně N.)

Hrubá síla

Potřebujeme zvolit co nejmenší N, pro které už bude skupin dostatečný počet. Označíme-li počet skupin $s(N)$, musí platit $s(N) \geq 2^K$. Jak ale $s(N)$ spočítat?

Pro $N = 4$ je snadné skupiny sestrojit ručně:

0000	**0001**	**0011**	**0101**	**0111**	**1111**
	0010	**0110**	**1010**	**1110**	
	0100	**1100**		**1101**	
	1000	**1001**		**1011**	

Pro trochu větší N si můžeme napsat jednoduchý program, který bude generovat všechny kódy a zařazovat je do skupin. Tak vznikla následující tabulka:

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$s(N)$	2	3	4	6	8	14	20	36	60	108	188	352	632	1182

Podle této tabulky můžeme snadno zjistit, že pro $K = 8$ (což jsme zadávali jako jednodušší podúlohu) je potřeba $N = 12$ labutí. $2^8 = 256$ totiž leží mezi $s(11)$ a $s(12)$.

Pro výrazně větší K ale tento postup není použitelný, neboť vyžaduje probrat a zařadit do skupin exponenciálně mnoho kódů.

Věštíme z tabulky

Jednoduchý vzorec pro $s(N)$ našim snahám zatím uniká, tak zkusme podle hodnot v tabulce odhadnout, jak rychle $s(N)$ přibližně roste. Trocha experimentování odhalí, že $s(N) \approx 2^N/N$. Kdyby to byla pravda, plynulo by z toho, že dokážeme zakódovat až $\lg s(N) \approx N - \lg N$ zpráv (kde lg značí dvojkový logaritmus). Měli bychom si tedy vystačit s přidáním řádově $\lg K$ bitů, což je mnohem méně než našich $\sqrt{K}$.

Zatím ovšem jenom hádáme. Časem dokážeme, že náš odhad počtu skupin je řádově správný; pokud jste netrpěliví, můžete mezitím zkusit najít čísla z tabulky v Online Encyclopedia of Integer Sequences.[40] Zde nejprve předvedeme kódování, kterému řádově logaritmický počet bitů stačí.

Skoro optimální kódy

Myšlenku kódování pomocí bloků lze vylepšit. Pokud by se nám podařilo zařídit, že žádný blok nebude tvořen samými jedničkami, nepotřebujeme bloky prostrkávat nulami. Z neexistence jedničkového bloku totiž plyne, že se ve zprávě nikdy nevyskytne více než $2b - 2$ po sobě jdoucích jedniček. Postačí tedy synchronizační značka s $2b - 1$ jedničkami.

Jenže jak se vyhnout jedničkovým blokům? Snadno: zprávu budeme považovat za zápis čísla ve dvojkové soustavě a toto číslo převedeme do soustavy o základu $2^b - 1$. Každou číslici pak zapíšeme dvojkově do jednoho bloku. Převodem mezi soustavami si sice pokazíme časovou složitost, ale stále zůstane polynomiální (zkuste vymyslet, jak převod zvládnout v čase $\mathcal{O}(K^2)$).

Kolik takto vytvoříme bloků? Pokud nějaké číslo x zapisujeme v soustavě o základu z, potřebujeme nejvýše $1 + \log_z x = 1 + \lg x / \lg z$ číslic. Naše číslo není větší než 2^K, takže počet bloků nepřekročí $1 + \lg(2^K)/\lg(2^b - 1) = 1 + K/\lg(2^b - 1)$.

Každý blok přitom zabere b bitů a navíc přidáme synchronizační značku délky $2b$. Vytvoříme tedy kód délky

$$N \leq 3b + \frac{bK}{\lg(2^b - 1)}.$$

Pěkně to vyjde, pokud K je mocnina dvojky. Tehdy nastavíme $b = \lg K$ a za chvíli ukážeme, že kód si opravdu vystačí s logaritmickým počtem přidaných

[40] `http://oeis.org/`

bitů. Dosazením do předchozí nerovnosti získáme:

$$N \leq 3 \lg K + \frac{\lg K \cdot K}{\lg(2^{\lg K} - 1)}.$$

Jmenovatele zjednodušíme a všechny malé členy posbíráme do $\mathcal{O}$, čímž dostaneme:

$$N \leq \underbrace{\frac{K \lg K}{\lg(K-1)}}_{Z} + \mathcal{O}(\lg K).$$

Zlomek Z je očividně „něco málo přes K“. O kolik přesně? Počítejme:

$$Z - K = \frac{K \lg K - K \lg(K-1)}{\lg(K-1)} = \frac{K \cdot (\lg K - \lg(K-1))}{\lg(K-1)}.$$

Nyní se nám bude hodit jedna ne úplně standardní nerovnost pro logaritmy:

$$\lg n - \lg(n-1) < 2/n. \qquad (*)$$

Když ji dosadíme do předchozího výpočtu $Z - K$, dostaneme:

$$Z - K \leq \frac{K \cdot 2/K}{\lg(K-1)} = \frac{2}{\lg(K-1)} \leq 2. \qquad (\text{pro } K \geq 3)$$

Potvrdilo se tedy podezření, že Z není o mnoho větší než K, což nyní dosadíme do nerovnosti pro N a získáme kýžené:

$$N \leq K + 2 + \mathcal{O}(\lg K) = K + \mathcal{O}(\lg K),$$

takže náš kód je až na konstantu skrytou v $\mathcal{O}$-čku optimální. (Tedy aspoň pro K, které je mocninou dvojky. Zkuste vymyslet, jak kód upravit, aby tento předpoklad nepotřeboval. Nápověda: každé přirozené číslo lze rozložit na součet navzájem různých mocnin dvojky.)

Nerovnost s logaritmy

Ještě si dlužíme důkaz nerovnosti $(*)$. S dovolením budeme předpokládat, že n je sudé číslo; pro liché bychom postupovali obdobně.

Označíme $\ell_i = \lg(n-i+1) - \lg(n-i)$ a uvážíme součet $S = \ell_1 + \ell_2 + \ell_3 + \ldots + \ell_{n/2}$. Všimneme si, že:

- V součtu S se sousední členy vyruší: $S = \lg n - \lg(n-1) + \lg(n-1) - \lg(n-2) + \lg(n-2) - \lg(n-3) + \ldots - \ldots - \lg(n/2) = \lg n - \lg(n/2) = 1$. (Takovým sumám se říká teleskopické podle starodávných dalekohledů, jejichž části se do sebe podobným způsobem zasouvaly.)
- Posloupnost $\ell_1, \ell_2, \ell_3, \ldots, \ell_{n/2}$ je neklesající. Vskutku: pro každé t platí $\lg t - \lg(t-1) \leq \lg(t-1) - \lg(t-2)$. Rozdíl logaritmů totiž můžeme napsat jako logaritmus podílu:

$$\lg \frac{t}{t-1} \leq \lg \frac{t-1}{t-2},$$

což odlogaritmujeme:

$$\frac{t}{t-1} \leq \frac{t-1}{t-2}.$$

Vynásobením součinem jmenovatelů (ten je pro zajímavá t nezáporný) dostaneme:

$$t \cdot (t-2) \leq (t-1) \cdot (t-1),$$

čili

$$t^2 - 2t \leq t^2 - 2t + 1,$$

a to je pravda pro všechna t.

- V každé posloupnosti reálných čísel je minimum menší nebo rovno aritmetickému průměru. Zde je minimem ℓ_1 a průměrem $S/(n/2) = 2/n$, jinak řečeno

$$\lg n - \lg(n-1) \leq 2/n,$$

což je přesně nerovnost, kterou jsme chtěli dokázat.

(Podobným trikem by šla dokázat i nerovnost v opačném směru, totiž $\lg n - \lg(n-1) \geq 1/n$. Zkuste vymyslet, jak.)

Počítáme skupiny

Abychom uspokojili hloubavou mysl, vraťme se ještě k počtu skupin $s(N)$, který jsme zatím pouze „vyvěštili“.

Uvažme nějaký kód délky N a počítejme, jak velká je skupina, do které patří. Kdyby byly všechny skupiny stejně velké, stačilo by vydělit počet kódů velikostí skupiny. Jenže už v našem příkladu pro $N = 4$ narazíme: existují skupiny velikostí 1, 2 i 4.

Zkusme přijít na to, jak pro daný kód α zjistit, jak velká je jeho skupina. Ta je tvořena všemi rotacemi řetězce α. Pokud jsou všechny rotace různé, skupina obsahuje N kódů. V opačném případě je kód α roven nějaké své rotaci. Takové řetězce musí být nutně periodické (tzn. jsou tvořeny opakováním nějakého kratšího řetězce; rozmyslete si, proč).

Toto pozorování nám pomůže spočítat $s(N)$ pro prvočíselné N. Délka periody periodického řetězce totiž musí být dělitelem jeho délky, takže pokud je N prvočíslo, jediné periodické řetězce jsou $\mathbf{0}\ldots\mathbf{0}$ a $\mathbf{1}\ldots\mathbf{1}$. Dvě ze skupin proto mají velikost 1 a ostatní velikost N. Celkem se v nich nachází 2^N kódů, tudíž musí platit:

$$s(N) = (2^N - 2)/N + 2.$$

Naše hypotéza se tedy aspoň pro prvočíselná N potvrdila.

(Vrtá vám hlavou, proč je $2^N - 2$ dělitelné číslem N? To plyne z Malé Fermatovy věty a drobným rozšířením našich úvah o řetězcích bychom dokonce získali její kombinatorický důkaz.)

Pokud je N složené číslo, je situace daleko komplikovanější a neumíme ji vyřešit bez použití trochu pokročilejší kombinatoriky (ale moc pěkné, zkuste si najít tak řečené Burnsidovo lemma). Prozradíme alespoň, co vyjde:

$$s(N) = \frac{1}{N} \cdot \sum_{d \setminus N} \varphi(d) \cdot 2^{N/d}.$$

Suma běží přes všechny dělitele čísla N, $\varphi(d)$ je Eulerova funkce, která udává počet čísel od 1 do $d-1$ nesoudělných s d. Výsledné $s(N)$ opět řádově nepřekročí $2^N/N$.

Závěrem

Po troše počítání jsme našli řešení, které přidává pouze řádově logaritmický počet bitů, a to je až na konstantu optimální.

Kdyby nám na konstantách záleželo, problém by byl mnohem obtížnější. V zásadě bychom potřebovali vybrat si nějaké reprezentanty skupin. Vhodnými kandidáty jsou jejich lexikograficky nejmenší prvky – těm se říká *náhrdelníky* a v naší tabulce pro $N = 4$ leží na prvním řádku. Množinu všech náhrdelníků bychom pak taktéž uspořádali (třeba zase lexikograficky) a chtěli bychom v ní umět najít i-tý nejmenší náhrdelník, aniž bychom všechny náhrdelníky vyjmenovali. Ačkoliv podobné algoritmy jsou známé třeba pro permutace, nalezení i-tého nejmenšího náhrdelníku v polynomiálním čase je stále otevřený problém.

Pokud vás teorie náhrdelníků zaujala stejně jako mne, doporučuji začíst se do knížky Combinatorial Generation od Franka Ruskeyho (dostupná i online).

Program (C) – generátor skupin:
`http://ksp.mff.cuni.cz/viz/24-5-2-generator.c`

Program (C) – kodér:
`http://ksp.mff.cuni.cz/viz/24-5-2-koder.c`

Program (C) – dekodér:
`http://ksp.mff.cuni.cz/viz/24-5-2-dekoder.c`

Martin „Medvěd“ Mareš

24-5-3 Struktura organizace

Obecné řešení

Aby skupina mohla komunikovat, musí existovat nějaký šéf, který má (nepřímo) podřízené všechny zaměstnance. To platí i pro podskupinku zaměstnanců, kterou pošleme do akce.

Pro jednoduchost tedy zvolme šéfa skupinky (S), která jde do akce. Každý z jeho přímých podřízených (P) buď do akce jít nemusí, nebo může. V prvním případě to odpovídá jedné variantě (pokud tam nejde P, nemůže jít ani libovolný z jeho (nepřímých) podřízených, jelikož by nebyl schopen předat zprávu např. S).

V druhém případě lze vzít do akce i nějaké podřízené P. Počet možností, kolika způsoby je lze vybrat, je přesně počet možností, kolika můžeme vybrat akční skupinu, kde šéf bude P.

Celkový počet možností, kolika vybrat podřízené S, spočteme uvážíme-li, že výběr je nezávislý pro každého podřízeného P, tedy

$$\text{Možnosti}(S) = \prod_{P \in \text{podřízení } S} (1 + \text{Možnosti}(P)).$$

Pokud chceme zjistit počet skupin s libovolným šéfem, stačí sečíst počty možností přes všechny vedoucí, tedy

$$\text{AkčníchSkupin} = \sum_{S \in \text{všichni}} \text{Možnosti}(S).$$

Tahle myšlenka se v programu implementuje přímočaře, časová složitost bude $\mathcal{O}(NM)$, kde N je počet zaměstnanců a M čas potřebný na jednu aritmetickou operaci. Aritmetických operací vskutku provedeme $\mathcal{O}(N)$, jelikož operace uvnitř $\sum$ nebo $\prod$ můžeme „naúčtovat“ podřízeným, kteří se jich účastní, a každému podřízenému takto naúčtujeme nejvýše konstantní počet operací.

Obrovská čísla a Karacubův algoritmus

Proč je ale ve složitosti zmiňováno M? Bohužel počet možností, kolika lze poskládat skupinku, roste rychle. Horní odhad je počet možností, jak vybrat libovolnou skupinku zaměstnanců (2^N), a není příliš nadhodnocený. Uvážíme-li např., že šéf S má K přímých podřízených a další zaměstnanci neexistují, bude počet možných skupin vyslaných do akce $2^K + K$, což se od triviálního odhadu (2^{K+1}) příliš neliší.

Potřebujeme tedy počítat s obrovskými čísly, která mají $C = \mathcal{O}(N)$ cifer. Budeme je v paměti reprezentovat jako pole číslic. Sčítání pak lze stihnout v čase $\mathcal{O}(C)$ vcelku triviálně. S násobením je to horší. Ukážeme si zde, jak dvě čísla vynásobit v čase $\mathcal{O}(C^{\log_2 3})$ pomocí Karacubova algoritmu.

To, že čísla ukládáme po cifrách, je podstatné. Existují reprezentace (např. pomocí zbytků), ve kterých jde násobit i sčítat v čase $\mathcal{O}(C)$. Problém pak mají s výpisem výsledku.

Základní myšlenka Karacubova algoritmu je rozděl a panuj.[41] Nechť násobíme čísla A a B o C cifrách. Rozdělme si každé na 2 čísla A_h a A_d o $C/2$ cifrách tak, že bude platit $A = A_h 10^{C/2} + A_d$ (efektivně poloviny čísla dle cifer v desítkovém zápisu), pro B analogicky. Pak platí $A \cdot B = A_h B_h 10^C + (A_h B_d + A_d B_h) 10^{C/2} + A_d B_d$, efektivně tedy potřebujeme kromě sčítání 4 násobení (vynecháme-li násobení mocninami desítky, což je však v zápisu po cifrách jednoduchý posun).

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

Podívejme se na násobení pořádně. S A_hB_h a A_dB_d moc neprovedeme, s $A_hB_d + A_dB_h$ se však dá ještě pracovat. Konkrétně lze snadno nahlédnout, že

$$A_hB_d + A_dB_h = (A_h + A_d)(B_h + B_d) - A_hB_h - A_dB_d.$$

Hle – poslední dva součiny už známe. Spočteme tedy součiny Malý $= A_dB_d$, Střední $= (A_h + A_d)(B_h + B_d)$ a Velký $= A_hB_h$ a pak platí $A \cdot B =$ Malý $+$ (Střední $-$ Malý $-$ Velký) $\cdot\, 10^{C/2} +$ Velký $\cdot\, 10^C$. Tím jsme zredukovali počet potřebných násobení na tři.

Čas potřebný k spočítání součinu čísel o C cifrách, $T(C)$, lze tedy vyjádřit jako $T(C) = 3T(C/2) + fC$, kde f je vhodná konstanta. Čas fC odpovídá času spotřebovanému na sčítání, posouvání apod., vše jde stihnout lineárně či lépe vzhledem k počtu cifer. Vyřešením takovéhle rekurence zjistíme, že násobení spotřebuje čas $\mathcal{O}(C^{\log_2 3}) \approx \mathcal{O}(N^{1{,}585})$.

S tímhle násobením (a uvážením, že $C = \mathcal{O}(N)$) tedy bude program mít časovou složitost $\mathcal{O}(N \cdot N^{\log_2 3}) \approx \mathcal{O}(N^{2{,}585})$ a paměťovou $\mathcal{O}(NH)$, kde H je počet hladin stromu, tj. kolik (nepřímých) šéfů má libovolný zaměstnanec maximálně nad sebou (+1).

Program (Pascal):
`http://ksp.mff.cuni.cz/viz/24-5-3-karacuba.pas`

Binární stromy

V případě, že strom zaměstnanců je úplný binární (což jsme zadávali jako podúlohu), lze algoritmus zjednodušit. Konkrétně víme, že oba podstromy synů budou pro každý vrchol stejné. Tedy i počty možností, kolik skupin můžeme stvořit, se bude shodovat.

Pro šéfa v hloubce h (velký šéf má hloubku 0, jeho přímí podřízení 1, atd.) bude tedy platit

$$\text{Možnosti(hloubka } h) = (1 + \text{Možnosti(hloubka } h+1))^2,$$

samozřejmě že pro listy (zaměstnance bez podřízených) platí Možnosti $= 1$.

Dále v hladině s hloubkou h bude 2^h zaměstnanců. Počet akčních skupin pak bude

$$\text{AkčníchSkupin} = \sum_{h=0}^{H} 2^h \cdot \text{Možnosti(hloubka } h).$$

Tohle lze stihnout spočítat v čase $\mathcal{O}(HM)$, kde H je hloubka stromu ($H = \lceil \log_2(N+1) \rceil$) a M je opět náročnost násobení. Celkově (s Karacubovým algoritmem) bude časová složitost $\mathcal{O}(N^{\log_3 2} \cdot \log_2 N)$.

Rychlejší násobení

Násobení nás evidentně brzdí. Není možné jej stihnout rychleji? Lze ukázat, že Karacubův algoritmus je součástí třídy algoritmů Tooma a Coo-

ka, které spočítají násobení v čase $\mathcal{O}(C^{\log(2k-1)/\log(k)})$, kde k je přirozené číslo. Pro každé $\varepsilon > 0$ tedy můžeme zvolit dost velké k tak, abychom násobili v čase $\mathcal{O}(C^{1+\varepsilon})$; konstanta v $\mathcal{O}$ přitom s klesajícím ε obludně roste.

Naznačme si však, jak pracuje jeden z (asymptoticky) nejrychlejších algoritmů na násobení, Schönhageův-Strassenův algoritmus – počítá součin čísel v čase $\mathcal{O}(C \log C \log\log C)$. Vzhledem k tomu, že však spoléhá na některé složitější výsledky z algebry, nebudu zde jeho správnost dokazovat, čtenář si v případě zájmu jistě vyhledá tento algoritmus detailně sám. (Doporučuji se podívat i na Carmichaelovu funkci, která souvisí s volbou základu p okruhu $\mathbb{Z}$.)

Základní myšlenka je, že vynásobení frekvenčních koeficientů po provedení Fourierovy transformace odpovídá konvoluci v „přímém“ obrazu.

Uvažujme Fourierův obraz daného čísla A. To je nějaký vektor $\mathcal{F}[A]$, pro jehož k-tou složku platí:

$$\mathcal{F}[A]_k = \sum_{j=0}^{C-1} A_j \alpha^{jk}.$$

kde A_j je j-tá cifra A, α C-tá odmocnina z jedničky (primitivní, tj. taková, že $\alpha^k \neq 1$ pro $0 < k < C$), analogicky pro B. Definujeme-li Fourierův obraz D jako součin Fourierových obrazů A a B, tedy

$$\mathcal{F}[D]_k = \mathcal{F}[A]_k \cdot \mathcal{F}[A]_k,$$

pak bude platit

$$D_k = \sum_j A_j B_{k-j},$$

přičemž suma jde přes všechny hodnoty, kde sčítanci mají smysl. Vzhledem k této definici však platí

$$A \cdot B = \sum_k D_k 10^k.$$

Tedy bude stačit jen znormalizovat zpět D_k na cifry (jelikož konvoluce nezaručuje, že vyjdou jednotlivé cifry, ale jen že předchozí suma je rovna součinu).

K implementaci budeme potřebovat znát ještě inverzi Fourierovy transformace, která lze zapsat jako

$$D_k = \frac{1}{C} \sum_{j=0}^{C-1} \mathcal{F}[D]_j \alpha^{-jk},$$

a způsob, jak rychle spočíst Fourierovu transformaci (inverze evidentně, až na normalizaci, je Fourierova transformace s použitím primitivní odmocniny $1/\alpha$) a dále jak najít ono číslo α, aniž bychom ztráceli přesnost.

První problém lze vyřešit použitím algoritmu pro rychlý výpočet Fourierovy transformace, v implementaci je použit Cooleyův-Tukeyův algoritmus, který pracuje v čase $\mathcal{O}(C \log C)$.

Problémům s přesností se nevyhneme, pokud budeme počítat Fourierovu transformaci klasicky, tj. v komplexních číslech. Pomůže ale přesunout se do nějakého konečného okruhu, v našem případě do celých čísel modulo 469 762 049 (kde 33 je primitivní 2^{26}-tá odmocnina z jedničky).

Po použití tohoto algoritmu bude celý program pracovat v čase $\mathcal{O}(C^2 \log C \log\log C)$.

Program (Pascal):
`http://ksp.mff.cuni.cz/viz/24-5-3-s-s.pas`

Pavel Čížek

Medvědí poznámka: Ani Schönhageův-Strassenův algoritmus není posledním slovem v oblasti rychlé aritmetiky. S použitím takřka ďábelských triků se dá násobit i v lineárním čase. O těchto algoritmech a fascinující historii jejich objevování znamenitě vypráví pan Donald Knuth ve svých Seminumerical Algorithms (2. díl jeho veledíla The Art of Computer Programming). Pokud by vás zajímalo, jak se takové věci dělají, rád se nechám přemluvit k půlnoční přednášce na soustředění. A pokud toužíte „jen“ po pochopení Fourierovy transformace, zkuste nahlédnout na stránky přednášky z ADS2.[42]

Martin „Medvěd“ Mareš

24-5-4 Sraz na náměstí

Napřed chvíli předpokládejme, že architekt náměstí byl při smyslech a udělal ho konvexní. Řešení pro nekonvexní náměstí není o moc složitější, ale problém se řeší lépe postupně. Taktéž předpokládejme, že žádné dva vrcholy nemají stejnou y-ovou souřadnici. Programu to nijak nevadí (pokud se vrcholy se stejnou y-ovou souřadnicí prochází např. zleva doprava), ale ve vysvětlování by rušily.

Všimneme si, že alespoň jedna z nejdelších vodorovných úseček bude končit ve vrcholu. Pokud se totiž podíváme na nějakou úsečku, která nekončí ve vrcholu, tak buď není od kraje ke kraji, nebo oběma konci končí na hranách náměstí. Tyto dvě hrany jsou buď rovnoběžné, potom úsečku můžeme posouvat jak nahoru, tak dolů, dokud nenarazíme na vrchol, aniž by se změnila délka. A nebo tyto hrany rovnoběžné nejsou, ale v tom případě se jedním směrem od sebe vzdalují, úsečku tedy můžeme posunout tímto směrem a tím ji prodloužit.

Tedy pro vyřešení problému s konvexním náměstím nám stačí zamést jej přímkou odshora dolů (což bylo ukázáno v geometrické kuchařce v této knize). Budeme si udržovat, která hrana je aktivní na levé a na pravé straně. V každém vrcholu spočítáme délku úsečky od tohoto vrcholu k protější aktivní hraně. Poté vyměníme aktivní hranu na straně, kde se nacházel vrchol.

[42] `http://mj.ucw.cz/vyuka/ads2/`

Nyní, co za problémy nám přinese nekonvexnost náměstí? Prvním je to, že z vrcholu už nemusí vést jedna úsečka nahoru a druhá dolů. Může se nám stát, že obě vedou nahoru nebo obě dolů. A z toho plyne další drobný problém. Úsečka už teď nemusí být v dané výšce jen jedna, ale může jich být několik vedle sebe, oddělených od sebe zuby.

Jak to vyřešíme? Jednotlivé úsečky, co aktuálně existují, si uložíme do vyhledávacího stromu, v pořadí odleva doprava. Jejich konce se sice stále mění, proto nemohou být uložené, ale to nevadí, můžeme uložit hrany, na kterých ty konce leží, a počítat je průběžně dle potřeby. Jejich pořadí zůstane po celý život úsečky stejné.

Když potkáme vrchol, který má jednu svou úsečku nahoru a jednu dolů, najdeme úsečku, která v té horní končí, a hranu v ní nahradíme. Pokud má vrchol obě hrany dolů, pak buď dělí existující úsečku na dvě (pokud se náměstí nachází na vnější straně úhlu), nebo vytváří novou úsečku začínající v tomto vrcholu.

Pokusíme se tedy najít úsečku, která tento bod obsahuje. Pokud existuje, úsečku rozdělíme na dvě. Pokud ne, vytvoříme novou úsečku. Vrchol, kde vedou obě hrany nahoru, funguje obdobně, jen dvě úsečky spojujeme nebo aktuální jednu úsečku uvnitř zubu mažeme.

V každém případě zjistíme délku té úsečky, na kterou jsme sáhli. V případě, že rozdělujeme nebo spojujeme, uvažujeme tu vcelku, neboť je to ta nejdelší, kterou máme k dispozici.

Nyní, k odhadům složitostí. Pokud má náměstí n vrcholů, tak má také tolik hran. Ve stromu budeme mít maximálně tolik úseček, neboť ke vzniku nové úsečky potřebujeme vždy vrchol. Takže paměťová složitost bude lineární. Na začátku potřebujeme vrcholy setřídit a děláme $\mathcal{O}(n)$ operací na stromě, kde každá operace trvá $\mathcal{O}(\log n)$. Dohromady máme tedy časovou složitost $\mathcal{O}(n \cdot \log n)$.

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-5-4.cpp`

Michal „Vorner“ Vaner & Lucka Mohelníková

24-5-5 Řezání kabelů

S úlohou o řezání kabelů se můžete setkat na Medvědových cvičeních z Programování II na Matfyzu jako s řezáním trámů. Možná budu nosit dříví do lesa, než s ním dojdu na pilu, ale rád bych úvodem řekl něco o řešeních, která nikam nevedou. Nebojte, optimální řešení také zmíním a nakonec se dozvíte i pár slov o tom, jak úloha souvisí s kompresí dat.

Zopakujme ve stručnosti zadání: Kabel o délce K máme nařezat na n kusů zadaných délek k_1 až k_n. Můžeme přitom vždy řezat jen jeden kus na dva; takový

řez nám zabere tolik času, jako je délka řezaného kusu. Hledáme postup, kterým nařežeme celý kabel za nejkratší možnou dobu T^*.

Hrubá síla

Hrubá síla dá správný výsledek vždycky. Bohužel, tady je příliš hrubá i pro velmi malé vstupy.

Kusům, jejichž délky jsou na vstupu, říkejme *základní kusy*. Do podoby původního kabelu je za sebe můžeme slepit $n!$ způsoby.[43]

Na kabelu si můžeme udělat rysky, podle kterých budeme řezat a kterých bude pro každý způsob $n-1$. Počet řezů je to jediné, čím si můžeme být docela jisti – někteří z vás viděli spojitost se známou úlohou o lámání čokolády, ale rysky podle mě dávají ještě jednodušší představu; podle každé očividně musíme kabel přeříznout právě jednou.

Různých pořadí výběru rysek je $(n-1)!$. Udržujeme si v paměti dosavadní nejlepší postup řezání a v čase lineárním vzhledem k n s ním porovnáváme každý nově vygenerovaný postup. Celkem tedy hrubá síla trvá $\mathcal{O}(n! \cdot (n-1)! \cdot n) = \mathcal{O}((n!)^2)$. To je mnohem, mnohem, mnohem horší než exponenciální. Paměť $\mathcal{O}(n)$ nás pak ani nebude zajímat a honem poběžíme hledat něco, co má šanci doběhnout před koncem světa. (Takže do zimy.) ;-)

Heuristiky s půlením délek

Nemálo z vás přistupovalo k úloze s dobrou intuicí, že se vyplatí kabel nejprve rozříznout někde „uprostřed“, abyste čas na řezání velkého kusu investovali jednou a řezali pak už jen menší kusy.

Takto vágní popis algoritmu se rozrostl v celou řadu heuristik, bez výjimky chybných. Samotné rozdělení kusů do dvou v součtu stejně dlouhých skupin je problém dvou loupežníků,[44] o kterém jsme loni měli úlohu a který patří mezi těžké problémy.[45] Příklad ze zadání je přímo protipříkladem na heuristiku ze zmiňované loňské úlohy (rozdělování od nejdelších kusů). Nemohu vyloučit, že některý z algoritmů jdoucích tímto směrem bude dost blízko korektnímu řešení, ale vážně o tom pochybuji.

Hladové lepení – optimální algoritmus

Jak mělo vypadat optimální řešení? Mělo se na to jít z opačného konce. Místo abychom kabel řezali, budeme ho z už nařezaných kousků lepit. Potom jenom pustíme záznam postupu pozpátku.

[43] `http://cs.wikipedia.org/wiki/Permutace`
[44] `http://ksp.mff.cuni.cz/viz/23-4-3/reseni`
[45] `http://ksp.mff.cuni.cz/viz/kucharky/tezke-problemy`

Lepit k sobě budeme vždycky dva nejkratší kusy kabelu, které zrovna máme. Opakujeme, dokud nemáme celý kabel. Je to tak prosté, až se nechce věřit, že je to správně. Korektnost postupu si ale hned dokážeme.

Nejprve si všimneme, co se stane, když budeme líní. Líný programátor nevyhodnocuje aritmetické výrazy, jenom k nim připisuje další operace. Pokud si v průběhu lepení poznamenáváme délky *slepených kusů* líně, dojdeme k tomu, že každý *základní kus* (jeden z n kusů na vstupu) přispěje do celkového času tolikrát, kolikrát se účastnil lepení sám nebo jako součást už slepeného kusu.

Už z tohoto pozorování je možné uhodnout, že bude moudré lepit nejdřív dva nejmenší kusy, protože u těch nejméně vadí, že se budou lepení účastnit víckrát. Exaktní důkaz povedeme sporem. Označme si l_i počet lepení, kterých se účastnil základní kus i, čas řezání $T_l = \sum_{i=1}^{n} k_i \cdot l_i$. Optimální čas řezání $T^* = \min_l T_l$. *Optimálním postupem* myslíme postup, který trval dobu T^*.

Dokazovat budeme tvrzení, že dva základní kusy x a y, které se v nějakém optimálním postupu účastnily nejvíce lepení a jako první se slepily spolu ($l_x = l_y = \max_{1 \le i \le n} l_i$), jsou ty dva nejkratší ($k_x = \min_i k_i$; $k_y = \min_{i \neq x} k_i$; tedy $k_x \le k_y \le k_i \forall i$). Kdyby v optimálním postupu x a y nebyly dva nejkratší kusy kabelu, musel by existovat kus $z \neq x$ o délce $k_z < k_y$. Tento kus by se díky volbě l_y účastnil $l_z \le l_y$ lepení. Prohodíme l_y a l_z, jinak postup zachováme.

$$\begin{aligned} l'_y &= l_z, \\ l'_z &= l_y, \\ l'_i &= l_i \forall i \notin \{y, z\} \end{aligned}$$

- Pokud $l_z = l_y$, čas řezání T_l se prohozením nezměnil a postup už vyhovuje tvrzení. Chtěli jsme, aby nějaký takový postup existoval.
- Pokud $l_z < l_y$, prohozením jsme čas T_l snížili, protože $l_z \cdot k_z + l_y \cdot k_y > l'_y \cdot k_y + l'_z \cdot k_z$ a levou dvojici sčítanců jsme v T_l při přechodu k l' vyměnili za pravou. Čas po prohození $T_{l'} < T_l$, ale předpokládali jsme $T_l = T^*$, což je spor s předpokladem. Lepšího než optimálního času řezání dosáhnout nemůžeme, takže původní postup nebyl optimální.

Tvrzení dokázáno a s ním i korektnost algoritmu. Zapomeneme, že jsme x a y slepili, a po nalezení zbytku optimálního postupu si na to zase vzpomeneme; v tom tkví celé kouzlo.

Složitost optimálního řešení

Jakou má náš algoritmus složitost? To záleží, jak chytře budeme průběžně hledat nejkratší kusy. Dobré je vložit všechny délky kusů do haldy a vždycky dva nejkratší slepit a výsledný kus zase vrátit, dokud nebudeme mít celý kabel. Při jednom lepení potřebujeme dva výběry minima z haldy a jedno vložení do haldy. Tyto operace s haldou trvají $\mathcal{O}(\log n)$, protože v haldě máme $\le n$ kusů.

Jak už jsme zjistili dřív, lepení je $n-1$, celkem tedy potřebuje náš algoritmus čas $\mathcal{O}(n \log n)$. Paměti potřebuje $\mathcal{O}(n)$ kvůli haldě.

Pokud bychom chtěli vypisovat na výstup řezy a ne lepení, mohli bychom si lepení ukládat na zásobník a na konci programu je vypsat. Časové ani paměťové nároky algoritmu to nezhorší.

Při implementaci haldy si musíme dát pozor, aby zvládala pracovat s duplicitními klíči. Pokud v haldě máme dvě stejná čísla, je zcela očividně jedno, které z nich vybereme. Kdybychom v jiné úloze měli v haldě složitější objekty, už na volbě záležet může.

Drobné vylepšení

Ještě si můžeme rozmyslet, že haldu programovat nemusíme. Stačí si všimnout, že každý další slepený kus je nejméně tak velký, jako je ten předchozí. Když je budeme dávat do fronty, budeme je z ní vybírat v setříděném pořadí. Algoritmus tedy můžete najít ve vzorové implementaci zhruba takto:

1. Načti počet kusů kabelu `n`, pokud `n` < 2, skonči.
2. Načti délky kusů `k`.
3. Setřiď `k` vzestupně.
4. Vytvoř frontu délek slepených kusů `s`, výstupní zásobník `o`.
5. Vyber do `a`, `b` první dva prvky `k`.
6. Do `o` ulož (`a`, `b`), do `s` přidej `a + b`.
7. Proveď $(n-2)$-krát…
   - Vyber minimum `a` z čel front `k` a `s` a z původního umístění ho odeber.
   - Vyber minimum `b` z čel front `k` a `s` a z původního umístění ho odeber.
   - Do `o` ulož (`a`, `b`), do `s` přidej `a + b`.
8. Každou dvojici z `o` vypiš ve formátu `"(a+b) -> a + b"`.

Můžeme si všimnout, že nejpomalejší na celém postupu je třídění; pokud použijeme některý z rychlých algoritmů,[46] zůstaneme na časové složitosti $\mathcal{O}(n \log n)$. Pokud ale dostaneme vstup už setříděný, můžeme si polepšit – zbytek algoritmu totiž běží v čase $\mathcal{O}(n)$, jelikož během každého lepení děláme jen konstantní počet operací konstantní složitosti.

Dynamika? Pomalá…

Některé z vás mohlo napadnout, že by mohlo existovat řešení založené na dynamickém programování; jedno takové jsem dokonce dostal. Původně jsem nevěřil tomu, že funguje, ale opravdu je to tak. Ovšem je pomalé a těžkopádné proti tomu, které jsem už ukázal. V podstatě se zakládá na přístupu hrubou silou, ale navíc potřebuje důkaz celkem netriviálního tvrzení.

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

To tvrzení říká, že v řešení hrubou silou není potřeba zkoušet všechna různá pořadí nařezaných kusů, protože když najdeme nejkratší postup řezání pro setříděnou permutaci, jde upravit na nejkratší postup pro každou jinou. Setříděnou permutací myslíme takovou, že délky kusů na kabelu zleva doprava rostou. Díky tomuto omezení rozsahu úlohy srazíme složitost hrubé síly na $\mathcal{O}(n!)$, což je jenom mnohem pomalejší než exponenciální – další dvě „mnohem“ už si mohu odpustit. Když navíc přidáme dynamické programování, získáme už polynomiální algoritmus.

Ten zkouší, podle které rysky v pevně daném, setříděném pořadí základních kusů je nejvýhodnější začít řezat. Pro každou z $\mathcal{O}(n^2)$ posloupností po sobě jdoucích základních kusů postupně spočítá minimální čas, za který jde nařezat. Postupuje přitom od těch, které obsahují nejméně kusů, po ty, které jich obsahují nejvíce. Jednokusové posloupnosti jdou nařezat triviálně za nulový čas. Pro delší budeme počítat časy řezání začínajících vybranou ryskou, z nich minimum přes všechny rysky uvnitř této posloupnosti.

Řez podle vybrané rysky rozdělí posloupnost základních kusů na dvě kratší, pro které výsledek už známe. Spočítáme tedy minimální čas řezání posloupnosti, které začíná tímto řezem. Časy řezání kratších posloupností sečteme a přičteme k nim čas potřebný pro jejich oddělení, tedy celkovou délku právě řezané posloupnosti. (Reálnou délku, už ne v počtu kusů!)

Časy si můžeme v průběhu výpočtu uchovávat třeba v tabulce (matici, vícerozměrném poli), kterou budeme indexovat délkou posloupnosti (opět v počtu kusů) a pořadovým číslem rysky, na které začíná. Posloupnost délky n kusů je jen jedna, celý kabel. V jemu odpovídající buňce najdeme na konci výpočtu minimální celkový čas řezání.

Teď ještě najít i konkrétní postup. Je to klasická dynamika… Pro každou posloupnost si budeme pamatovat (v extra tabulce), který řez je pro ni nejvýhodnější provést jako první. To už v průběhu výpočtu zjišťujeme, tak si to teď budeme i pamatovat. Z takové informace už je na konci výpočtu postup řezání celého kabelu triviální sestavit.

Algoritmus získaný metodou dynamického programování potřebuje $\mathcal{O}(n^2)$ paměti kvůli tabulce pro rekonstrukci postupu a $\mathcal{O}(n^3)$ času, protože při výpočtu hodnoty každé z $\mathcal{O}(n^2)$ buněk tabulky spotřebuje čas $\mathcal{O}(n)$ na hledání nejlepší rysky.

Čas třídění délek kusů je asymptoticky menší než $\mathcal{O}(n^3)$, takže složitost nezvýší, buněk tabulky je $\mathcal{O}(n^2)$ proto, že tolik je různých dvojic začátek-konec posloupnosti.

Připomínám, že jsme nedokázali onen netriviální předpoklad, že stačí úlohu vyřešit pro setříděné pořadí základních kusů. Důkaz nechávám jako cvičení pro

pokročilé. Kdybyste ho nemohli vymyslet a moc vás zajímal, zeptejte se mě mailem nebo raději na fóru.

Bodování

Za přehledně popsaný optimální algoritmus včetně výpočtu časové a paměťové složitosti, se zdůvodněním korektnosti (důkaz jsem nechtěl, ten by byl za bonusový bod) bylo možné získat plný počet 9 bodů. Za špatně popsaný optimální algoritmus, u kterého jste se složitostí ani korektností vůbec nezabývali, jste dostávali 6 bodů. Stejný počet bodů jsem měl v plánu dávat i dobře popsaným algoritmům se zdůvodněním korektnosti a výpočtem složitosti, které nejsou optimální, ale pořád běží v polynomiálním čase. Přišel mi ale jen jeden 4bodový s divokým popisem a bez zdůvodnění korektnosti. Ten mi zabral na opravení nejvíc času. Pamatujte, že počet přidělených bodů je nepřímo úměrný času, který opravující org nad řešením stráví. ;-)

Méně než čtyři body dostávala řešení, která nefungovala nebo nebyla polynomiální. Použitelné jsou totiž obě kategorie zhruba stejně. Nulu nedostal nikdo; konkrétní počet bodů jsem uděloval podle přítomnosti užitečných pozorování o úloze, přehlednosti vyjadřování (ani nad nefunkčním řešením nechci strávit odpoledne) a za snahu.

Souvislost s kompresí dat

Na konec slibovaná perlička ohledně komprese dat. Jak spousta z vás postřehla, na postup řezání je možné se dívat jako na binární strom. Základní kusy tvoří listy, slepené kusy tvoří vnitřní vrcholy, celý kabel je kořen. Zároveň každý vnitřní vrchol má právě dva syny a představuje jedno řezání.

Zapomeňme na to, že šlo o kabely, k základním kusům připišme písmena abecedy a na délky kabelů se koukejme jako na četnosti písmen v nějakém textu. Na hrany směřující doleva napišme nuly, na hrany směřující doprava jedničky a už po cestě z kořene do listu můžeme číst kód, kterým budeme znak zapisovat.

Díky tomu, že písmena jsou jenom v listech, není žádný kód prefixem (předponou) jiného, takže text zapsaný pomocí takto zakódovaných písmen je jednoznačně dekódovatelný.

Když se znovu podíváme, co je vlastně čas řezání, zjistíme, že je to vážený součet délek kódů, kde váhy jsou četnosti znaků. To je ale přeci celková délka zakódovaného textu! Jak jsme o kousek výš dokázali, menší už být nemůže...

Tomuto optimálnímu prefixovému kódu se říká Huffmanovo kódování.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-5-5.c`

Tomáš „Palec“ Maleček

24-5-6 Minové pole

Je zřejmé, že minové pole samotné má velikost $\mathcal{O}(MN)$ a celé je musíme vypsat, budeme se tedy snažit o právě takovou složitost.

Nejprve jednorozměrná varianta: tam obdélníky popisující dosah min (dále jen obdélníky) jsou vlastně úsečky, a tak nás jen zajímá, kde na řádce začínají a kde končí.

Uvědomíme si, že nás často budou zajímat hranice, a tak si vytvoříme pole o délce $n+1$, které místo políček v matici obsahuje informace o hranách čtverečků. To nám pomůže vyřešit případy 1×1.

Toto pole budeme chtít projít právě jednou, a tak si do něj uložíme levé a pravé hranice obdélníků na vstupu. Na levou hranici uložíme `+1` a na pravou `-1`. Pak budeme procházet naše pole hranic zleva doprava, v pomocné proměnné budeme udržovat součet plus a mínus jedniček, a kdykoli budeme uvnitř čtverečku (tedy mezi hranicemi), tak vypíšeme pomocnou proměnnou.

Teď ještě tu těžší část algoritmu – dvourozměrné obdélníky. Nedala by se stejná myšlenka s plus a mínus jedničkami použít i pro obdélníky? Dala, vyřešíme nejprve sloupečky a pak zopakujeme náš původní, řádkový postup.

Chtěli bychom, aby na konci druhé fáze v pomocné matici měl každý obdélník na příslušných řádkách jednu `+1` a jednu `-1` přesně tam, kde je jeho levá a pravá svislá hranice.

Tohle by ale hravě vytvořil náš původní algoritmus, pokud bychom jej spustili na sloupečky, do levého horního rohu vložili `+1`, a do levého spodního rohu `-1`. To vyřeší levou hranici, pro pravou hranici uděláme to samé, jen s opačnými znaménky.

Pro pořádek sesumírujeme: levý horní roh dostane `+1`, pravý horní `-1`, levý spodní `-1` a pravý spodní `+1`. Spuštění algoritmu na sloupečky vytvoří levou hranici obdélníků z `+1` a pravou hranici obdélníků z `-1`. V matici samozřejmě budou vyšší čísla, protože tento postup provádíme pro všechny obdélníky současně, stejně jako v jednorozměrné variantě. Spuštění algoritmu znovu, ale nyní na řádky, už dodá správné součty do políček.

Časová složitost byla opravdu $\mathcal{O}(MN)$, protože jsme nejprve za každý obdélník uložili čtyři hodnoty do matice, a pak ji dvakrát prošli – jednou po sloupcích a jednou po řádcích. Matici jsme měli pouze jednu, o velikosti $(M+1) \times (N+1)$, a tak je paměťová složitost stejná jako časová.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-5-6.c`

Martin Böhm

24-5-7 Cesta přes hranice

Mapa měst je vlastně ohodnocený neorientovaný graf, ve kterém jsou některé vrcholy celnicemi. Pro jednoduchost bude Linz vrchol A, Pasov vrchol B a Praha vrchol C.

Jednodušší varianta úlohy je vlastně jen hledání nejkratší cesty mezi vrcholy A a B a pak mezi vrcholy B a C, kde při cestování musíme zohledňovat, i kolika celnicemi jsme projeli. Na vzdálenost mezi dvěma vrcholy se budeme dívat jako na dvojici čísel (i, j), kde i je počet celnic, kterými jsme projeli, a j je vzdálenost, kterou jsme při tom urazili.

Tyto dvojice pak budeme porovnávat lexikograficky, tedy $(i, j) < (k, l)$ právě tehdy, když $i < k$ nebo $i = k$ & $j < l$. Nyní jen stačí použít Dijkstrův algoritmus a při porovnávání vzdáleností zohledňovat počet projetých celnic a máme výsledek. O detailech Dijkstrova algoritmu se můžete dočíst v naší kuchařce o haldě a Dijkstrově algoritmu.[47]

Nyní k těžší variantě. Opět hledáme nejkratší cestu z A do C přes B, akorát s tím rozdílem, že každou celnici započítáváme jen jednou. Je důležité si uvědomit, že nemusíme nutně použít nejkratší cesty A do B a z B do C, protože se nám může stát, že méně výhodnou cestu z A do B pak efektivně využijeme při cestování z B do C.

Po chvilce přemýšlení si všimneme, že v nejkratší cestě určitě bude existovat vrchol X takový, že nejdříve jdeme z A do X, pak z X do B, poté se z B vracíme zpět do X a nakonec cestujeme z X do C.

Jinými slovy při cestě z B do C nejdříve jdeme po stejné cestě, po které jsme přišli, pak se od ní odpojíme ve vrcholu X a už cestu z A do B nikdy křižovat nebudeme.

Proč? Kdybychom cestu z A do B křížili vícekrát, tak by to znamenalo, že jsme mezi těmito dvěma kříženími našli kratší cestu bez celnic, než je na příslušné části cesty z A do B, tedy by i původně bylo výhodnější jít po tomto nově nalezeném úseku.

Nyní, když víme, že nejkratší cesta takový vrchol X obsahuje, tak můžeme zkusit všechny možnosti toho, který to bude (včetně vrcholů A, B, C). Pokud zvolíme vrchol X pevně a vzdálenost X a A bude (i, j), vzdálenost X a B bude (k, l) a vzdálenost X a C bude (m, n), tak celková délka trasy při využití X bude $(i + k + m, j + 2l + n)$.

Jako X tedy vyzkoušíme všechny možné vrcholy a nejmenší vypočítaná hodnota bude naším řešením. Ke kompletnímu řešení nám už jen zbývá spočítat vzdálenosti z vrcholů A, B, C do všech ostatních vrcholů a to uděláme tak, že

[47] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`

pro každý z nich zvlášť spustíme Dijkstrův algoritmus a tím například zjistíme vzdálenost vrcholu A od všech ostatních vrcholů.

Časová složitost obou variant je stejná jako časová složitost Dijkstrova algoritmu, tedy $\mathcal{O}(n^2)$, nebo $\mathcal{O}((n+m)\log n)$, pokud v Dijkstrově algoritmu používáme haldu.

Ve vzorovém zdrojovém kódu můžete vidět řešení těžší varianty v jazyce C++. Pro přehlednost hlavních částí algoritmu není počítána výsledná cesta, ale pouze optimální vzdálenost.

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-5-7.cpp`

Karel Tesař

24-5-8 Jak hraje deskovky počítač?

Úkolem bylo prozkoumat Dvonn a zamyslet se nad součástmi algoritmu Alfa-beta, které jsou specifické pro tuto hru. To je právě zajímavá část vývoje umělé inteligence robota hrajícího hru, pokud je kostra algoritmu již daná. Do Dvonnu se sice pustili jen dva odvážlivci, nicméně obě řešení se mi líbila.

Nezaručuji, že zde předvedené myšlenky povedou k nejlepšímu možnému počítačovému soupeři, určitě lze toto řešení v mnohém ještě vylepšit. Předpokládána je alespoňpovšechná znalost pravidel.[48] Tahem se myslí tah jednoho hráče (někdy také půltah).

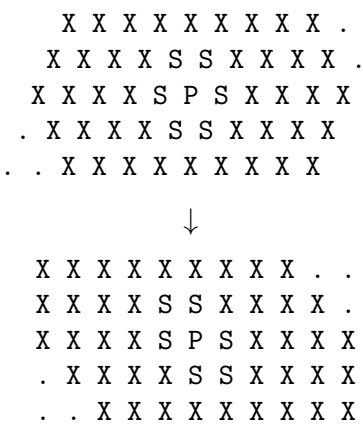
Zaměříme se především na část hry po rozestavení kamenů. Pro první část, tedy rozestavování kamenů, je doporučenou strategií dávat své kameny na okraj a co nejblíže červeným kamenům, ale nepokládat moc svých kamenů vedle sebe. Podle tohoto popisu je možné udělat ohodnocovací funkci pro Alfa-beta prohledávání.

Reprezentace pozice a generování tahů

Základem programu jistě musí být nějaká *reprezentace pozice* a na ní postavené generování možných tahů. Políčka na desce jsou v šestiúhelníkové mřížce, tu však lze reprezentovat přímo jen těžko. Proto políčka na desce trochu posuneme, přesněji řečeno i-tou řádku posuneme o $(i-1)/2$ políček a dostaneme dvourozměrné pole.

Lépe půjde transformace pochopit z obrázku na následující straně, který je převzat z řešení Vojty Hlávky. `X` znamená herní políčko, políčka `S` jsou sousedé `P`, `.` je políčko mimo desku (nicméně ve výsledném poli být musí).

[48] `http://deskovehry.blogspot.com/2009/10/pravidla-dvonn.html`



V každém prvku tohoto dvourozměrného pole musí být uložena barva a výška sloupku (0 pro prázdné políčko), a jestli se někde ve sloupku nachází červený kámen. Jednou možnou implementací je mít 3 pole, každé pro jednu vlastnost. Spolu s proměnnou značící, kdo je na tahu, máme takto kompletní reprezentaci pozice.

Generování tahů lze udělat jednoduše procházením všech políček a pro sloupky (věžičky) hráče na tahu vyhledat, kam mohou skočit. Nicméně v koncovce, kdy je spousta políček prázdných, už se vyplatí generovat tahy chytřeji.

Jednou z možností je udržovat si pro oba hráče seznam jejich sloupků na desce. Ten pak stačí projít a podívat se, kam mohou sloupky skočit. Seznamy se vygenerují před prohledáváním a budou se udržovat při provádění a vracení tahů.

Možná ještě efektivnější, ale složitější na implementaci je před prohledáváním vygenerovat všechny tahy a pak jen udržovat jejich seznam. Je však třeba si dát pozor při odstraňování kamenů na to, že bude třeba smazat mimo jiné i tahy vedoucí mezi odstraněné kameny.

Provádění tahů je většinou přímočará záležitost, v této hře však má háček: je třeba zjišťovat, jestli se nějaká skupina kamenů nestala po tahu nedosažitelnou od červených kamenů.

Podíváme se proto na okolní kameny právě odebraného kamene, konkrétně na to, jestli se vyskytují v nějakých shlucích (např. dva sousedi vedle sebe, prázdné políčko, dva sousedi a prázdné políčko dávají dva shluky). Shluky mohou být maximálně tři. Pokud je shluk jen jeden a není skákáno s věžičkou obsahující červený kámen, určitě se nic odstraňovat nebude.

Jinak je nejspíš nutné spustit prohledávání, v kterých komponentách příslušejících shlukům jsou červené kameny. Asi nejlépe to půjde prohledáváním do

šířky, dokud v každé komponentě nenajdeme červený kámen nebo ji neprojdeme celou (v tom případě ji odstraníme).

Pokud neskáčeme se sloupkem obsahujícím červený kámen, můžeme zastavit prohledávání už po odstranění všech komponent až na jednu – určitě v ní červený kámen někde bude.

Při prohledávání stromu hry je třeba tahy i vracet, je tedy nutné udržovat si historii provedených tahů. Vracení odstraněných kamenů lze udělat pomocí spojového seznamu. Ten se vytvoří při provedení tahu a při jeho vracení se projde.

Pro účely určení, kdo vyhrál, se hodí udržovat si součet výšek sloupků hráče, což lze jednoduše doplnit do provádění a vracení tahů.

Ohodnocování pozice a tahů

Ohodnocování pozice bývá pro algoritmy založené na Minimaxu nejspíše tou nejtěžší částí. Na jednu stranu by mělo být velmi rychlé, vyplatí se totiž prohledávat o jedna hlouběji, než mít pomalou ohodnocovací funkci. Na druhou stranu chceme umět rozlišit slibné pozice od těch špatných. Tahle část řešení úlohy je bez praktického vyzkoušení nejvíce diskutabilní.

Z hlediska efektivity není dobré, aby ohodnocovací funkce prošla v každém listu prohledávacího stromu celé herní pole, hodí se tedy udržovat si některé vlastnosti inkrementálně – měnit je jen při provádění a vracení tahů na základě políček ovlivněných tím tahem. Z takových vlastností už se pak může spočíst výsledná hodnota v každém listu v čase nezávislém na velikosti herní desky.

Přestože vyhraje ten, kdo má vyšší součet výšek věžiček, není podle toho možná dobré ohodnocovat, protože některé věžičky může jednoduše vzít soupeř. Spíše je zajímavější určit pro vyšší věžičky poblíž červeného kamene, kdo by vyhrál, kdyby se o tu věžičku začalo bojovat (hráči by na ni střídavě pokládali kameny).

Je tedy třeba vědět, které kameny mohou na tu věžičku doskočit, a udržovat si součet ovládaných věžiček pro oba hráče (ale jen těch poblíž červených kamenů). Na druhou stranu je nutné dát si pozor, jelikož některé kameny mohou napadat či chránit více věžiček najednou.

Zajímavou heuristikou může být počet možných tahů, tedy kdo má více tahů, může lépe ovlivňovat hru a je ve výhodě. V koncovce často vyhraje ten, kdo zahraje posledních pár tahů, přičemž soupeř musí kola vynechávat, protože mu došly tahy.

Výhodné jsou často jen tahy vedoucí na soupeřův nebo červený kámen anebo chránící vlastní vysokou věžičku, ty by měly mít větší vliv na hodnocení. Je však třeba jejich počet udržovat při provádění tahů, což nemusí být lehké.

Někdy je výhodné mít ve svém sloupku červený kámen, protože pak s ním lze uskočit v případě možnosti připravit soupeře o jeho věžičky. Sloupek s červeným kamenem se asi hodí započítávat, jen když má možnost někam se pohnout.

Pokud bychom toto dokázali udržovat inkrementálně, ohodnocení pozice hráče by vypadalo takto, přičemž konstanty chtějí ještě doladit:

$$\begin{aligned}&\texttt{pocetTahu} + 2 \cdot \texttt{pocetTahuNaVezSoupere} + \\ &\qquad 10 \cdot \texttt{soucetOvladanychVezicek} + \\ &\quad 30 \cdot \texttt{pocetVezicekSCervenymKamenem}.\end{aligned}$$

Ohodnocení pozice z pohledu bílého hráče je pak jednoduše ohodnocení bílého mínus ohodnocení černého. Z pohledu černého to samé vynásobené -1.

Alfa-betě se kvůli efektivnímu ořezávání hodí mít tahy seřazené od těch nejlepších. Jako první se asi vyplatí vyzkoušet tahy, které odstraní více soupeřových kamenů než našich nebo které tomuto odpojení napomáhají (po tahu půjde skupina odpojit jedním tahem).

Potom bývají dobré tahy, díky kterým můžeme nějakou vysokou věžičku získat, a dále ty, při nichž skáčeme na soupeřův kámen. Až naposledy se vyplatí zkoušet tahy, při nichž skočíme na svůj vlastní kámen, což je většinou nevýhodné.

Herní strom

U hry se často zkoumá *velikost herního stromu*, aby bylo možné odhadnout, jak moc těžké je hru vyřešit, tedy najít vyhrávající strategii. Definuje se jako počet listů stromu, neboli počet různých her, které je možné sehrát.

Obvykle se nedá spočíst přesně a odhaduje se umocněním typického větvení (počtu možných tahů hráče v nějaké pozici) na maximální délku hry (někdy maximální až na výjimečně dlouhé hry). Pro jednoduchost se omezíme na jedno konkrétní rozmístění kamenů, tj. vynecháme první část hry.

Pro Dvonn je možné odhadnout větvení počtem kamenů hráče na začátku (23) krát počet možných směrů (6), tedy 138. V každém tahu hráče se musí uvolnit alespoň jedno políčko a na konci musí být alespoň jedno políčko obsazené, což dává 48 půltahů a tedy maximálně $138^{48} = 5{,}18 \cdot 10^{102}$ možných her. Proti tomu je odhadovaný počet atomů ve vesmíru jako nic.

Skutečná velikost herního stromu je samozřejmě o dost nižší a odhad na větvení lze podstatně zlepšit. Můžeme například využít faktu, že počet sloupků při hře neustále klesá a tedy klesá i počet možných tahů.

Další věc, která se pro hry odhaduje, je počet dosažitelných stavů. Jelikož jeden stav se ve stromě může vyskytovat mnohokrát, bývá toto číslo mnohem menší, přesto však pořád astronomické. Odhadnout tento počet shora je pěkné kombinatorické cvičení.

Složitost různých her a více informací lze najít na Wikipedii.[49] Tolik v „krátkosti“ pro Dvonn. Pokud vás toto téma zajímá hlouběji, můžete se mi ozvat. :-) Užijte si léto!

Pavel „Paulie“ Veselý

[49] `http://en.wikipedia.org/wiki/Game_complexity`

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
0.	Vzorový řešitel			40	300.0
1.	Vojtěch Hlávka	GŠlapanice	3	39	268.3
2.	Martin Raszyk	G_Karvina	2	28	235.3
3.	Lukáš Ondráček	GVolgogrOS	3	25	218.1
4.	Dominik Macháček	GLanškroun	3	30	167.2
5.	Mark Karpilovskij	GJarošeBO	3	19	165.0
6.	Michal Pokorný	SŠkybernHK	4	26	161.0
7.	Vojtěch Vašek	GHli	3	24	155.7
8.	Alexander Mansurov	GNVPlániPH	3	21	150.4
9.	Jiří Eichler	SlovanGOL	4	20	138.0
10.	Martin Španěl	ArcibisGPH	3	20	136.5
11.	Jan Knížek	G_Strakon	1	17	116.1
12.	Jerguš Greššák	ŠPMNDaGB	3	13	110.9
13.	Ondřej Mička	GJírovcČB	3	12	108.2
14.	Matej Lieskovský	GOmskPha	2	15	106.9
15.	Vojtěch Sejkora	SPSE_Pard	3	18	100.9
16.	Michal Punčochář	GJírovcČB	2	12	99.1
17.	Štěpán Trčka	GSlavičín	1	16	96.5
18.	Dalimil Hájek	GKepleraPH	1	19	94.6
19.	Rastislav Rabatin	GJHroncaBA	3	11	90.8
20.	Lukáš Folwarczný	GKomHavíř	4	9	89.7
21.	Jan-Sebastian Fabík	GJarošeBO	2	9	87.0
22.	Aneta Šťastná	GOmskPha	2	12	80.2
23.	Martin Mirbauer	PORGPha	4	13	77.1
24.	Martin Šerý	GJírovcČB	2	5	49.1
25.	Jonatan Matějka	GJírovcČB	2	6	46.0
26.	Jitka Fürbacherová	GKlatovy	3	7	45.7
27.	Kateřina Zákravská	GJar	3	5	44.6
28.	Ondřej Cífka	GNAlejíPH	3	5	43.4
29.	Ondřej Hübsch	GArabskáPH	2	5	43.0
30.	Petra Pelikánová	GJarošeBO	3	5	41.3
31.	Ráchel Sgallová	GZborovPH	2	5	40.7
32.	Petr Houška	GJírovcČB	2	4	39.3
33.	Anna Zákravská	GJar	3	5	38.9
34.	Joel Jančařík	MensaG	4	6	38.8
35.	David Bernhauer	GZborovPH	4	6	38.5
36.	Jan Hadrava	GZborovPH	4	4	32.6
37.	Sabína Fraňová	GDubNVahom	3	5	31.1
38.	Jindřich Pilař	GBroumov	4	7	30.2

39.	Tereza Hulcová	GKlatovy	3	7	28.5
40.	Bohumil Mravenec	GArabskáPH	3	4	25.0
41.	Václav Volhejn	GKepleraPH	-1	3	24.8
42.	Pavel Kratochvíl	VOŠGSvětlá	4	6	24.5
43.	Radovan Švarc	G_ČTřebová	1	4	23.3
44.	Josefina Mádrová	GDobruška	4	4	21.4
45.	Pavel Salva	VOŠŠumperk	2	3	19.4
46.	Dominik Smrž	GOhradníPH	2	2	19.1
47.	Pavel Bárta	SPŠTrutnov	2	6	17.2
48.	Štěpán Šimsa	GJungmanLT	3	2	16.3
49.	Tomáš Velecký	GBezručeFM	1	2	15.9
50.	Jan Žárský	VSSKopř	1	2	14.1
51.	Zuzana Vozárová	GJHroncaBA	4	2	13.8
52.	Vojtěch Bednárik	MG_Vsetín	4	1	10.7
53.	Vojtěch Polívka	GMikulášPL	4	1	9.5
54.	František Zajíc	G_Nymburk	-1	4	9.2
55.	Tomáš Hromada	MG_Vsetín	4	1	9.0
56.	Michal Hruška	GJirsíkaČB	4	1	7.6
57.	Matěj Židek	GBroumov	4	1	6.2
58.	Vladan Gloncák	GLŠtúraTN	3	1	6.0
59.	Břetislav Hájek	GČesBrod	-2	5	5.9
60.	Juda Kaleta	GKlatovy	3	2	5.2
61.	Jan Pavlík	VOŠŠumperk	4	1	1.3

Obsah

Martin Böhm a kolektiv

Korespondenční seminář z programování XXIV. ročník

Autoři a opravující úloh:
Martin Böhm, Jan Bok, Pavel Čížek, Karel Král
Tomáš Maleček, Martin Mareš, Lucka Mohelníková, Jitka Novotná
Jirka Setnička, Jiří Setnička, Karel Tesař, Michal Vaner
Pavel Veselý, Peter Zeman

Autoři příběhů v zadání:
Pavel Veselý, Jan Matějka, Martin Böhm, Jiří Setnička, Radim Cajzl

Vydal MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 421. publikaci.

TEX-ová makra pro sazbu ročenky vytvořili Martin Mareš, Jan Matějka a Radim Cajzl.

S jejich pomocí ročenku vysázel Radim Cajzl.

Obrázek na obálce nakreslila Lucie Mohelníková.

Sazba byla provedena písmem Computer Modern v programu TEX.

Vytisklo Reprostředisko UK MFF.

Vydání první, 198 stran
Náklad 200 výtisků
Praha 2012

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

ISBN 978-80-7378-227-6

ISBN 978-80-7378-227-6