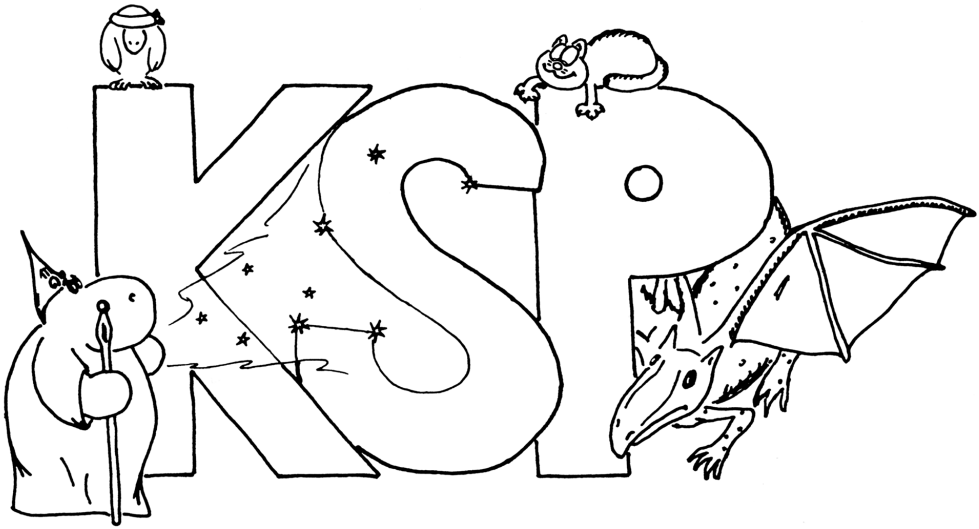
PETR KRATOCHVÍL A KOLEKTIV

Korespondenční seminář z programování

XX. ročník – 2007/2008



matfyzpress

VYDAVATELSTVÍ
MATEMATICKO-FYZIKÁLNÍ FAKULTY
UNIVERZITY KARLOVY V PRAZE

PETR KRATOCHVÍL A KOLEKTIV

Korespondenční seminář z programování

XX. ročník – 2007/2008

matfyzpress

Praha 2008

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

ISBN 978-80-7378-055-5

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož dvacátý 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. Proto některé úlohy *KSP* svou obtížností vysoko přesahují rámec běžného středoškolského vzdělání, a tudíž i požadavky při přijímacím řízení na vysoké školy. To ovšem vůbec neznamená, že nemá smysl takové problémy řešit – při troše přemýšlení není příliš obtížné nějaké (i když někdy ne to nejlepší) řešení nalézt. Nakonec – posuďte sami.

KSP probíhá tak, že student od nás jednou za čas dostane poštou zadání obvykle šesti úloh, v klidu domácího krbu je (ne nutně všechny, počítají se nejlepší čtyři) vyřeší, svá řešení v přiměřeně vzhledné podobě sepíše a do určeného termínu zašle na naši adresu (ať už fyzickou či elektronickou). My je poté opravíme a spolu se vzorovými řešeními a výsledkovou listinou pošleme při vhodné příležitosti zpět na adresu studenta. Tento cyklus se nazývá *série,* resp. kolo.

Za jeden školní rok obvykle proběhne pět sérií. Závěrečným bonbónkem je pak pravidelné *soustředění* nejlepších řešitelů semináře, konané obvykle na začátku ročníku dalšího a zahrnující bohatý program čítající jak aktivity ryze odborné (přednášky na různá zajímavá témata apod.), tak aktivity ryze neodborné (kupříkladu hry a soutěže v přírodě).

Náš korespondenční seminář není ojedinělou aktivitou svého druhu – existují korespondenční semináře z fyziky a matematiky při MFF, jakož i jiné programátorské semináře (kupříkladu bratislavský). Rozhodně si však nekonkurujeme, ba právě naopak – každý seminář nabízí něco trochu jiného, řešitelé si mohou vybrat z bohaté nabídky úloh a najdou se i takoví nadšenci, kteří úspěšně řeší několik seminářů najednou.

Velice rádi vám odpovíme na libovolné dotazy jak ohledně studia informatiky na naší fakultě, tak i stran jakýchkoliv informatických či programátorských problémů. Jakýkoliv problém, jakákoliv iniciativa či nabídka ke spolupráci je vítána na adrese:

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/*

Zadání úloh

Nad Škytánií zapadlo slunce. Všichni lidé oslavovali konec dalšího úspěšného dne v kolotoči života a pomalu se chystali na kutě. Až na jednoho ...

Uprostřed Temného hvozdu v temné věži zazvonil na potemnělém nočním stolku černý budík. Z postele se natáhla ruka v černé sametové noční košili a zamáčkla ho. Temný mág se posadil na posteli a dlouze zívl. Natáhl si své černé bačkůrky se zajíčky a s pomalostí člověka, který už má svá nejlepší léta za sebou, vstal. Protáhl se a z nočního stolku sebral zvoneček. Zazvonil.

Chvíli netrpělivě pozoroval dveře a pak zazvonil znovu. Stále nikdo nepřicházel.

„Sehnat dneska dobrý služebnictvo ...“ ozval se z tmavého rohu místnosti sarkastický hlas kocoura Felixe, který byl sice vzhůru, ale jako každý jiný kocour nespěchal se vstáváním z pěkně vyhřátého pelíšku. Temný mág si povzdechl, odložil zvoneček a šoural se ke dveřím.

„Mám já tohle zapotřebí? To má být spolehlivost?“ mumlal si pod vousy, když vystupoval po temném schodišti. Otevřel dveře a uviděl zombíka Vildu skloněného nad stolem. Vilda byl vesnický prosťáček, který jednoho dne dostal úžasný nápad – jít zabít Temného mága a stát se hrdinou. Naštěstí pro něho mu to Temný mág rozmluvil a dokonce ho přijal jako svého sluhu. To se rozumí, že správný temný mág nemůže mít jako sluhu jen tak někoho, a tak udělal z Vildy zombii. Bohužel nekromancerství nepřišel nikdy na chuť, takže nechal Vildu naživu a přikázal mu, aby se pravidelně každý den natíral zelenošedou barvou a používal výhradně deodorant Eau-de-zdechlina.

„Ty jsi neslyšel, že jsem zvonil? A co to tu vlastně vyvádíš?“ zahartusil mág sotva nabral dech po namáhavé cestě do schodů.

„Chtěl jsem Vám udělat radost, Vaše temné mágstvo,“ odpověděl Vilda a dal se do vysvětlování (nebo spíše vytmavování): „Našel jsem ve skříni šachovnici a ona na ní jsou i nějaká světlá políčka. Chtěl jsem jí celou pootáčet tak, aby tam byla jen tmavá políčka.“

„To není špatný nápad, ale někde jsem slyšel, že jedna dáma musí stát na bílém políčku. Tak nech tady to jedno rohové políčko světlé, ať si pak můžu zahrát šachy.“

20-1-1 Temná šachovnice — 6 bodů

Vaším úkolem je pomoci Vildovi splnit mágův rozkaz. Protože je ale šachovnice kouzelná, není to úkol jednoduchý. U kouzelné šachovnice se totiž políčka přebarvují tak, že se zvolí libovolný sloupec nebo libovolný řádek a všechna světlá políčka v tomto řádku/sloupci se stanou tmavými a všechna tmavá světlými.

Na začátku máte šachovnici s běžně obarvenými černými a bílými políčky. Cílem je přebarvit políčka na šachovnici pomocí posloupnosti popsaných tahů (prohození barev v nějakém řádku či sloupci) tak, aby všechna políčka byla tmavá až na jedno v levém horním rohu, které má zůstat bílé.

Šachovnice, kterou měl Vilda k dispozici, měla rozměr 4×4, takže stačí popsat postup k obarvení pro tento rozměr šachovnice (pokud by šachovnice obarvit nešla, dokažte proč to nejde).

Bonus: Pokud vyřešíte Vildův problém i pro obecnou šachovnici o velikosti $N \times N$, 3 bonusové body vás neminou.

Vilda usilovně přebarvoval šachovnici, ale jeho mágstvu se stále něco nelíbilo.

„Není tu nějak moc světla?“ rozhlédl se po místnosti. A skutečně. Všude bylo tolik světla, že by se tam dalo snad i číst. Takhle by to ovšem nešlo. V temné věži musí být tma. Je to součástí temno-mágské image.

Temný mág si vykasal noční košili a vylezl na stoličku, aby sundal od stropu malou černou lucerničku.

„Krákrákrááá...“ vřítil se do místnosti havran Kiri, prolétl těsně nad Vildovou hlavou a zamotal se mágovi do košile.

„Co ... to ...“, stačil ze sebe ještě vykoktat mág, než se zřítil ze stoličky i s lucernou. Když se za Vilíkovy asistence posbíral a přepočítal si žebra, spatřil na zemi vzniklou pohromu. Temná lampička byla na cimpr campr. Temný kámen, který se používá místo knotu, byl roztříštěný na milion kousků.

„No mňaucta, to je zase nepořádek,“ okomentoval situaci kocour Felix, který se rozhodl, že konečně vstane, a právě vešel do místnosti hledaje něco k snědku.

„To je nadělení,“ povzdechl si mág. „Bez temné lampičky tu už nebude temno. Můj nepořádek, moje neumyté nádobí, moje špína – na to všechno se teď budu muset koukat. A jestli se to rozkřikne ve Škytánii ...“ Posadil se na židli a podepřel si ustaranou hlavu dlaněmi.

Být temným mágem bylo pěkně těžké. Nejen, že jste si museli neustále udržovat potřebný respekt a dbát na vzhled, ale každou chvíli napadlo nějakou bandu barbarských válečníků, že vyčistí temný hvozd a přitom vás – jako mimochodem – zapíchnou. Mág byl zvyklý tyhle věci řešit jednoduše – nad šálkem černého čaje. To byste nevěřili, co si takový barbar nechá nakecat, když mu seberete meč a dáte do ruky čajové sušenky. Ale tohle byl úplně jiný problém. Bez temné lampičky přijde o svůj image. A pak mu sem začnou lézt lidi jako do holubníku ... Ne! Tomu je potřeba učinit přítrž.

Mág vyskočil ze své židle: „Vildo! Sbal nám věci. Vydáme se najít nový temný kámen!“ Poslední slova zaduněla věží s temnou ozvěnou ... Nastala nepříjemná chvíle trapného ticha.

„Krá?“ dodal Kiri s nevinným pohledem slepého havrana, ale jeho špatné načasování celkový dojem okamžiku už nezachránilo.

Tolik věcí bylo potřeba na cestu nachystat. Černý chléb, černé sametové oblečení, tmavé deky a spoustu dalšího vybavení, které bylo temné, nebo alespoň úplně černé. Mág chtěl vyrazit co nejdříve, a tak se Vilda pěkně oháněl ...

20-1-2 Příprava na cestu — 12 bodů

Váš další úkol je opět pomoci zombíku Vildovi. Tentokrát s přípravou na cestu. Mág chce vyrazit již za M minut a Vilda během nich musí věnovat N úkolům. Pomozte mu se rozhodnout, kolik času se má věnovat jednotlivým činnostem, aby pravděpodobnost toho, že bude Temný mág spokojen, byla co největší.

Na vstupu váš program dostane čísla N a M. Poté bude následovat N řádků, každý s $M+1$ čísly a_0 až a_M. Číslo a_i udává, jaká je pravděpodobnost, že mág bude se splněním dílčího úkolu spokojen, když se Vilda bude věnovat jeho plnění i minut. Výstupem programu by měl být pro každý úkol počet minut, který mu má Vilda věnovat, aby byl součin pravděpodobnosti úspěchů všech jednotlivých úkolů co největší. Formálně je tedy výstupem programu posloupnost nezáporných čísel $b_1, \ldots, b_N$ takových, že jejich součet je M a součin čísel A_{B_1} pro první činnost, a_{b_2} pro druhou činnost, ..., a_{b_N} pro N-tou činnost byl co největší.

Příklad: Pro vstup 2 2
0.15 0.2 0.9
0.01 0.4 0.8
je správný výsledek 0 2. Výsledná pravděpodobnost úspěchu je pak $0.15 \cdot 0.8 = 0.12$.

Vše bylo nachystáno. Kocour Felix nervózně přešlapoval a všem dával hlasitě najevo, že on opravdu jít nemusí. Vilda zamkl temnou věž temným klíčem a následně ho schoval pod temnou rohožku.

„Kupředu," zavelel mág, i když sám neměl přesnou představu, kam se pro temný kámen vydat. Jedno ale bylo jasné. Nejprve se musí dostat z temného hvozdu. Většina lidí se do hvozdu bála jen vstoupit, neboť se proslýchaly různé pověry o oživlých stromech a krvelačných bestiích, které tam žijí. Mág měl mnohem prozaičtější důvod, proč se dostat ven – není v něm vidět na cestu.

Po necelém dni putování dorazili k temné řece, která protékala hvozdem a oddělovala jeho temnou část od té mnohem temnější. Řeka byla opravdu široká, takže bylo sotva vidět na druhou stranu. Na břehu byl vytažený vrak lodi, která kdysi sloužila pro přepravu bláznů, sebevrahů a jiných dobrodruhů, kteří měli převážně namířeno do temné věže ... na šálek čaje.

Felix si smočil v řece tlapku.

„Brrr... ta je studená. Tak jsme se prošli a teď bychom mohli jít domů, ne?" obdařil přítomné potměšilým úsměvem.

„V žádném případě,“ zamítl jeho nadšení mág. „Vildo, sekeru, pilu a hybaj na dřevo . . . “

20-1-3 Oprava lodi — 8 bodů

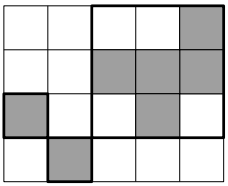
Vaším úkolem je spočítat, kolik dřeva bude Vilda potřebovat na opravu trupu lodi. Plášť lodi je vlastně dvojrozměrné pole, jehož každé políčko je čtvereček s jednotkovým obsahem. Každé políčko může být v pořádku nebo děravé. Dvě děravá políčka spolu sousedí, pokud sousedí hranou. Cílem Vildy je trup lodi zazáplatovat, a to tak, aby každá záplata byla obdélníková, žádné dvě záplaty se nepřekrývaly a každá souvislá oblast děr byla pokryta právě jednou záplatou (žádná díra nesmí zbýt nezazáplatovaná a dvě sousedící díry musí být zakryty stejnou záplatou).

Díry jsou rozmístěny tak, že když bude každá souvislá oblast děr překryta nejmenší možnou záplatou, pak k překryvu nikdy nedojde.

Dřeva chce Vilda použít samozřejmě co nejméně.

Na vstupu dostane váš program čísla M, N a D. Hodnoty N a M jsou rozměry pláště (šířka a výška) a D je počet děr. Následuje D řádků, které popisují, které jednotkové čtverce jsou děravé (každá díra je popsaná souřadnicemi řádek, sloupec). Výstupem programu je číslo, které udává, kolik jednotek dřeva bude nejméně potřeba, aby byla každá souvislá oblast děr byla pokryta jedním obdélníkovým kusem dřeva.

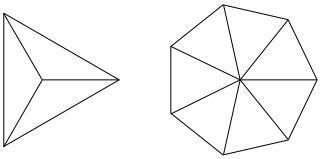
Příklad: Pro trup lodi 5×4 ($N = 5$, $M = 4$) a 7 děr na souřadnicích $(2, 3)$, $(4, 2)$, $(1, 5)$, $(3, 4)$, $(3, 1)$, $(2, 5)$, a $(2, 4)$ je třeba 11 jednotek dřeva. Pro lepší představu, zazáplatování trupu lodi vypadá takto:



Díky Vildově zručnosti a mágovým zkušenostem se během krátké doby podařilo zazáplatovat celý trup. Všemi silami a jedním kouzlem pak loď spustili na vodu. Už se chtěli vydat na cestu, ale Kiri při svém náhodném poletování naslepo nechtěně přistál i na kormidle. Ozvalo se křupnutí a celé kormidlo se sesypalo. A protože havran nemá ruce, kočky nepracují a temný mág je tu hlavně od přemýšlení, nezbývalo Vildovi nic jiného, než kormidlo opravit . . .

20-1-4 Kormidlo — 10 bodů

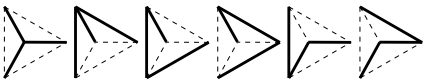
Správné námořnické kormidlo s N loukotěmi je v podstatě pravidelný N-úhelník, jehož střed je spojen s každým z N bodů na obvodu. Skládá se tedy z $2N$ rovných dílů. Kormidlo s třemi a sedmi loukotěmi si můžete prohlédnout na následujícím obrázku.



Vilda chce ale kormidlo opravit co nejdříve, a tak se rozhodl, že ho sestaví neúplné – pouze z N rovných dílů. Přitom chce, aby z každého z $N+1$ vrcholů kormidla vedl alespoň jeden díl a všech N rovných dílů bylo navzájem spojeno (tj. kormidlo se nerozpadá na dva či více dílů).

Napište program, který dostane na vstupu $N \geq 3$. Výstupem vašeho programu by měl být počet způsobů, kterými může sestavit Vilda kormidlo s N loukotěmi z N dílů tak, aby z každého vrcholu neúplného kormidla vedl alespoň jeden díl a kormidlo se nerozpadalo na více částí. (Pro znalce můžeme také říct, že chceme spočítat počet koster daného kormidla.)

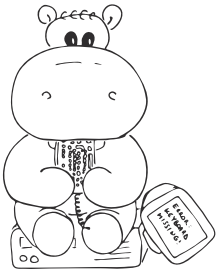
Příklad: Pro $N = 3$ lze kormidlo sestavit 16-ti způsoby. Jsou to



, posledních 5 ve 3 otočeních.

Kormidlo bylo opraveno a Felix přemluven, aby opustil pevnou půdu pod nohama a nastoupil s nimi na loď. Vypluli. Proud řeky byl mírný a plavba probíhala hladce. Už byli za polovinou, když si mág všiml, že na druhém břehu se něco pohybuje ...

20-1-5 Praktická – Studentův rozvrh — 10 bodů



Po vyhodnocení průběhu testovacího kola v poslední sérii loňského ročníku a po zpracování vašich připomínek jsme se rozhodli, že zavedeme praktickou úložku nastálo. A o co tedy půjde?

KSP byl spíše teoretickým seminářem, ve kterém šlo především o nalezení algoritmicky správného řešení a na implementaci nebyl kladen velký (resp. téměř žádný) důraz. V tomto trendu chceme samozřejmě pokračovat, ale s malou výjimkou. Pátá úložka v každé sérii nyní ponese přízvisko „praktická“ a bude prověřovat nejen vaše teoretické znalosti, ale také vaše schopnosti programátorské.

V praktické úložce nemusíte řešení vůbec popisovat, nebo jakkoli komentovat, ale zato musíte *jenom* odladit funkční program, který danou úlohu vyřeší. Odevzdávat budete pouze zdrojový kód, a to přes webovou aplikaci CodEx (The Code Examiner), která sídlí na adrese *https://codex2.ms.mff.cuni.cz/ksp/*. Přihlašovací jméno a heslo do CodExu je totožné s přihlašovacím jménem a heslem

do webového submitovátka, které již znáte řadu let. Pokud nemáte dosud zřízený účet na submitovátko, musíte se nejprve zaregistrovat (automaticky vám bude vytvořen i účet na CodExu). CodEx bude přes prázdniny odstaven, ale začátkem září jej opět spustíme (detaily se objeví na webových stránkách KSP).

Opravování probíhá tak, že CodEx převezme váš zdrojový kód, zkompiluje ho a následně jej pustí na sadu testovacích dat. Každý test má navíc nastaven časový a paměťový limit, který vaše řešení nesmí překročit. Za úspěšně vyhodnocené testy dostanete body a celkový součet bodů ze všech testů tvoří hodnocení vašeho řešení.

Vzhledem k tomu, že je velice obtížné napsat perfektní řešení na první pokus, budete mít pokusů více (detaily se dozvíte přímo v CodExu). Do výsledku se vám bude počítat nejlepší odevzdané řešení.

Odevzdávat můžete pouze zdrojové kódy napsané v jazycích Pascal, C a C++. Příznivcům ostatních jazyků se omlouváme, ale není v našich silách rozumně testovat i jiné jazyky (zvláště pak některé exotické, nebo interpretované).

Další podrobnosti a technické detaily naleznete přímo v CodExu. Pokud byste měli jakékoli dotazy, technické potíže apod., obraťte se na známou adresu KSP či na příslušnou sekci diskusního fóra. Přejeme hodně zábavy při řešení.

Zadání:

Každý student řeší na začátku semestru stejný problém: které přednášky si zapsat a které ne. Týden není nafukovací, a tak se stane, že jsou některé přednášky naplánované na ten samý čas. Každý student si chce samozřejmě zapsat přednášek co nejvíc, takže musí pečlivě vybírat ... a to je právě úkol pro vás. Napište program, který z dané množiny přednášek vybere co největší podmnožinu, ve které se žádné dvě přednášky nepřekrývají.

Seznam přednášek, které by student rád vystudoval, je uložen v souboru `prednasky.in`. Na prvním řádku se nachází číslo N, které označuje počet přednášek, a na následujících N řádcích se nachází jednotlivé přednášky. Přednášky jsou číslovány od 1 do N, takže na řádku $i+1$ se nachází i-tá přednáška. Přednáška je popsána dvěma celými čísly s_i a f_i oddělenými mezerou, kde s_i je čas začátku přednášky a f_i je čas konce přednášky. Čas si můžete představit např. jako počet sekund od začátku týdne (všechny časy jsou kladné a vejdou se do 32-bitového `integeru`).

Přednášky i a j se *nepřekrývají*, pokud platí, že $f_i < s_j$ nebo $f_j < s_i$.

Nalezený rozvrh uložte do souboru `rozvrh.out` v následujícím formátu: Na prvním řádku bude jedno číslo M, které určuje počet vybraných přednášek. Na druhém řádku pak bude seznam M čísel – čísel přednášek oddělených mezerami setříděný vzestupně podle času začátků přednášek.

Pokud existuje více rozvrhů s maximálním počtem přednášek, stačí vypsat libovolný z nich.

Příklad:

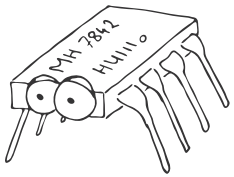
`prednasky.in`
```
5
2 4
1 7
6 9
9 11
5 8
```

`rozvrh.out`
```
3
1 5 4
```

20-1-6 Hrady, hrádky, hradla — 12 bodů

Milí řešitelé, letos jsme se vám rozhodli poodhalit tajemství, jež se skrývá uvnitř vašich počítačů. Nejsem si jistý, zdali máte někdy podobný pocit, ale mně se, když jsem byl menší, zdály děje uvnitř té bíle krabice vyloženě magické (i dnes se mi občas stane, že mě něco opravdu „magického“ překvapí). Nemohl jsem pochopit, jaktože ti malí plastikoví broučci umí tak rychle počítat a jakže vlastně zjistí, co po nich člověk chce. Tak se tomu teď spolu pojďme podívat na zoubek.

Začněme nejdříve hezky od základů a jelikož počítače jsou přístroje v podstatě schopné pouze pracovat s čísly, a to ještě jenom se dvěma, totiž jedničkou a nulou, matematice se chtě nechtě nevyhneme. Odložme ještě na chvíli otázku, jak si poradíme s většími čísly (k tomu nám pomůže zapisovat je ve dvojkové soustavě), a prozkoumejme, co všechno dokážeme provádět s jednotlivými bity (dvojkovými číslicemi čili nulami a jedničkami).



Mezi základní bitové operace patří negace (NOT), logický součet (OR), logický součin (AND) a nakonec ještě exkluzivní logický součet (XOR). Všechny tyto operace mají jeden výstup (0 nebo 1) a až na operaci NOT dva vstupy.

Nejjednodušší z operací je operace NOT. Na vstupu dostane jednu číslici a na výstupu má jedničku právě tehdy, je-li na vstupu nula a naopak. Operaci AND si můžeme představit jako zamčenou místnost se dvěma dveřmi za sebou. Abychom se dostali dovnitř, musíme odemknout oboje dveře. Operaci OR si představíme podobně, nyní však nejsou dveře za sebou, ale vedle sebe, a abychom se dostali dovnitř, stačí nám otevřít libovolné jedny dveře (nebo oboje). Operace XOR dává na výstupu jedničku právě tehdy, když má oba vstupy různé, zájemci si mohou sami zapřemýšlet kolikero dveří (popřípadě různě propojených předsíní) by bylo na tuto operaci třeba.

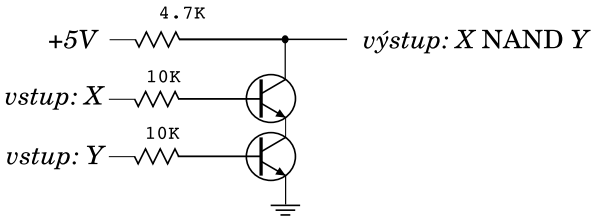
Všechny vícevstupové operace mají také své negované varianty, které vzniknou tak, že na výstup aplikujeme ještě operaci NOT. Budeme jim říkat NAND, NOR a XNOR. (Je jednoduché si rozmyslet, proč nemá smysl zavádět negovanou operaci NOT.) Negované varianty odpovídají situaci, kdy je v místnosti

zavřený tygr, vychovatelka či jiná šelma, kterou byste neradi viděli, a jste rádi, že jsou mezi vámi alespoň jedny zamčené dveře.

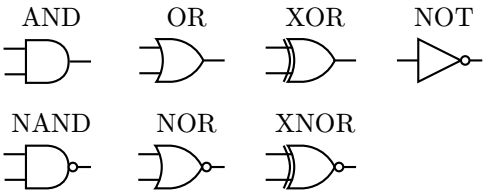
Chování základních operací lze také snadno popsat tabulkou:

x y	x AND y	x OR y	x XOR y	NOT x
0 0	0	0	0	1
0 1	0	1	1	
1 0	0	1	1	0
1 1	1	1	0	

V předchozích odstavcích jsme se na operace dívali jako na nějaké funkce, které dostanou na vstupu nějaké hodnoty a podle těchto hodnot se rozhodnou pro výstup. Nyní se na ně podíváme z elektrického hlediska. V elektronice odpovídají hodnoty 0 a 1 nějakým napěťovým úrovním. Pro nejběžnější logiku TTL je to 0 = 0V a 1 = +5V. Jiné logiky používají například 0 = -12V a 1 = +12V. Operaci se říká hradlo, což je nějaké fyzické zapojení tranzistorů, odporů, diod a jiných součástek. Aby se v zapojeních nevyskytovalo obrovské množství součástek, vyrábějí se integrované obvody, které obsahují obvykle několik hradel stejného typu. Takto třeba vypadá zjednodušené schéma zapojení obvodu NAND v technologii TTL (prosím nelekejte se, zapojení je tu spíše pro úplnost).



V elektrotechnických schématech se hradla označují následujícími značkami. Všimněte si že negované varianty mají na konci vždy malé kolečko.

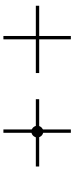


Takové schéma je vlastně orientovaný graf (to je množina vrcholů s některými dvojicemi vrcholů spojených orientovanou hranou), který má ve vrcholech obrázky hradel, vrchol má obvykle několik vstupních hran a jednu výstupní. Orientace hran se do obrázku nekreslí, neboť je vždy jasná ze symbolu hradla

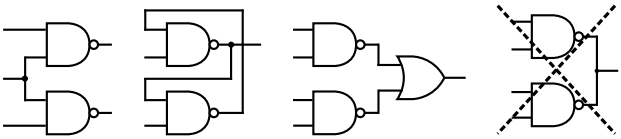
(víme, kde má vstupy a kde výstupy). Vstupní hrany se obvykle zapojují na výstupní, popřípadě se několik vstupních spojí dohromady; spojovat výstupní hrany s výstupními je ovšem zapovězeno.

Teď co to pro nás znamená: Hradlo je nějaký obrázek, který má na jedné straně jednu nebo více vstupních čárek, na něž se čarami připojí výstupy z jiných hradel, nebo se prohlásí za vstup celého schématu. Každé hradlo má jednu výstupní čárku, která se buď zapojí na vstup nějakého hradla, nebo se prohlásí za výstup celého schématu. Spojení dvou míst se značí nepřerušovanou čarou mezi nimi.

Ovšem ne všechna schémata jsou rovinná (rovinná se říká takovým, která lze nakreslit bez křížení čar spojujících hradla, což by vám mohlo připomínat rovinné grafy). Proto křížení čar v obrázku neznamená, že jsou elektricky spojeny (obrázek nahoře). Pokud chceme nakreslit spojené čáry (dráty), doplníme do místa křížení malé kolečko (obrázek dole).



Hradla můžeme propojovat například takto (poslední, přeškrtnutá varianta je zakázána):



A nyní co bude vaším úkolem, tedy zadání:

1. Sestavte hradlo `XOR` pomocí hradel `NAND`. Tím myslíme nakreslete schéma obvodu s hradly typu `NAND`, které se chová jako hradlo `XOR`. [4 body]
2. Ukázali jsme si 6 dvouvstupových logických funkcí (`AND`, `OR`, `XOR` a jejich negace). Existují ještě nějaké další? Nalezněte všechny dvouvstupové logické funkce. [4 body]
3. Dokažte, že každou dvouvstupovou logickou funkci (všechny už jste nalezli v předchozí úloze) můžete vytvořit pouze z hradel `NAND`. [4 body]

Mág popošel na příď lodi a zahleděl se do temnoty. A doopravdy, na druhém břehu se něco pohybovalo. A hned na několika místech. To budou zase ti temní elfové, pomyslel si mág a ledabyle naznačil Vildovi, aby si lehl na podlahu. Vilda okamžitě uposlechl a sotva se sehnul, prosvištělo mu nad hlavou několik šípů, které s tlumeným žbluňknutím zmizely v řece.

„To jsou celí oni," postěžoval si mág. „Nejdřív střílí, a pak teprve kladou otázky."

„No prosím, já vám říkal, abychom zůstali doma. Ale mě tady nikdo neposlouchá," přisadil si Felix a naježil srst.

„Kdo se opovažuje útočit na temného mága – pána hvozdu?!“ zaburácel temný mág a jeho hlas se nesl na všechny strany s údernou silou vichřice. Chvíli počkal, než se vzduch opět uklidnil, a pak se otočil k posádce lodi. Vilda se třásl na zemi, Kiri visel na kormidle v poloze „netopýr“ a Felixova srst vypadala, jako by právě proběhl silným elektro-magickým polem.

„M-mohl bys nás příště laskavě varovat, než použiješ svůj Hlas?“ ozvalo se ze středu chlupaté koule, která byla ještě před chvílí kocourem.

„Snad se zas tak moc nestalo,“ ohradil se mág. „Alespoň nás teď elfové nebudou obtěžovat.“

Dopluli ke břehu. Vilda vyskočil a uvázal loď k velkému kameni. Byl by použil strom, ale v temném hvozdu si jeden nemůže být jistý, kdy si takový strom vzpomene, že se půjde projít. I ostatní mezi tím vystoupili a poslechli si další variaci na ódu „já chci domů“ v kočičím podání. Vilda posbíral vybavení a vydali se na cestu. Ušli sotva pár kroků, když se odnikud vynořila skupinka temných elfů a v mžiku je obklíčila.

Ach ne, už zase? Být temným mágem je těžší, než byste mysleli. Nejtěžší je umět rozhodnout, kdy magii použít, a kdy ne. A tohle bylo přesně jedno z těch rozhodnutí. Samozřejmě by nebyl problém proměnit všechny přítomné elfy ve slintající idioty, ale to by znamenalo komplikace. Ostatní elfové by to nepřešli bez povšimnutí. Lezli by mu do věže a on by se příšerně nudil při nekonečných rozhovorech s elfími delegacemi. Nebo by to rozpoutalo válku, a to by mu tak ještě na stará kolena chybělo. Raději si poslechne, co mu chtějí.

Elfové nepromluvili ani slovo. Jen taktně – tedy s napjatými dlouhými luky – naznačili, aby je následovali. Prodírali se hvozdem dobrou hodinu, než přišli do pravé temno-elfské vesnice. Běžný pozorovatel by ji možná ani nerozeznal od jiné části hvozdu, ale zkušené oko rychle odhalilo výdutě ve stromech a umně ukryté provazové žebříky. Naproti se k nim svižnou chůzí blížil drobný postarší elf v honosném rouchu, které již na první pohled budilo dojem nadřazenosti.

„Zdravím vás,“ zašvitořil přátelsky a ostatním elfům s luky naznačil, že se můžou ztratit. „Omlouvám se za komplikace, ale vypadá to, že došlo k menšímu nedorozumění. Náš bývalý šaman, kterému se mimochodem dnes ráno přihodila malá – ehm – nehoda, vydal jisté nesmyslné příkazy . . . “ pokračoval elf s potměšilým úsměvem na rtech. „Zkrátka a dobře jste se sem dostali nedopatřením a já se vám jménem našeho kmene velice omlouvám. Budete-li mít zájem, můžete strávit noc v naší vesnici. Také jste zváni na slavnostní večerní rituál, při kterém bude zvolen nový šaman.“

Mágova družina byla pěkně unavená, a tak velkorysou nabídku přijala. Rituál byl velice působivý a mága nesmírně zajímalo, jak to ti elfové vlastně dělají. Bohužel se žádný z nich nechtěl o prastaré tajemství jejich kmene podělit, a tak si mág řekl, že na to přijde sám . . .

20-2-1 Volba šamana — 7 bodů

Volba šamana je delikátní záležitost. Jedná se o prastarý rituál, při kterém je nejstarší elf zvolen šamanem. Před začátkem si všichni stoupnou do kruhu tak, že každý elf vidí pouze své dva sousedy a nikoho jiného. Takže např. ani nevědí, kolik celkem jich v kruhu je.

Začátek rituálu je oznámen úderem do speciálního bubnu z hroší kůže a bubnování pokračuje po celou dobu rituálu. Mezi dvěma po sobě jdoucími údery si může každý elf vyměnit krátké zprávy se svým kolegou vlevo i vpravo (tzn. může každému říct nějakou zprávu a také si jeho zprávu vyslechnout). Každý elf si může pochopitelně také něco pamatovat.

Všichni elfové musí ve stejném okamžiku vědět, že rituál již skončil. V okamžiku, kdy skončí, musí všichni vědět, kdo je novým šamanem (včetně šamana samotného).

Aby tento rituál fungoval vždy, musí mít všichni elfové stejné instrukce (stejný algoritmus). Navíc nelze předpokládat, že by o sobě navzájem cokoliv věděli – tj. pro účely rituálu zná na počátku každý elf pouze své jméno a svůj věk (o ostatních neví nic). Pro jednoduchost předpokládejte, že žádní dva elfové nejsou stejně staří.

Navrhněte algoritmus, který musí každý elf provádět, aby rituál fungoval, tedy aby byl vždy zvolen nejstarší elf.

Ráno se temná družinka nasnídala a vyrazila směrem ven z temného hvozdu. Cesta ubíhala pomalu a ani sám mág si nebyl tak docela jistý, jestli jdou dobře. Stromy se občas přesazují, jak se jim zlíbí, takže není vůbec jednoduché se v temném hvozdu orientovat. Několik hodin bloudili a chodili stále dokola.

„Přísahal bych, že tenhle strom už jsem viděl," zamumlal si mág pod vousy.

„Ale však já už vás také několikrát viděl," přitakal strom. „Takhle byste se z hvozdu nikdy nedostali. Běžte pořád tímhle směrem, až dorazíte na louku. Tam musíte najít Doubka. Je to můj starý známý a ten vám poradí, kudy dál."

Chvíli to trvalo, než se temnému mágovi podařilo skrýt z tváře výraz překvapení, ale nakonec se sebral a odpověděl: „Hmm, to by nám velice pomohlo. A jak toho Doubka mezi všemi těmi stromy najdeme?"

„To je snadná pomoc. Doubek je pátý nejvyšší strom na louce."

Temný mág poděkoval a vydal se směrem, který mu strom naznačil mávnutím větví. Po delší chvíli dorazili na louku. Byla velká a přetékala slunečním světlem. Chvíli jen tak stáli na okraji a mžourali. Když jejich oči přivykly, spat-

řili uprostřed louky dvě řady stromů. Obě byly seřazené od největšího stromu po nejmenší.

„Tak, jdeme najít Doubka!“ zavelel temný mág a vykročil kupředu.

20-2-2 Setříděné stromy — 10 bodů

Na rozdíl od temného mága si my můžeme úlohu trochu zobecnit. Nebudeme hledat pátý, ale obecně k-tý nejvyšší strom.

Máme tedy dvě řady stromů, které jsou obě setříděné podle velikosti, a chceme nalézt k-tý nejvyšší z jejich sjednocení. Porovnání výšek dvou stromů je pochopitelně složitá operace, takže bychom ji chtěli použít co nejméněkrát.

Temný mág samozřejmě pospíchá, takže lineární řešení (slévání posloupností) je příliš pomalé.

Příklad: Mějme posloupnosti 12,8,3,1 a 20,19,15,4,2. Pátý největší prvek z jejich sjednocení je 8.

Mág si stoupl před strom, který se zdál být pátý nejvyšší z obou řad: „Strom Doubek, předpokládám.“

„Ale ano! A vy račte být?“ sklonil k němu strom svoji košatou korunu.

„Já jsem temný mág! Bydlím v temné věži na druhé straně řeky a právě teď jsem tu trochu zabloudil,“ vysvětloval mág.

„To chápu, vy lidé se neustále někde ztrácíte. Ale nic si z toho nedělejte, já vás zase najdu,“ prohlásil Doubek veselým tónem a vysadil si kořeny ze země. „Pojďte za mnou!“

Doubek byl na strom velice rychlý a poměrně vzdělaný. Mág měl co dělat, aby s ním udržel krok jak po cestě, tak v konverzaci. Asi po hodině svižné chůze dorazili na cestu, která procházela hvozdem. Doubek se zastavil a z koruny vyklepal několik veverek, havrana a spícího kocoura.

„Tudy se dostanete ven z hvozdu,“ ukázal jim cestu mávnutím větví. A než mu stačil mág poděkovat, Doubek se otočil a po několika krocích zmizel mezi ostatními stromy.

Teď už cesta ubíhala pohodlně. Druhý den začal hvozd řídnout a okolo poledne se před nimi otevřela rovná krajina. Na večer se utábořili a když druhý den opět vyrazili, spatřili v dálce před sebou malé městečko. Temný mág stejně neměl přesnou představu, kde by měl hledat temný kámen, a tak se rozhodl, že návštěva města nemůže být na škodu.

Když se k městu přiblížili, potkali na cestě vůz naložený senem. Táhli ho dva statní oři a na kozlíku seděl drobný človíček. Když uviděl povedenou družinku, otevřel ústa dokořán, až mu z nich vypadla fajfka.

„Tak co, strejdo? Uhneš nám, nebo tady na sebe budeme čumět do soudného dne?“ nadhodil konverzačním tónem Felix a ladně vyskočil na kozlík.

„Krá-krá, kráááááá!“ přisadil si Kiri a napůl přistál, napůl se zřítil do kupky sena.

Majitel povozu vzal nohy na ramena a přitom křičel, jako když ho na nože berou.

„Tak, to bychom měli,“ prohlásil spokojeně kocour a uvelebil se na seně. „Je libo svezení?“

Temný mág si povzdechl. Nějak mu nepřipadalo správné děsit obyčejné lidi. Na druhou stranu to alespoň vylepší jeho image. Ovšem ten vůz musí vrátit jeho majiteli. Je přece mág, ne nějaký špinavý zloděj.

Nasedli na vůz, Vilda si sedl na kozlík a pobídl koně. Už byli skoro ve městě, když si všimli, že se něco děje. Lidé pobíhali zmateně sem a tam a většina spěšně město opouštěla. Kdokoli se na ně podíval, odhodil všechny věci a s křikem na rtech utíkal, seč mu nohy stačily. Být temným mágem má i své stinné stránky. Například se nikoho nemůžete zeptat na cestu, protože než dokončíte otázku, dotyčný je dávno pryč a ještě přitom křičí nesrozumitelná slova.

Na náměstí se šikovaly stráže a jejich kapitán právě vysílal poštovního holuba.

20-2-3 Morseovkabezoddělovačů — 10 bodů

Interní vojenská sdělení se zásadně šifrují morseovkou. Kapitán ovšem ve zmatku zašifroval zprávu tak, že mezi písmeny a slovy zapomněl dělat oddělovače. Poštovní holub má namířeno do sousedního městečka, ve kterém je další vojenská posádka. Otázkou zůstává, jestli se jim vůbec podaří vzkaz rozluštit. Pomůžete jim?

Máte rozluštit text zakódovaný morseovkou bez oddělovačů. K dispozici máte slovník vojenských termínů (tj. slov, která mohou být v textu použita). Vašim cílem je vypsat překlad, který má nejméně slov. Pokud je možných nejkratších překladů víc, vypište libovolný z nich.

Příklad: Slovník obsahuje slova `attack`, `diet`, `indemnify`, `knights`, `pig`, `send`, `sets`, `the`, `zombie` a holub přinesl následující zprávu:

```
....-.-..-.....-.--...--.....-...
```

Správný překlad je: „`send the knights`“. Zprávu je ještě možné přeložit jako „`send diet pig sets`“, avšak takový překlad je o slovo delší, takže to není správné řešení.

Pozn.: Omlouváme se všem češtinářským puristům za příklad s anglickými slovíčky. Bohužel se nám nepodařilo nalézt vhodný a rozumně krátký příklad v naší mateřštině.

Posádka se sešikovala a s rozklepanými koleny zírala, jak uprostřed náměstí zastavil vůz řízený zombií a přitom se ozvalo zlověstné zakrákání, které z celé situace udělalo jedno velké klišé. Z vozu neohrabaně slezla postava celá v černém. Kapitán polkl naprázdno. Teď začnou komplikace. Jednomu ze strážných vypadla z roztřesené ruky halapartna a zařinčela na dláždění.

„Jménem města –," vypravil ze sebe chvějícím se hlasem kapitán, ale mág ho zastavil pozdvihnutou rukou.

„Ale kapitáne, nechcete dělat žádná ukvapená rozhodnutí, že ne?" zašvitořil mág medovým hlasem. „Což takhle probrat vzniklou situaci nad šálkem dobrého černého čaje? A vaši muži si zatím mohou dát pauzu, vypadají trochu napjatě."

Kapitán byl jako omámený. Pomalu se otočil ke své jednotce a zavelel pohov. Pak se i s temným mágem odporoučel do své kanceláře. Na náměstí zůstaly stráže ve velmi nejisté situaci a opatrně pozorovaly vůz s Vildou. Ze sena vyskočil Felix, protáhl si kočičí hřbet a ladným krokem se začal procházet. Přišel k jednomu strážnému, posadil se těsně před něho a naklonil hlavu ke straně. Strážný si ho prohlížel s napjatým výrazem.

„Baf!" pronesl Felix suše.

Na dláždění zařinčelo několik halaparten a celá posádka se ztratila za dupotu sandálů v obláčku prachu.

„Ani se nerozloučili," postěžoval si Felix a začal se mýt.

Temný mág se po chvíli vrátil i s kapitánem, který měl na tváři nepřítomný výraz.

„Dobrá zpráva. Kapitán nám nejen dovolil zůstat ve městě, jak dlouho budeme chtít, ale také byl tak laskav a prozradil mi, kde se nachází místní knihovna."

Knihovna nebyla velká a očividně v ní už dlouho nikdo neuklízel. Po zemi se válely nejrůznější knihy a svitky. Police byly prožrané červotoči a pořádně zaprášené. Zkrátka když jste si při pohledu na tuhle spoušť představili slovo úklid, první, co vás napadlo jako asociace, byl krumpáč.

Mág smutně pokýval hlavou. „Nejprve bychom měli – Felixi!! Co to tam děláš?!"

„Copak můžu za to, že tu není žádná bedýnka s pískem?" opáčil kocour ublíženě. „A navíc je tu stejně nepořádek!"

20-2-4 Slovíčkaření — 10 bodů

Mág s Vildou potřebují uklidit v knihovně. To mimo jiné znamená, že musí setřídit všechny knížky podle názvů. Jak jste si již v našem příběhu zvykli, času nemají hrdinové nazbyt, takže by to potřebovali udělat opravdu rychle.

Máte dánu množinu slov – názvů knížek – a chcete ji setřídit podle abecedy. Všechna slova jsou zapsána pomocí konečné uspořádané abecedy Σ. Její velikost $|\Sigma|$ značme A. Dvě písmenka z abecedy můžete porovnat v konstantním čase,

ale nezapomeňte, že porovnání dvou slov už není konstantní operace, její časová složitost je lineární k délce kratšího ze slov.

Snažte se o co nejmenší časovou složitost nejen vzhledem k počtu slov N, ale také vzhledem k jejich délce (součet délek všech slov si označme P) a k velikosti abecedy (A).

Příklad: Potřebujeme setřídit slova nad českou abecedou: kuchařka, zpěvník, zahradničení a necrotelecomicron. Vzestupně setříděná budou v pořadí: kuchařka, necrotelecomicron, zahradničení a zpěvník.

Uklízení v knihovně jim zabralo pěkných pár dní. Naštěstí mág znal několik užitečných kouzel, která pomohla věci urychlit, a stráže už je vůbec neobtěžovaly.

„Á, konečně jsem to našel," zvolal mág nadšeně a vítězoslavně pozvedl knihu nad hlavu. „Tady se píše, že se musíme vydat směrem . . . "

20-2-5 Hroší ohrádka — 10 bodů

Opět se zde setkáváme nad praktickou úložkou. Způsob odevzdávání a všechny ostatní detaily jsou stejné jako v minulé sérii. Takže pokud jste zapomněli, jak to v praktické úložce chodí, nebo jste se do řešení KSP zapojili teprve teď, podívejte se na úložku 20-1-5 z minulé série, kde naleznete potřebné informace.

Zadání: V jedné nejmenované zoo řeší problém. Přemnožili se jim hroši a je potřeba pro ně postavit nový výběh. Zoo hodlá využít opuštěný výběh po bobří rodince, která emigrovala do zahraničí při poslední povodni. Na pozemku se nachází spousta kůlů, které lze využít jako základ oplocení, avšak samotný plot už je zničený. Zoo se proto rozhodlo oplotit co největší možnou plochu tak, že použije stávající kůly a mezi nimi napne pletivo. Zůstává ovšem otázka, jak to provést – a to už je úkol pro vás.

Váš program dostane na vstupu seznam souřadnic kůlů v souboru `kuly.in`. Na prvním řádku je jediné číslo N, které představuje počet kůlů. Na následujících N řádcích se nacházejí souřadnice jednotlivých kůlů popsané jako dvojice čísel x_i, y_i. Souřadnice jsou celočíselné a vejdou se do 32-bitového integeru.

Nedá příliš přemýšlení, že největší plochu bude mít právě konvexní obal zadaných bodů. Váš program má za úkol tento obal spočítat a vypsat do souboru `plot.out` v následujícím formátu:

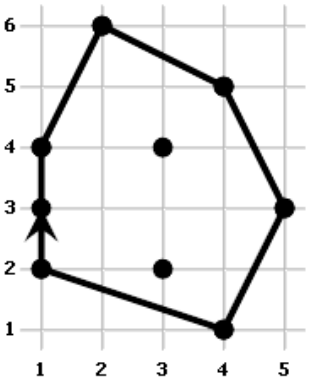
Na prvním řádku bude jediné číslo K. Na následujících K řádcích se nachází seznam hraničních kůlů, mezi kterými je plot natažen. Kůly musí být vypsány ve správném pořadí (tak, jak tvoří plot). Jako první musí být vypsán kůl s nejmenší x-ovou souřadnicí (pokud je takových kůlů víc, vyberte ten, který má zároveň nejmenší y-ovou souřadnici). Pokud si navíc kůly představíme zakreslené v rovině (tak, že x-ová osa roste doprava a y-ová roste vzhůru), pak

musí být kůly vypsány po směru hodinových ručiček. Pro bližší pochopení se podívejte na následující příklad.

Příklad:

kuly.in

```
9
3 2
1 2
4 1
3 4
1 3
1 4
4 5
2 6
5 3
```



plot.out

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

Hint: Nepoužívejte reálná čísla či složité matematické funkce. Souřadnice jsou celočíselné, takže si celou dobu vystačíte s celými čísly. Pokud nevěříte, vezměte si k ruce matematicko-fyzikální tabulky (nebo podobnou literaturu).

Pozn: Můžete předpokládat, že obsah konvexního obalu je vždy nenulový. Tj. nenastane případ, že by všechny kůly ležely na jedné přímce.

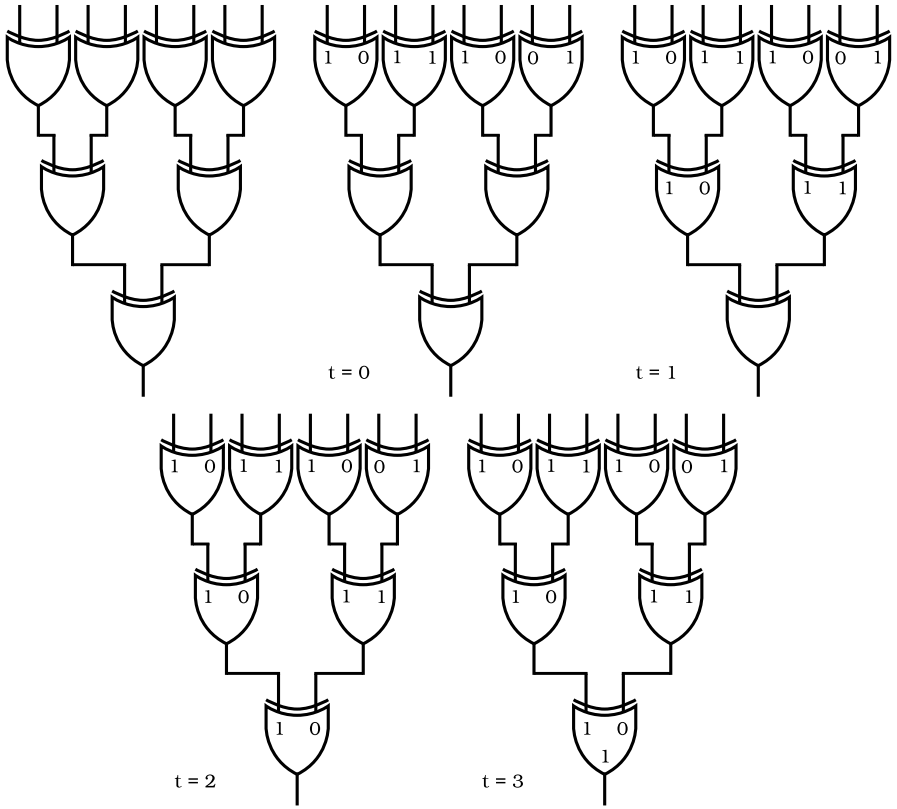
20-2-6 Hrady, hrádky, hradla — 11 bodů

Milí řešitelé, minule jsme si povídali o věcech takřka tajemných a nahlédli jsme do vnitřností vašich počítačů. Zjistili jsme, jak fungují ty nejmenší části mikroprocesoru, a pokusili jsme je alespoň trochu ovládnout. V tomto dílu našeho seriálu se spolu podíváme, jak rychle a efektivně naše obvody vlastně fungují.

Rychlostí zde míníme funkci závislosti času stráveného výpočtem, na velikosti vstupu. Pokud vám to připomíná časovou složitost algoritmu, jste na správné stopě. Stačí nám říct, že doba průchodu signálu hradlem, tj. doba, za kterou se po připojení signálů na vstup hradla nastaví na jeho výstupu správný výsledek operace, je jednotková. Poté si můžeme přestavit, že výpočet probíhá po hladinách. Pod pojmem „hladina“ si nemusíme představovat nic složitého, je to několik hradel, ke kterým výpočet dorazí ve stejné chvíli. Výpočet bude tedy probíhat tak, že v čase $t = 0$ se na vstup zapojení nastaví vstupní data. V čase $t = 1$ budou výsledky na výstupech hradel, která počítají výsledek ze vstupních dat. V čase $t = 2$ budou výsledky na výstupech hradel, která počítají výsledky z hradel předchozích dvou hladin a tak dále, až se spočítají výsledky na poslední hladině, která vydá konečný výsledek celého obvodu.

Abychom si ještě lépe rozuměli, připravili jsme si pro vás jednoduchý obvod, který zjišťuje, zda je na vstupu lichý či sudý počet jedniček. Na obrázku je

nakreslen pro $n = 8$:



Zapojení funguje jednoduše: počítá si pro každou dvojici čísel ze vstupu, zdali je v ní počet jedniček lichý nebo ne. Následně provádí stejný výpočet, ovšem pro již vypočítané mezivýsledky, kterých je jenom polovina, a tak dále, až všechny mezivýsledky spojí do jediného výsledku. Takto přímočaře to sice funguje jen tehdy, je-li velikost vstupu mocnina dvojky (tedy $n = 2^k$ pro nějaké k), ale i kdyby nebyla, můžeme si vstup doplnit nulami na nejbližší vyšší mocninu dvojky, čímž výsledek nezměníme a n maximálně zdvojnásobíme.

Hladiny obvodu odpovídají „patrům“ na obrázku. Na první hladině leží $n/2$ hradel, na druhé $n/4$, ..., na i-té $n/2^i$, až na k-té jich je $n/2^k = n/n = 1$. Čas strávený výpočtem proto je $k = \log_2 n$. Naše zapojení tedy má logaritmickou časovou složitost.

Výrazem *efektivně* zde míníme hlavně spotřebu energetickou. Každá součástka v obvodu spotřebovává energii. Pro naše účely budeme předpokládat, že

hradlo spotřebovává jednu jednotku energie, tj. všechny typy hradel jsou stejně náročné. Bude nás tedy zajímat funkce vyjadřující závislost počtu hradel na velikosti úlohy. Opět si můžeme všimnout, že je daná definice skoro povědomá a může připomínat definici paměťové složitosti algoritmu.

Spočítejme si hradla v našem příkladu po hladinách od nejnižší k nejvyšší. Celkem jich je $1 + 2 + 4 + \ldots + n/4 + n/2 = \sum_{k=0}^{\log_2 n - 1} 2^k$. Součet řady můžete zjistit přímočaře ze vzorce pro součet geometrické řady. Napovíme, že se vám bude hodit vztah $x = a^{\log_a x}$. Pokud vám vyjde složitost lineární, a to $n - 1$, došli jste k správnému výsledku.

Tady naše povídání končí a abyste i vy dostali svůj prostor, následuje zadání úložek:

1. Navrhněte obvod, který bude počítat „hromadný OR“. Tedy trochu formálněji: na vstupu máme n bitů a na výstupu máme jednu hodnotu, která je 1 právě tehdy, je-li mezi vstupními bity aspoň jedna jednička. [4 body]
2. Dokažte, že vaše řešení první úlohy je nejlepší možné, čili že lepší časové ani paměťové složitosti už dosáhnout nejde. [7 bodů]

„Á, konečně jsem to našel,“ zvolal mág nadšeně a vítězoslavně pozvedl knihu nad hlavu. „Tady se píše, že se musíme vydat směrem k Lávové hoře.“

Mág položil knihu na stůl a už se chystal provést složité kopírovací kouzlo, když se do věci vložil Felix: „Tak to vám tedy opravdu nezávidím. To budete mít docela štreku.“

„Kráá?“ ozval se nechápavě Kiri.

„Hm, na tom něco bude. Lávová hora není zrovna za rohem a létat na koštěti, to už na mě není. Průvan mi nedělá dobře na klouby.“

„No vidíte! Já říkal hned, že jsme měli zůstat doma,“ pochvaloval si kocour náhlou změnu plánu. Mág mu ale hbitě zkazil náladu: „Vildo, zajdeš na trh a pořídíš koně. Já tu zatím zkopíruji tu knihu, ať můžeme vyrazit ještě dnes!“

Vilda zašel k místnímu handlíři. Mezi dveřmi mu kývl na pozdrav a hned přešel k věci: „Rád bych si pořídil jednoho koně. Nemusí jezdit moc rychle, ale musí vydržet dlouhou cestu. Jo, a taky aby se neplašil, pokud uvidí něco neobvyklého.“

Handlíř vypadal velmi vyděšeně při pohledu na nazelenalého zákazníka. Byl by vzal nohy na ramena, ale z jedné strany mu v tom bránila zeď a mezi dveřmi stál Vilda.

„T-t-třeba t-t-tohohle?“ zakoktal handlíř, když se mu vrátila řeč a ukázal přitom prstem na krásného bělouše.

„A v černé by nebyl?“

„B-bílý táhne první. To je výhoda, ne?“ vyblekotal handlíř první argument, který mu přišel na mysl. V tu chvíli si ale Vildy všiml i bělouš, vzepjal se na zadní a málem tak převrhl koryto s vodou.

„Máte pravdu, tento není ten pravý," změnil handlíř názor.

Ale jak se ukázalo, žádný z přítomných koní nebyl dostatečně klidný, aby snesl pohled na Vildu. „Možná bych to mohl ještě stihnout, než . . . " zamumlal si handlíř pod vousy a zmizel v přilehlém uzenářství. Za chvíli se vrátil a táhl za sebou něco, co budeme z nedostatku lepší terminologie nazývat koněm.

Kůň nevypadal, že by snesl cestu třeba jen za město, ale Vildův vzhled mu zřejmě nevadil. Dokonce se mu přiblížil natolik, že mu dokázal vytáhnout z kapsy svačinu.

„Ale vždyť je slepý na jedno oko!" všiml si Vilda.

„Ano. A ani za to nebudu chtít příplatek."

„Prosím?"

„Když uvidíte něco opravdu neobvyklého, prostě okolo toho projdete zleva."

Netrvalo dlouho a obchod byl uzavřen. Ale když došlo k placení, nastal problém. Cena nepředstavovala žádné potíže, na té se shodli k oboustranné spokojenosti. Zato se ukázalo, že vybrat správné mince je těžký úkol.

20-3-1 Platba koně — 12 bodů

Je zadán obnos peněz P, požadovaný za koně, a dvě hromádky mincí (jedna Vildova a jedna handlířova). V každé hromádce mincí jsou mince různých hodnot a od každé hodnoty může být více mincí. Na Vildově hromádce je celkem N mincí, na handlířově M. Každá hodnota je desetinné číslo s přesností na 2 desetinná místa. Vaším úkolem je nalézt způsob, jak zaplatit přesně P na co nejmenší počet vyměněných mincí, nebo zjistit, že to provést nelze.

Poznámka: Hodnoty mincí nemusí tvořit žádný systém běžných platidel ani být jinak „rozumné".

Příklad: Pro $P = 15{,}00$, Vildovu hromádku obsahující dvě mince o hodnotách 20,00 a 45,73 a handlířovu hromádku s mincemi o hodnotách 5,00 a 15,00 je správné řešení takové, že Vilda použije minci o hodnotě 20 a handlíř mu vrátí 5. Pro $P = 16{,}16$ a stejné hromádky mincí řešení neexistuje.

Naložili koně, mág se nechal vysadit do sedla a vyrazili na cestu. Cestovali dva dny, když se v křoví vedle cesty ozvaly hlasy. Družinka se zastavila a všichni pozorně naslouchali.

„Dneska byl nějaký nesdílný. Asi mi už nevěří," stěžoval si první hlas.

„Taky? Prý máme přepadat u severní cesty. Ale číslo stromu mi neřekl. Za prvním se nedá schovat, ale od druhého až k devadesát devátému je to stále ještě hrozně moc možností," rozumoval někdo další.

První si přisadil: „Hm, mně řekl jen součet čísel stromů."

„Mně součin. Ale to mi moc nepomůže, z toho se nedají určit," kňoural druhý.

„I z toho součtu mi bylo hned jasné, že součin nikomu k ničemu nebude," souhlasil první.

„To ti z toho bylo jasné? Tak to já už vím, která čísla to jsou! Jdu si pro luk," zajásal druhý.

„Vážně? Tak v tom případě to vím také, kam jen jsem dal meč?" pookřál první.

Mág sesedl z koně a celá družinka se ukryla za nedalekým kamenem. Sotva se jim podařilo přimět koně, aby si lehl, vyskočili lupiči na cestu.

„Už odešli. Tak bychom mohli jít také," prohlásil Felix, kterému se celá záležitost vůbec nelíbila.

„Ano, ale potřebujeme jít okolo nich," zaškaredil se Vilda.

„Vildo, ty jsi nezabalil čaj!" postěžoval si mág, když se prohraboval sedlovou brašnou. „No, co se dá dělat. Vypadá to, že je budu muset proměnit v ježky."

„Stejně tak, jako všechno ostatní, co se u každého stromu jen pohne?" neodpustil si rýpnutí Felix. Kiri při představě proměny zděšeně opustil strom, na kterém seděl, a s mírným žuchnutím přistál vedle koně.

„No tak ježků je tu stejně málo, tak by to tak moc nevadilo. Ale alespoň bych si ušetřil práci, kdybych věděl, za kterými stromy jsou . . . "

20-3-2 Dva lupiči — 8 bodů

Úkolem této úlohy je najít všechny možné dvojice stromů, za kterými mají lupiči čekat. Nestačí řešení pouze najít, je třeba také zdůvodnit, proč je správné a proč žádné další dvojice neexistují.

Pozn.: předpokládejme, že lupiči jsou dokonalí matematici (mají cvik v počítání lupu) a jsou z takových informací opravdu schopni zjistit svá čísla stromů v podstatě okamžitě. Tedy prohlášení: „Tak to já už vím, která čísla to jsou," není myšleno ironicky, ale opravdu znamená, že z dostupných informací dokáže odvodit ona čísla.

Pro případ, že by vám nějaké drobnosti utekly, uvedeme zde všechny informace ještě jednou: Velitel lupičů si myslí dvě celá čísla mezi 2 a 99. Lupiči A prozradí jejich součin, lupiči B prozradí jejich součet. Navíc A ví, že B zná součet, jen neví, kolik to je (a naopak). Rozhovor pak probíhá následovně:

A: „Nevím, jaká čísla si myslí náš velitel."

B: „Věděl jsem, že to nebudeš vědět."

A: „Opravdu? V tom případě už vím, co je to za čísla."

B: „V tom případě to vím taky."

Felix opovržlivě přičichl ke zlatě, které vyplašení ježci nechali za sebou při zoufalém úprku. Ani tolik spěchat nemuseli, mezi našimi hrdiny nebyl nikdo ochotný běhat a mág prohlásil, že bodliny na zádech jako jejich trest zcela stačí.

„Tak to bychom měli," pochvaloval si mág potěšen, že se mu tak složité kouzlo podařilo.

„A dej si patentovat tenhle způsob výroby ostnatého drátu," navrhl Felix, který si při pozorování zlata všiml žížaly, která se nemohla zahrabat, protože měla po celém těle bodliny.

„Tomu se říká aktivní obranný systém. Teď ji alespoň nesezobne kos," vysvětlil mág a začal listovat v knize. „Utáboříme se tu na noc," dodal po chvíli čtení.

Vilda rozdělal oheň, vybalil deky a uvařil večeři. Noc uběhla rychle a až na vzdálené dupání ježků nic nerušilo mírumilovné ticho. Vildu ráno probralo sluníčko a rosa.

„Vaše Mágstvo, kam pokračujeme dál?" zeptal se Vilda, sotva si protřel oči. Podíval se na mága a znejistěl: „Vždyť vypadáte jako kdybyste celou noc nespal!" A opravdu. Mág měl červené oči, které držel sotva otevřené. Seděl u ohniště a okolo sebe měl rozloženou hromadu papírů popsanou divnými symboly.

„Uáááh, cožeto?" zívl ospalý mág. „Aha, cesta. To je právě ten problém. Tady v knize sice píšou, že musíme najít Svítící bažinu. Ale celé je to popsané těmi starodávnými hádankami, takže nemám ponětí, kam dál."

Hádanka Vildovi tak tajemná nepřipadala. Stálo v ní: „Cesta nejdelší – stále z kopce, jen osmkrát vykročit nahoru. A na konci cesty té, stoč se vpravo k obzoru."

„A já nevím, jak takovou nejdelší cestu, co vede stále z kopce, jen osmkrát kousek do kopce, najít. A věštění nepomohlo," postěžoval si smutně mág.

„To se divím. Posledně se ty vosy vykouřit podařilo," ohodnotil jeho věštecké schopnosti Felix.

Kiri se probudil a slétl z větve, na které spal. Dopadl doprostřed papírů a vytvořil v nich malý kráter. Zoufale krákal a máchal křídly, aby se z té hromady dostal. Vzduch se naplnil poletujícími papíry a havraním peřím.

Vilda se natáhl, aby havrana vytáhl, když v tom mu přímo na nose přistál papír s nějakými čísly. Byly na něm vypsané výšky jednotlivých bodů na cestě.

„Ale vždyť stačí vybrat nejdelší takovou posloupnost čísel na tomhle papíru," zaradoval se a začal počítat.

„Kdybych nebyl tak ospalý, tak bych na to přišel sám ..." zamumlal mág, když usínal.

20-3-3 Cesta z kopce — 8 bodů

Je zadána posloupnost A obsahující N čísel a nějaké číslo k. Úkolem je nalézt indexy a, b ($a \leq b$) takové, že $b - a$ je největší možné a posloupnost $A_a, \ldots, A_b$ obsahuje maximálně k dvojic těsně po sobě jdoucích prvků A_i, A_{i+1}

takových, že $A_i < A_{i+1}$. Pro všechny ostatní dvojice platí $A_i \geq A_{i+1}$. Tedy až na k míst je posloupnost nerostoucí.

Příklad: Pro $k = 1$ a posloupnost $9, 7, 6, 4, 8, 6, 3, 9, 1$ je správným výsledkem $a = 1$ a $b = 7$, tj. podposloupnost $9, 7, 6, 4, 8, 6, 3$.

„Tady to vypadá strašidelně," postěžoval si Vilda, když se blížili k bažině.

„Samá voda. Alespoň nějaká lávka, kdyby tu byla," remcal Felix.

„Lávka tu sice není, ale jsou tu provazové mosty. A kolik," pochvaloval si mág.

A opravdu. Mezi stromy vedly stovky provazových mostů. Mág ze své kopie knihy vytáhl mapu Svítící bažiny a s pomocí ostatních se vydrápal na nejbližší most.

Procházeli bažinou křížem krážem, od jednoho stromu k druhému. Felix několikrát navrhl, že na ně počká na té či oné křižovatce, než se na ni zase vrátí.

„Rozhled na rašelinu bude z tamté křižovatky sice nádherný, ale klidně si ho nechám ujít."

Když i mágovi došla trpělivost a konstatoval unaveně: „Ta mapa je špatně. Tady už jsme byli nejméně šestnáctkrát a na tamté křižovatce třináctkrát. Bloudíme mezi nimi stále tam a zpět. Mezi támhletou zatracenou křižovatkou a tímhle prokletým ztrouchnivělým stromem vede určitě nejvíce cest v celé bažině. Ale nemůžu je najít na mapě."

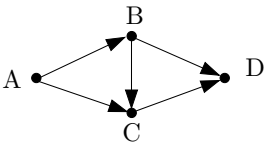
„To bude tím, že ta mapa je pěkně stará. Provazové mosty zůstaly natažené mezi stejnými stromy, ale tohle je bažina a ty stromy v ní nedrží. Prostě se od té doby přemístily," mudroval Vilda a strčil do nejbližšího stromu. Strom se posunul o dobrý metr.

„V tom případě stačí zjistit, mezi kterými vede nejvíce různých cest. Z toho pak dokážu zorientovat mapu . . . "

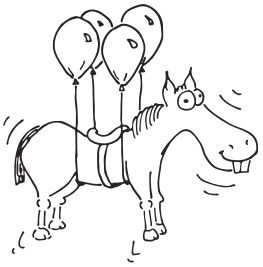
20-3-4 Orientace na mapě — 10 bodů

Na vstupu váš program dostane popis orientovaného grafu znázorňujícího mapu. Víte, že tento graf neobsahuje žádný orientovaný cyklus, čili že neexistuje žádná orientovaná cesta délky alespoň 1, která by začínala a končila ve stejném vrcholu. Úkolem programu je vypsat dvojici vrcholů, mezi kterými vede nejvíce různých cest v celém grafu. Za různé jsou považovány libovolné dvě cesty, které se liší alespoň jednou hranou. Pokud je takových dvojic vrcholů více, stačí libovolná z nich.

Příklad: Pro tento graf je řešením dvojice vrcholů A a D:



Konečně vylezli z bažiny na správnou stranu. Zajásali, rozvázali nebohého koně a mág z něho sňal levitační kouzlo. Vyškrábali se na nejbližší kopec a rozhlédli se po okolí. V dálce spatřili dýmající horu zahalenou v temných mracích. Lávová hora byla konečně na dohled ...



20-3-5 Asfaltování — 11 bodů

Blíží se zima a CodEx už se těší na vaše dárečky k Vánocům v podobě řešení úlohy třetí série. Způsob odevzdávání a všechny ostatní detaily zůstávají stejné, jako v minulých sériích. Takže pokud jste zapomněli, jak to v praktické úložce chodí, nebo jste se do řešení KSP zapojili teprve teď, podívejte se na úložku 20-1-5 z první série, kde naleznete potřebné informace.

Zadání:

Obyvatelé Hipopotámie si již dlouho stěžovali na nekvalitní silnice. Když si jednou i vrchní cestář při cestě do práce v kočáře vyrazil zub, rozhodl se k radikální akci. Doslechl se, že v sousední zemi začali na cesty používat novinku zvanou asfalt, a myšlenka na vyasfaltování všech silnic v Hipopotámii byla na světě. A jak si vrchní cestář usmyslel, tak se i stalo. Při realizaci nápadu se ale nižší cestáři museli potýkat s nemilým problémem: asfalt se dovážel ze sousední země v ohromných barelech, ve kterých bylo asfaltu tak akorát na dvě cesty (byly to opravdu ohromné barely). Potíž byla v tom, že jakmile se barel narazil a asfalt z něj začal vytékat, nedal se už proud asfaltu ničím zastavit a bylo tedy nutné vyasfaltovat najednou dvě na sebe navazující cesty. Jenže jak rozvrhnout asfaltování, aby cestáři neskončili ve městě s poloplným barelem a se všemi cestami vedoucími do města už vyasfaltovanými? Důsledky zalití náměstí a několika přilehlých čtvrtí do asfaltu by byly pro cestáře jistě nemilé...

Napište program, který pro zadané propojení měst cestami rozhodne, zda je možno cesty podle popsaných pravidel vyasfaltovat, a pokud ano, vypíše jednu z možností, jak rozvrhnout, která dvojice cest bude asfaltována ze kterého barelu.

Na prvním řádku vstupního souboru `asfalt.in` se nacházejí dvě celá čísla N a M oddělená mezerou ($1 \leq N \leq 10000$ a $1 \leq M \leq 40000$), kde N určuje počet měst a M počet cest v Hipopotámii. Dále ve vstupním souboru následuje M řádků popisujících jednotlivé cesty. Každý řádek obsahuje dvě celá čísla A a B oddělená mezerou – čísla měst (města číslujeme od jedné do N), mezi kterými cesta vede. Předpokládejte, že mezi každými dvěma městy se lze po cestách dostat (pokud ne přímo, tak přes jiná města).

Výstupní soubor `asfalt.out` bude buď obsahovat jediný řádek s textem `no`, pokud neexistuje způsob, jak vyasfaltovat všechny cesty a neskončit s poloplným barelem v nějakém městě, nebo bude obsahovat $M/2$ řádků s popisem postupu asfaltování. Každý řádek postupu bude popisovat využití jednoho barelu s asfaltem, tj. bude obsahovat tři celá čísla oddělená mezerou, která představují po řadě: číslo města, ve kterém má asfaltování začít, číslo města, do kterého se má pokračovat a číslo města, ve kterém má asfaltování skončit.

Nezapomeňte, že každá cesta má být vyasfaltována právě jednou!

Příklad 1:

`asfalt.in`
```
3 3
1 2
2 3
3 1
```

`asfalt.out`
```
no
```

Příklad 2:

`asfalt.in`
```
5 8
1 5
1 4
1 3
1 2
3 2
3 4
4 5
4 2
```

`asfalt.out`
```
5 1 4
5 4 3
4 2 3
3 1 2
```

20-3-6 Hrady, hrádky, hradla — 12 bodů

Milí řešitelé, nyní už víme, jak vlastně obvody fungují, jak jsou rychlé a energeticky náročné. V dnešním pokračování našeho seriálu si povíme něco o reprezentaci dat v binární podobě, tj. pomocí jedniček a nul.

Reprezentací dat zde myslíme nějaký způsob, jakým převést čísla nebo i jiné údaje do nul a jedniček, se kterými už umíme pracovat. Nejednodušší způsob, jak to udělat je dvojková neboli binární číselná soustava. Čísla se pak mezi desítkovou a dvojkovou soustavou převádějí následovně:

Dvojková → Desítková

Vezmeme číslo zapsané ve dvojkové soustavě a budeme postupně brát cifry po jedné zprava doleva. Každou z nich vynásobíme dvojkou umocněnou na počet cifer, které jsou od aktuální cifry vpravo. Pro číslo 101010 to bude vypadat následovně: $0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 = 2 + 8 + 32 = 42$.

Jelikož $a_0 \cdot 2^0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + \ldots + a_n \cdot 2^n$ lze přepsat jako $a_0 + 2(a_1 + 2(a_2 + 2(\ldots + 2a_n)\ldots)))$, funguje i následující jednodušší převodní algoritmus: začneme nejlevější číslicí, znásobíme ji dvěma, přičteme další číslici, výsledek znásobíme dvěma, a tak dále.

Desítková → Dvojková

Pokud je zadané číslo sudé, zapíšeme si nulu, v případě, že číslo je liché, zapíšeme jedničku a od čísla jedničku odečteme. Pak číslo vydělíme dvěma a postup zopakujeme. Jedničky a nuly budeme zapisovat zprava doleva tak dlouho, až se naše číslo stane nulou. Číslo 13 bychom převáděli takto:

číslo desítkové	*číslo dvojkové*	*výpočet*
13	1	$(13-1)/2$
6	01	$6/2$
3	101	$(3-1)/2$
1	1101	$(1-1)/2$

Všimněte si, že tento postup je vlastně předchozí postup pro převod z dvojkové soustavy spuštěný pozpátku.

Jiný používaný způsob, jak kódovat čísla do jedniček a nul, je kód BCD (Binary Coded Decimal). Myšlenka je následující: Když se rozhodneme trochu plýtvat, klidně můžeme kódovat desítková čísla po číslicích. Přitom na každou číslici použijeme 4 bity, a to tak, že využijeme 10 kombinací z 16 možných. Číslo 1394 bychom zapsali jako 0001 0011 1001 0100. Díky mírnému plýtvání jsme si značně zjednodušili práci, tedy aspoň v případech, kdy čísla daleko častěji rozebíráme na číslice, než s nimi počítáme. (Pěkný příklad jsou třeba digitální hodinky – ty musí umět číslo zobrazovat na displeji, ale jinak stačí umět přičítat jedničku.)

Někdy se také potřebujeme vypořádat s tím, že pokud přenášíme nějaký údaj do jiného obvodu (třeba po počítačové síti, a nebo ho ukládáme na disk), může se cestou trochu poškodit – některé bity se mohou změnit z nul na jedničky nebo naopak. Tehdy se samozřejmě hodí umět chybu odhalit, nebo dokonce automaticky opravit. Zkusme si třeba na konec binárního čísla přidat nulu nebo jedničku tak, aby celkový počet jedniček v čísle byl sudý (tomu se říká paritní bit). Pokud v takovém čísle nastane jedna chyba, ihned ji odhalíme, jelikož vždy vznikne číslo s lichým počtem jedniček. Existuje spousta sofistikovanějších kódů, které dokáží detekovat nebo opravovat větší počty chyb, ale my si už místo dalšího povídání raději naservírujeme nové úložky.

Vaším úkolem nyní bude vymyslet obvod na detekci chyb, který bude testovat dělitelnost třemi (taková kontrola chyb je o něco silnější než parita).

1. Navrhněte a nakreslete obvod, který pro číslo zadané v binární reprezentaci zjistí, zda je dělitelné třemi. [6 bodů]
2. Úkol je stejný, ovšem čísla jsou na vstupu v BCD reprezentaci. [6 bodů]

Ráno opět vyšlo slunce. Ne snad proto, že by muselo, ale prostě už bylo tak zvyklé. Opatrně vystrčilo pár paprsků a začalo vymetat zbytky tmy ze Škytánie. Jeden z paprsků polechtal i spícího kocoura Felixe. Kocour dlouze zívl, protáhl se, a přitom nechtě vrazil tlapku přímo do obličeje spícího havrana.

„Krá! Krá?!“ ozval se Kiri popuzeně, čímž vzbudil zbytek družiny.

„To musíš dělat takový rámus?!“ obořil se na něj mág, sotva si protřel oči. „Jestli mě ještě jednou takhle probudíš, proměním tě v žížalu!“

Havran se zatvářil provinile a na svoji omluvu zakrákal: „Never-r mor-re!“

„Vildo, připrav snídani! Chci vyrazit co nejdřív!“ zavelel mág a vstal. Zadíval se do dálky, kde se tyčila Lávová hora. „Máme před sebou ještě kus cesty . . . “

Sestoupili z pahorku, na kterém nocovali a vklouzli do řídkého lesíku. Cesta ubíhala pomalu a les nenápadně houstl. Po několika dnech už nebylo dál možné cestovat s koněm. Stromy rostly příliš blízko u sebe, všude samé houští a mlází. Přeložili všechny věci na Vildova záda a koně poslali domů.

Následujícího dne večer je přepadla ošklivá bouře. Silný vítr kymácel stromy a hustě pršelo. Každou chvíli oblohu prořízl blesk a promptně vyrobil další hromádku palivového dříví.

„Musíme najít nějaký úkryt!“ křičel mág do ohlušujícího větru a ze všech sil se snažil udržet na nohou.

„Myslím, že jsme před chvílí procházeli kolem jeskyně! Můžeme se zkusit schovat tam!“ zahulákal Vilda a rukou chytil Kiriho, který začal prohrávat svůj boj se vzdušnými víry.

Z posledních sil se doplahočili do jeskyně. Sedli si na kamenné podloží, aby popadli dech, a jak už to tak v příbězích bývá, sotva se posadili, oslepil je záblesk doprovázený obrovskou ránou. Vysoká borovice, která stála kousek od vchodu do jeskyně, byla rozpůlená ve dví a z jejího středu se lehce kouřilo.

„Předpokládám, že teď bych měl říct něco jako: ’To bylo o fous!’“ dodal sarkasticky Felix. „Proč jsme sakra nezůstali doma?!“

Vilda začal vybalovat věci a chystal se rozdělat oheň. Vyndal z batohu troud, jehož čvachtavé plesknutí o kámen oznamovalo všem, že rozdělat oheň bude přece jen trochu těžší, než by se snad mohlo zdát. Mág si povzdechl a seslal doprostřed jeskyně kouzelný plamen.

„To by mělo na chvíli stačit,“ prohlásil spokojeně a šel se usušit.

Když se trochu zahřáli, všimli si, že z jeskyně vede několik chodeb. Navíc stěny byly popsány zvláštními značkami. Většina značek byla vodorovná nebo svislá. Mág si je se zájmem prohlížel a pak se ťukl do čela.

„To je přece písmo lesních druidů!“ zvolal vítězoslavně a začal se přehrabovat ve svém vaku. Vytáhl tlustou knihu. Kniha byla opravdu . . . tlustá. Mág se na ni podezřívavě podíval. Z knihy ukápla kapka vody.

Po hezky dlouhé chvilce strávené rozlepováním listů a sušením nacucaného papíru nalezl mág konečně tu správnou stranu a začal s překladem značek na zdi.

20-4-1 Druidí nápisy 10 bodů

Mág se s problémem vypořádá jako vždy po svém, ale my obyčejní smrtelníci, kteří neovládáme magii (nebo alespoň ne tak dobře jako Temný mág), si zadání trochu zformalizujeme. Druidí text je zadán jako posloupnost svislých a vodorovných značek bez jakýchkoli mezer, takže si jej lze představit jako posloupnost bitů. K dispozici máme překladovou tabulku, která každému písmenu přiřazuje sekvenci bitů (můžete předpokládat, že všechna písmena jsou reprezentována sekvencemi s maximální délkou řekněme 5 bitů). Dále máme slovník všech možných slov, která se mohou v textu vyskytnout.

Chtěli bychom text přeložit, avšak problém je v tom, že překlad může být mnohoznačný. Naším cílem tedy bude najít celkem K nejkratších možných překladů textu. Délka překladu je udávána počtem slov (tj. chceme překlady s nejméně slovy). Číslo K není konstanta, takže ho nezapomeňte zahrnout do odhadu časové (případně i paměťové) složitosti.

Poznámka autora: Pokud vám tato úloha připomíná úlohu 20-2-3 (Morseovkabezoddělovačů), tak tato podobnost není čistě náhodná. Jen jsme nechtěli, aby se nám v příběhu znovu opakovala morseovka. V této úloze máte téměř stejný úkol, ale nyní musíte najít K nejkratších řešení (ne pouze jedno). Necítíte-li se ve své kůži při pomyšlení na druidy, klidně pracujte s morseovkou.

„Výborně! Už jsem to rozluštil," zajásal mág. Všichni ostatní se k němu seběhli a tázavě se zadívali na stěnu. „Zkusím to volně přeložit do naší řeči," oznámil a nakrabatil čelo při usilovném soustředění.

„Vstoupili jste – na posvátnou půdu – druidů. Za vaši – troufalost ... éé ... zahnete? To asi nebude ono. Že by 'zvadnete'?" Mág nespokojeně kroutil hlavou a chaoticky listoval knihou. „Néé, už to mám – 'zahynete'! Za vaši troufalost zahynete!" rozzářil se spokojeně, sklapl knihu a pohladil si plnovous s výrazem triumfu na tváři. Ostatní na něj zírali v němém úžasu. Nastala trapná chvíle ticha. Mágův úsměv se pomalu přeskládal do výrazu panické hrůzy.

„Myslíte, že bychom měli ...," zašeptal mág a kývl hlavou směrem k východu z jeskyně.

„Krá!" přitakal Kiri za všechny přítomné.

Rychle naházeli věci do vaků a mág lusknutím prstů uhasil oheň. Venku stále ještě zuřila bouřka, ale nic jiného jim nezbývalo. Vykročili do hustého deště. K jejich nemilému překvapení stál okolo vchodu do jeskyně početný hlouček postav. Všechny byly oblečeny do sněhobílých rouch. Tohle zvláštní oblečení zřejmě nesledovalo předpověď počasí, protože navzdory provazům vody padajícím z nebe bylo dokonale suché. Jedna z postav pozvedla ruku a bouře v jediném okamžiku ustala.

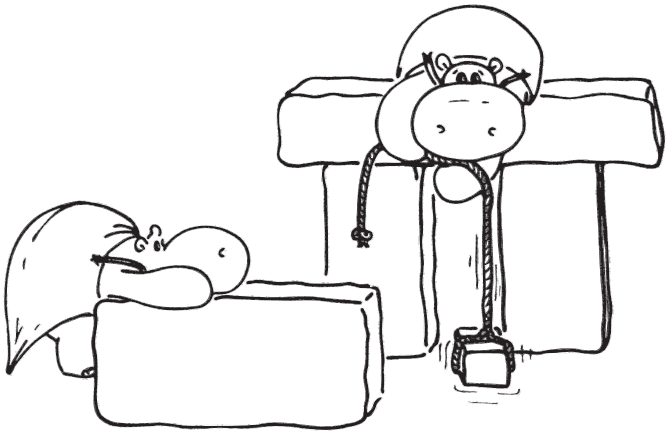
„A do kočičince ...," ulevil si Felix.

Druidi je vedli lesem dobrou hodinu a každou chvíli měnili směr. Dorazili na rozlehlou louku. Uprostřed stála rozestavěná jakási kamenná konstrukce a kolem ní se hemžili další druidi. Nedaleko bizardní stavby stál statný dub, pod kterým byly rozloženy stoly s nejrůznějším nářadím a nákresy. Došli až k dubu a tam se zastavili. Zanedlouho k nim přispěchal starší druid. Roucho měl úplně stejné jako ostatní, ale i přesto z něho na první pohled sálala autorita. Cestou k nim se několikrát zastavil a rozdal několik rozkazů.

„Zdravím vás," promluvil k nim neutrálním tónem. „Jmenuji se Rozmarýn a jsem tady Velkým druidem. Už vás nějakou dobu sledujeme. To víte, když Temný mág opustí Temný hvozd, asi má něco za lubem a je potřeba na něho dávat pozor," dodal s lehkým úsměvem. Mág se nesmál. „Každopádně bych chtěl říci, že kdybyste byli kdokoli jiný, nedošli byste živí ani do té jeskyně. Ale než vás propustíme a vyprovodíme z našeho území, budeme po vás chtít malou výpomoc . . . "

20-4-2 Stonehedge — 8 bodů

Druidi staví Stonehedge, což by se dalo přeložit jako „Kamenný plot". Ovšem není to obyčejný plot – je to spíše brána. A přímo do jiného světa. Do technických detailů nebudeme příliš zabíhat, protože je pořádně nepochopil ani mág po Rozmarýnově vyčerpávajícím výkladu. Podstatné je, že druidi mají seznam rohů kamenů, které je potřeba vytesat a správně umístit, a potřebují z nich vytvořit stavební plán.



Každý kámen má pravoúhlou základnu. Tedy jeho půdorys je mnohoúhelník, který má každou stranu rovnoběžnou s některou ze světových stran (v kartézské rovině bychom řekli rovnoběžnou s osami x nebo y). Půdorys kamene je ovšem popsán pouze souřadnicemi rohů (tj. vrcholy polygonu). Záznamy jsou rozházené, takže souřadnice rohů jsou zapsány v náhodném pořadí. Naštěstí ale

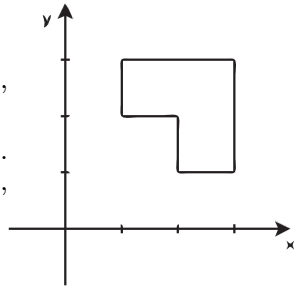
víme, že kameny v sobě nemají díry (tzn. všechny rohy leží na obvodu kamene) a jejich souřadnice jsou celočíselné.

Vašim úkolem je vzít záznamy o každém kameni a zjistit, jaký bude mít půdorys. Výsledkem vašeho snažení by měla být pro každý kámen uspořádaná posloupnost vrcholů (rohů). Jako první můžete uvést libovolný roh a všechny ostatní vypíšete v pořadí, v jakém se vyskytují na obvodu kamene po směru hodinových ručiček.

Příklad:

Máme jeden kámen s vrcholy $[1,3]$, $[2,2]$, $[1,2]$, $[3,1]$, $[3,3]$, $[2,1]$.

Začneme např. v prvním uvedeném bodě $[1,3]$. Dále pokračujeme po směru hodinových ručiček: $[3,3]$, $[3,1]$, $[2,1]$, $[2,2]$, $[1,2]$.



Bylo ráno. Naši hrdinové v klidu podřimovali, až na mága, který stál nad velkým kusem papíru a občas do něho něco zakreslil.

„Tak to by měl být poslední kámen," řekl, odložil tužku a promnul si unavené oči. Rozmarýn se zadíval do náčrtků a spokojeně pokýval hlavou.

„Děkuji vám. A teď, když mě omluvíte, musím jít dohlédnout na stavbu. Tady bratr Levandule vás doprovodí až na hranice našeho území."

Mág nakrabatil čelo: „Ehm – víte, že . . . jaksi . . . slovo Levandule je ženského rodu?" Levandule na něho vrhl ošklivý pohled. Rozmarýn jen pokrčil rameny: „Nikdo není dokonalý." A odspěchal za svými povinnostmi.

Cesta ubíhala . . . potichu. Levandule nebyl zrovna velký řečník a až na příležitostné Felixovo remcání se nikdo neodvážil promluvit. Les začal zvolna řídnout, a když se objevila první cesta, Levandule se s nimi mlčky rozloučil.

„Teď by se nám kůň hodil," nahodil Vilda smutně, ale nikdo ho nevnímal.

Pokračovali po cestě až do pozdního odpoledne, když se za nimi ozvalo klapání kopyt a zvuk kodrcajícího se povozu. Za chvíli vůz spatřili na vlastní oči. Byl celkem malý, přikrytý plachtou a táhl ho dobře živený hnědák. Na kozlíku seděl trpaslík a spokojeně bafal z dlouhé fajfky. K všeobecnému překvapení se povedené družinky vůbec nelekl a ještě jim kývnul na pozdrav.

„Brý vodpoledníčko," zahlaholil vesele a zastavil vůz. „Vypadáte, jako bystejc tady něco tó – hledali, nemejlim se?"

Mág se podíval na Vildu a Vilda zase na mága. „Vám na nás nepřipadá něco v nepořádku?" zkusil to opatrně mág.

„Ani né. Tadyc kluk má nějakou nezdravou barvičku, ale pár dní na sluníčku by to spravilo. A ten kocour, vypadá nějakto vopelichaně . . . "

„Já vám dám vopelichaně," obořil se na něj Felix.

„No né, vonoto i mluví! No to mě pověs za kšandy a nalakuj," pokračoval vesele trpaslík. „Tady u druidích zemí – tady už sem viděl lecojs," mávl ledabyle rukou. „Ale prominou mi, ešče sem se nepředstavil. Menuju se Tom. Tom Vytrhnul."

„Já jsem Temný mág, pán Temného hvozdu," oznámil mu vznešeně mág.

„Tak mák, jo?" prohlásil trpaslík s úsměvem. „Mě ešče žádnej mák nepřechytračil! Na mě si jen tak někdo nepříde. Ale že voni sou ten mák, tak já je zkusím. Dám vám malej rébous, a když ho uhodnou, svezu je, kam budou chtít."

„A když ne?" Zeptal se Vilda, ale mág ho okamžitě zpražil pohledem.

„Když ne, tak mi nechaj tadytoho mluvícího kocoura. Mít takovou atrakci, to by se zlaťáčky enom sypaly," usmál se na něho Tom a popotáhl ze své fajfky.

20-4-3 Mince — 7 bodů

Tom vyndal ze svého vozu malý soudek a na něj umístil 2×2 mince. Pak nechal mága, aby si zavázal oči. Mág neví, jak jsou mince na začátku otočeny a má za úkol otočit je všechny lícem vzhůru. V každém tahu si může mince libovolně osahat a libovolné otočit, avšak po hmatu nepozná, jak je která mince otočena, ani jestli jsou dvě mince otočeny stejně. Po každém tahu mu trpaslík otočí soudkem, aby ho pořádně zmátl. Pokud mág dostane všechny mince lícem vzhůru, trpaslík sám hru ukončí (místo toho, aby opět točil soudkem).

Pomozte mágovi najít vyhrávající strategii. A jako obvykle pamatujte, že mág má naspěch, takže by vaše strategie měla být co nejkratší.

Strategií rozumíme deterministický návod, který dovede mága po konečném počtu kroků k vítězství. Nezapomeňte, že počáteční stav je neznámý a vaše strategie musí fungovat vždy (na všechny možné počáteční konfigurace mincí).

Trpaslík zíral na čtyři mince otočené lícem vzhůru a nevěřícně kroutil hlavou: „To nejni možný! Mě ešče žádnej mák nikdy nepřechytračil!"

„A kolik mágů se o to pokoušelo?" nahodil mág a pozvedl obočí.

„No, voni byli první," připustil po chvíli Tom. „No co. Trpasličí slovo je trpasličí slovo. Tak si naskočej . . . "

Cesta na voze uběhla příjemně. Za dva dny stáli nedaleko hory. Země okolo byla teplá a tvářila se výhrůžně. Trpaslík se s nimi spěšně rozloučil a pobídl koně, aby byl co nejdřív pryč od té prokleté hory. Lávová hora sice zlověstně dýmala, ale lávu už hezkých pár let nechrlila. Budeme se muset dostat dovnitř, pomyslel si mág a vykročil po úpatí hory. Jeho družina ho neochotně následovala.

Vyškrábali se sotva sto metrů po úbočí, když mág objevil úzký tunel vedoucí do nitra hory. Byl příliš úzký, aby se jím protáhl člověk, ale na druhou stranu dostatečně široký . . . řekněme pro kocoura. Mág se zamyšleně podíval na Felixe.

„Na to zapomeň! V žádném případě! Ani mě to nenapadne!“ vřískal zděšeně Felix.

O chvíli později se už prodíral mezi vlažnými kameny a hlasitě si na všechno stěžoval, aby měl jistotu, že ho uslyší i všichni venku. Náhle se chodba začala rozšiřovat a vyústila do obrovské místnosti. Tady není něco v pořádku, pomyslel si Felix a mžoural do tmy. Ve tmě se něco pohnulo. Ale ne! To je přeci ...

Pokračování (pravděpodobně už bez Felixe) příště.

20-4-4 Skupinky pro chytré — 10 bodů

V tomto díle mágova dobrodružství se bohužel objevily jen tři zajímavé problémy, které jsme vám mohli zadat jako úložky. Sice bychom si mohli vymýšlet a příběh pozměnit tak, abychom do něho vtěsnali další úložku, ale to už by pak nebyl autentický a pravdivý. Na druhou stranu by bylo škoda, kdybychom vás o úložku ochudili, a proto jsme se rozhodli, že přidáme obyčejný (tzn. čistě programátorský) problém, který nemá s Temným mágem vůbec nic společného. Fanouškům Temného mága se tímto omlouváme a slibujeme, že jim vše vynahradíme v další sérii.

Tak tedy k našemu problému. Budeme pracovat se záznamy lidí. Každý člověk má jméno a IQ (přičemž IQ je celé kladné číslo). Z lidí budeme vytvářet skupinky. Každá skupinka má unikátní ID (identifikační číslo), které je opět celé a kladné. Na počátku máme pouze jedinou prázdnou skupinku s $ID = 1$.

Nad skupinkami chceme provozovat operace:

- `INSERT` – Vloží nového člověka.
- `FIND_BEST` – Nalezne člověka s nejvyšším IQ.
- `DELETE_BEST` – Člověk s nejvyšším IQ odchází za kariérou do zahraničí (odstraníme jej).

Výše uvedené operace dostanou vždy ID skupinky, nad kterou mají být provedeny. Žádná z operací nemodifikuje skupinku, ale místo toho vytvoří skupinku novou, ve které budou uloženy výsledky operace. ID nové skupinky bude nejmenší dosud nepoužité číslo. Pochopitelně operace `FIND_BEST` pouze vrací nalezeného člověka, takže nevytvoří novou skupinku. Skupinky nikdy nezanikají, takže je potřeba si je nějakým způsobem udržovat všechny.

Výsledek každé operace musíte oznámit ještě před tím, než začnete zpracovávat operaci další – tj. nesmíte si operace bufferovat a pak jich provést víc najednou.

Vašim úkolem je navrhnout a popsat vhodnou datovou reprezentaci a jak na ní budou probíhat požadované operace.

Příklad: Jak bylo řečeno, na začátku máme jen jednu prázdnou třídu, jejíž $ID = 1$. Budeme provádět operace:

- `INSERT("Aleš", IQ=130)` do $ID = 1$ vytvoří skupinku $ID = 2$.
- `INSERT("Petr", IQ=110)` do $ID = 2$ vytvoří skupinku $ID = 3$.
- `INSERT("Jana", IQ=140)` do $ID = 1$ vytvoří skupinku $ID = 4$.
- `FIND_BEST` v $ID = 2$ vrátí `"Aleš"`.
- `FIND_BEST` v $ID = 3$ vrátí také `"Aleš"`.
- `FIND_BEST` v $ID = 4$ vrátí ovšem `"Jana"`.
- `DELETE_BEST` z $ID = 3$ vytvoří skupinku $ID = 5$.
- `FIND_BEST` v $ID = 5$ vrátí `"Petr"`, protože Aleše odstranila předchozí operace.

20-4-5 Roboti na útěku — 15 bodů

Vánoce skončily a pod stromečkem jsme nalezli krásnou praktickou úložku. A protože nejsme hamouni, rádi se s vámi o ni podělíme. Způsob odevzdávání a všechny ostatní detaily zůstávají stejné jako v minulých sériích. Takže pokud jste zapomněli, jak to v praktické úložce chodí, nebo jste se do řešení KSP zapojili teprve teď, podívejte se na úložku 20-1-5 z první série, kde naleznete potřebné informace.

Zadání:

Gratulujeme, právě jste se stali majiteli dvou zánovních robotů. Bohužel obchodník, který vám je prodal (za extra výhodnou cenu), jaksi zapomněl zmínit, že oba roboti jsou uvězněni v bludišti. A aby toho nebylo málo, tak každý ve svém. Abyste mohli roboty používat, budete je muset nejprve dostat ven.

Bludiště je reprezentováno čtvercovou mřížkou s očíslovanými políčky, kde $[1, 1]$ je levý-horní (tedy severozápadní) roh. Políčka tohoto bludiště mohou být volně průchozí, nebo na nich může stát zeď. Každý robot má své bludiště i ($i \in \{1, 2\}$). V každém bludišti i se navíc nachází G_i stráží, kde $0 \leq G_i \leq 10$. Stráže se pohybují a roboti se od nich nesmí nechat chytit.

Na začátku každého tahu pošlete svým robotům příkaz (oběma stejný). Příkaz je pouze směr, kterým se má robot pohnout (tzn. sever, jih, východ nebo západ), a oba roboti se posunou o jedno políčko daným směrem. Pokud se robot nemůže pohnout (tj. na cílovém políčku je zeď), zůstane stát na místě. Robot je volný v okamžiku, kdy vyjede ven ze zadaného bludiště, a od té doby jakékoli další příkazy ignoruje.

Stráže se rovněž pohybují na začátku každého tahu – tedy ve stejném okamžiku jako roboti. Každá stráž se pohne o jedno políčko směrem, kterým se právě dívá. Jakmile stráž dorazí na konec své cesty (posune se o jedno políčko méněkrát, než je délka její hlídkovací trasy), udělá čelem vzad a bude se po-

sunovat zpět na políčko, kde začínala. Na počátečním políčku se opět otočí a pokračuje v hlídkování, dokud robot neopustí bludiště.

Hlídkovací trasy stráží jsou navrženy tak, aby žádná stráž nemusela procházet zdí. Trasy několika stráží se mohou protínat, ale jejich pohyb je navržen tak, aby se žádné stráže nikdy nesrazily (tzn. na konci tahu nebudou přebývat na stejném políčku a ani si během tahu vzájemně nevymění políčka). Navíc máte zaručeno, že žádná stráž nebude na začátku obývat stejné políčko jako váš robot.

Stráž chytí robota v případě, že se na konci tahu nachází oba na stejném políčku nebo si během jednoho tahu políčka vzájemně vymění.

Žádné z bludišť nemá rozměry větší než 20×20 a zadaná konfigurace (tzn. bludiště s počáteční pozicí robota a s hlídkovými trasami stráží) je vždy platná (tzn. vyhovuje pravidlům popsaným výše). Vašim úkolem je nalézt nejkratší sekvenci tahů, která dostane oba roboty ven z bludiště, aniž by byli polapeni strážemi. Pokud existuje víc řešení, stačí nalézt jedno libovolné.

Vstup je uložen v souboru `robots.in`. Obě bludiště jsou uložena ve stejném formátu těsně za sebou. Formát jednoho bludiště vypadá následovně:

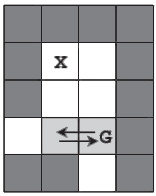
- Na prvním řádku se nachází dvě čísla R_i a C_i oddělená mezerou, která představují počet řádků (R_i) a sloupců (C_i) bludiště i.
- Následujících R_i řádků obsahuje vždy C_i znaků, kde každý znak popisuje jedno políčko bludiště. Znak `#` představuje zeď, znak `.` volně průchozí pole a znak `X` počáteční pozici robota v bludišti.
- Za mapou bludiště následuje řádek s jediným číslem G_i, které udává počet stráží v bludišti. Připomeňme si, že $0 \leq G_i \leq 10$.
- Následuje G_i řádků. Každý řádek obsahuje tři celá čísla a jeden znak oddělené mezerou a popisuje právě jednu stráž. První dvě čísla určují řádek a sloupec počátečního políčka stráže. Třetí číslo určuje délku hlídkovací trasy (v rozsahu $2 \ldots 4$). A konečně poslední písmeno určuje směr, kterým se stráž na začátku dívá (a kterým se také začne pohybovat). Směr je dán prvním písmenem světové strany v anglickém jazyce, tedy: `N`, `S`, `E` nebo `W` (pro **N**orth, **S**outh, **E**ast a **W**est).

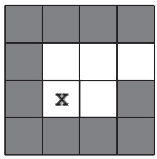
Výstup je očekáván do souboru `robots.out`. Na prvním řádku bude číslo K ($K \leq 10000$), které určuje, kolik tahů bude potřeba, abychom dostali roboty z bludišť. Následujících K řádků obsahuje jednotlivé příkazy v pořadí, v jakém se mají vykonat. Příkazy jsou zapsány písmeny `N`, `S`, `E` a `W`, která určují směr pohybu robotů. Obdobně jako u vstupu jsou tato písmena zkratkami za světové strany v anglickém jazyce. Pokud neexistuje korektní sekvence příkazů, která by dostala roboty z bludiště, pak bude výstupní soubor obsahovat jediný řádek s číslem -1.

Příklad:

Vstup `robots.in`:

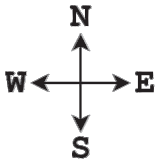
```
5 4
####
#X.#
#..#
...#
##.#
1
4 3 2 W
4 4
####
#...
#X.#
####
0
```





Výstup `robots.out`:

```
8
E
N
E
S
S
S
E
S
```

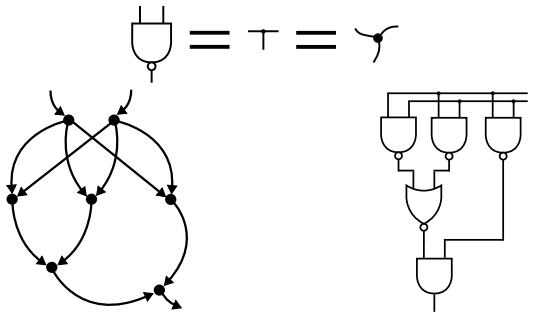


20-4-6 Hrady, hrádky, hradla — 12 bodů

V dnešním díle si budeme povídat o rovinnosti a co to taková rovinnost znamená. Ti z vás, kteří znají pojem rovinnost z teorie grafů, mohou definici rovinného zapojení přeskočit.

Nadefinujeme nejdříve pojem rovinný graf, neboť pro rovinný obvod platí definice analogicky. Definice zní: Rovinný graf (obvod) je takový graf (obvod), který má rovinné nakreslení. Rovinné nakreslení je takové nakreslení, při kterém se žádné dvě hrany (vodiče), při průmětu do roviny, nekříží. V jedné větě: Rovinný graf (obvod) lze nakreslit do roviny bez křížení hran (vodičů).

Na obrázcích je zapojení, o kterém vám prozradím, že je nerovinné, a graf, na který lze převést. Graf vyrobíme tak, že hradla zjednodušíme na vrcholy se třemi hranami. Stejně tak místa, kde spojujeme vodiče. Pro názornost ponechávám hranám orientaci, i když pro určení rovinnosti je veskrze zbytečná.



nerovinný graf *nerovinné zapojení*

Dokázat, že je graf rovinný není jednoduché, lze to například provést tak, že se najde jeho rovinné nakreslení. Opačná věta se dokazuje ještě o krapítek složitěji, zájemce o tuto problematiku nechť se poučí například v knize „Kapitoly z diskrétní matematiky“ od pánů Matouška a Nešetřila.

V praxi je tato problematika velmi důležitá, neboť pro spojování elektronických součástek se používají plošné spoje. Plošný spoj je destička ze dvou materiálů, jako podklad se používá sklotextit, což je destička ze skleněných, nebo podobných vláken slepená plnivem. Tato vrstva je vrstva nosná. Mnohem zajímavější je ovšem vrstva druhá, ta je z mědi. A právě v této měděné vrstvě se leptají cestičky, které spojují obvody. Díky této technologii výroby je důležité minimalizovat křížení, které se pak musí řešit drátovými propojkami. I když ve skutečnosti se nám do zapojení přimíchají navíc cestičky napájecí a problém bude ještě o něco složitější.

Nyní přichází vaše příležitost, máme pro vás tyto dva úkoly.

1. Vymyslete zapojení, které umí „prohodit“ dva signály. Tj. pokud obvod obklopíme kružnicí a půjdeme po průsečících s vstupy a výstupy, dostaneme z původní posloupnosti vstup A, vstup B, výstup B, výstup A posloupnost vstup A, vstup B, výstup A, výstup B. Aby zadání nebylo triviální a v návaznosti na dnešní téma musí takové zapojení být rovinné. [5 bodů]
2. Dokažte, že libovolné zapojení lze převést na rovinné za cenu zpomalení. Ale i tak se snažte toto zpomalení minimalizovat. [7 bodů]

„Domňauhajzlůůůcosetosakratadyseněcopálíí́í. . . “

Z úzké chodby vyletěl ohořelý kocour a těsně za ním vyšlehl plamen. Felix přistál na chladných kamenech venku a tiše doutnal. Přitom vrhal ošklivé pohledy do širokého a dalekého okolí a speciální dávku věnoval samotnému mágovi.

„Je tam drak,“ pronesl suše po nervózní pauze, ve které se každý snažil vyhnout kocourově pohledu.

„Drak," opakoval nepřítomně mág a pohladil si plnovous. „Tak to musí být Ohnivý drak!"

„Nevím, nestačil jsem si všimnout," ušklíbl se sarkasticky Felix a začal si olizovat popálený kožich.

„Ale, vždyť ve Škytánii žádní draci nejsou!" vykulil oči Vilda.

„To nejsou ... ehm – nebyli," ohradil se mág, když spatřil kocourův výraz. „Ale každopádně by mě zajímalo, jak se sem dostal. To musel přiletět až z Velkého pohoří ... "

20-5-1 Dračí cestování — 8 bodů

Dračí cestování není tak jednoduché, jak by se mohlo zdát. Let je pro tvora velikosti malého dopravního letadla namáhavý, a proto se musí drak dobře živit, aby měl dost sil na mávání křídly.

Největší pochoutkou (a zároveň jedinou potravinou, která má dostatečný energetický potenciál) je pro draka síra. Při letu drak spotřebuje 1 kg síry na 1 km.

Síra se bohužel ve Škytánii vyskytuje pouze u Ohnivé hory, takže veškerou síru na cestu si musel drak nést z Velkého pohoří. Délka trasy, kterou musel uletět, je 4200 km. Navíc si drak chtěl přivést co nejvíce síry s sebou, protože nevěděl, jak kvalitní a rozsáhlé budou místní sirné zásoby.

Zjistěte, jaké maximální množství síry si drak mohl přinést, když má nosnost 42 metráků (4200 kg) a ve Velkém pohoří bylo zrovna k dispozici 14 tun (14 000 kg) síry.

Drak si pochopitelně může cestu dělat překladiště síry, kde si kousek nákladu odloží a pak se pro něj vrátí.

„...no ovšem, to je jasné," prohlásil spokojeně mág. „Stačilo, aby si udělal překladiště u Knůllu a pak ještě dvě nebo tři v Troudském lese a měl vystaráno ... "

„To je úplně fuk, jak se sem dostal!" vykřikl Felix. „Podstatné je, že je tady a teď. Viděli jsme, slyšeli jsme a teď bychom mohli takticky ustoupit. Beztak s ním nic nezmůžeme."

„Krá!" přitakal Kiri, který doteď plnil pouze funkci těžítka mágova ramene.

„Máte pravdu, tady nemůžeme zůstat," pokýval hlavou mág. Sotva to dořekl, přeletěl jim nad hlavami obrovský stín. Mág zavřel oči a pozvedl ruce. Čelo se mu nakrabatilo hlubokým soustředěním. Mezi jeho dlaněmi se objevila modrá koule a rychle se zvětšovala. V mžiku pohltila celou družinu a s tichým „pop" zmizela.

O pár set metrů dál se zavlnil vzduch. Ozvalo se opět tiché „pop" následované hlasitým heknutím, když povedená čtveřice dopadla na zem. Mág si otřel čelo a roztřesenýma rukama si přihnul z měchu s vodou, aby se trochu uklidnil.

„Na toho draka sami nestačíme," posteskl si Vilda.

„To si pište, že nestačíte," ozval se jim za zády skřehotavý hlas. Všichni se jako na povel otočili. Stála tam shrbená stařena oblečená celá do černých šatů. Nebyla to temná čerň, kterou by ji mohla závidět i noc. Jen obyčejná všední vybledlá čerň, která o své nositelce prozrazovala maximálně to, že nerada pere. Na hlavě měla špičatý klobouk propíchlý několika jehlicemi.

„Kdo jsi a co tu děláš?" obořil se na ni mág. Žena na něj vrhla ošklivý pohled. Mág luskl prsty a zeptal se znovu: „Kdo jsi a co tu děláš?" a jeho hlas ukapával jako med z včelí plástve.

„Pch," oznámila mu stařena. „Si myslíš, že jsi nějakej mág, nebo co? Že si tady luskneš prsty a všechny tu omámíš? Já jsem Čarodějka! Na mě tyhle laciné triky neplatí!"

„Možná že neplatí, ale i tak jsi mi zodpověděla půlku mé otázky," usmál se mág.

Čarodějka se zamračila: „Kdyby ses pořádně podíval, všiml by sis, že mám klobouk, a nemusel by ses ptát."

Mág už otvíral pusu, aby kontroval nějakou peprnou narážkou, ale pak ji zase zavřel. Hádání mu tady nijak neprospěje. On se potřebuje dostat do dračí jeskyně a nasbírat dostatek Temných kamenů. „Poslyš," začal mág opatrně, „ty o tom drakovi něco víš?"

„Jistě, že vím. Vím to nejdůležitější, co se o něm dá vědět: Jeden by se měl od něj držet co nejdál!"

„No ne, vážně?" upřel na ni nevinný pohled Felix a při tom žoviálně pohupoval ohořelým ocasem na všechny strany.

„Ale my se potřebujeme dostat dovnitř a nasbírat nějaké Temné kameny," pokýval smutně hlavou mág.

„Pak bych tu možná měla něco, co by vám mohlo pomoct. Ale něco za něco."

Družinka následovala čarodějku až k jejímu domku uprostřed lesa. Když zastavili, ukázala čarodějka směrem na půdu: „Na půdě se mi přemnožili skřítci a já už nevím, co s nimi."

„To znám, jednou se mi to také přihodilo," přikývl mág. „Na to je nejlepší 'Piškorcův deratizátor'!"

„Žádná taková bylinka tady neroste. To bych musela vědět."

„Ale ne, to je zaklínadlo!" vysvětloval mág s převahou znalce. „Má ale jeden háček . . . "

„To mají zaklínadla vždycky," ušklíbla se čarodějka.

„Deratizovat se musí přesný počet skřítků. Když jich je víc, tak nějací přežijí a pak se dál množí. A když je jich míň, tak se musí přebytečná magenergie někam uvolnit a to bývá často . . . nepříjemné," dokončil neohrabaně mág.

„Myslíš tak nepříjemné jako tenkrát, kdy ti narostly oslí uši a celý týden jsi nemohl vyjít z ložnice?" pochechtával se Felix.

„Ne, myslím tak nepříjemné, že by přebytečná magenergie deratizovala jiné tvory v blízkém okolí. A začala by těmi menšími . . . “ zmrazil kocourovu zábavu mág.

20-5-2 Piškorcův deratizátor — 10 bodů

Je potřeba, aby při sesílání kouzla byl počet skřítků alespoň N. Na začátku je skřítků K, kde $K < N$. Každý den se přesně o půlnoci počet skřítků ztrojnásobí. Čarodějka umí každý den povolat nového skřítka nebo jednoho skřítka nechat zmizet. Samozřejmě nemusí dělat nic a nechat populaci takovou, jaká je.

Mág umí seslat (i několikrát za den) deratizační kouzlo, při kterém zmizí právě N skřítků. Aby ho mohl seslat, musí být skřítků alespoň N, jinak by se mohly stát ošklivé věci.

Navrhněte postup, jak skřítky každý den přidávat, odebírat a případně deratizovat, aby na konci nezbyl žádný. Musíte mága šetřit, aby se úplně nevyčerpal, neboť ho ještě čeká souboj s drakem, takže vámi navržený postup musí obsahovat co nejméně deratizací. Navíc by vytvoření vašeho plánu mělo trvat co nejkratší dobu, aby se jím stihli čaroděj s čarodějkou vůbec řídit.

Čarodějka i mág mohou kouzlit hned první den. Nemusí tedy čekat, až se jim K ještě ztrojnásobí.

Příklad: Na začátku mějme 4 skřítky a deratizační kouzlo jich zlikviduje 7. Sledujme populaci po jednotlivých dnech (v závorkách jsou počty skřítků):

1. (4) přidej skřítka (5)
2. (15) zmiz skřítka, deratizace, deratizace (0)

„Jo, všechna ta havěť je pryč,“ usmál se pod vousky Felix, když pečlivě prošmejdil celou půdu. „Ale mohli jste mi nechat jednoho na hraní . . . “

„To by tak ještě chybělo,“ odbyl ho mág. „Ty jsi neviděl, jak se ty potvory rychle množí?“

„Výborně, chlapci,“ pochválila je čarodějka a postavila před Felixe misku s mlékem.

„Krá, krá,“ ozval se Kiri a dožadoval se také nějaké odměny, ale pro havrana se v domě nenašel jediný pamlsek.

Čarodějka otevřela jednu ze svých truhel a podala mágovi zažloutlý svitek převázaný pečetí. Na pečeti byl symbol draka uzavřený do kruhu. Mág z části sfoukl a z části sklepal vrstvu prachu, která svitek pokrývala. „Co to je?“ zeptal se podezřívavě.

„To je svitek ochrany před draky. Po rozlomení pečeti se kolem svitku vytvoří neviditelná bariéra o poloměru 42 *metrů, do které není žádný drak schopen vstoupit,“ vysvětlovala trpělivě čarodějka.*

„A jak dlouho to vydrží?“

„Asi hodinu. Doufám. Alchymista, který mi ten svitek věnoval, byl celkem roztržitý . . . “

Mág poděkoval a celá družina se vydala zpět k Ohnivé hoře. Hora stála na svém místě a dýmala. Po drakovi nebylo ani vidu ani slechu. Vzduch byl nehybný a všude vládlo naprosté ticho. Klid před bouří, pomyslel si mág. Teď musíme najít vchod do jeskyně, kde bydlí ten drak. Konec konců, musel se přece nějak dostat ven.

Několik hodin chodili po úbočí hory, když si Vilda všiml velké díry vysoko ve skále. Tou by se určitě protáhl i drak. Družina se s funěním a hekáním vyškrábala až k výklenku.

„Ťuk, ťuk,“ řekl potichu Felix, sotva popadl dech. „Vidíte, nikdo není doma, tak můžeme zas jít, ne?“ Mág ho ale odstrčil z cesty a pevným krokem vykročil vpřed. Tunel se pozvolna rozšiřoval, až vyústil do obrovské sluje, do které by se vešlo . . . opravdu hodně draků. Naštěstí tam byl jen jeden. Ležel na hromadě horkých kamenů a spal.

Mág se rozhlédl po jeskyni. Na spoustě míst ležely Temné kameny, ale jeskyně byla příliš velká, než aby ji celou pokryl kouzlem ze svitku . . .

20-5-3 Ochrana před draky — 13 bodů

Pro naše potřeby si úlohu trochu zjednodušíme. Představíme si pouze dvourozměrný půdorys dračí jeskyně. Máme seznam souřadnic, kde se nachází Temné kameny. Svitek má dosah N metrů a mág se s ním snaží pokrýt co nejvíc kamenů. Kámen je považován za pokrytý, pokud je jeho vzdálenost od svitku menší nebo rovna N.

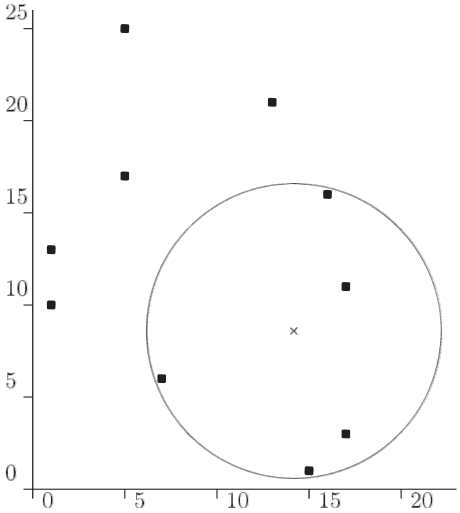
Napište algoritmus, který dostane seznam bodů v rovině a nalezne bod takový, že když v něm nakreslíme kruh o poloměru N, bude tento kruh pokrývat maximální možný počet bodů. Střed kruhu může být kdekoli, ne nutně v nějakém zadaném bodě.

Pokud existuje takových bodů víc, stačí vypsat jeden libovolný z nich. Souřadnice se uvádějí jako reálná čísla (a vejdou se do nějakého float typu v počítači).

Příklad: Řekněme, že poloměr svitku bude 8 metrů a v jeskyni je 10 kamenů na následujících souřadnicích:

(1.0, 13.0), (5.0, 17.0), (7.0, 6.0), (13.0, 21.0), (17.0, 3.0), (5.0, 25.0), (15.0, 1.0), (16.0, 16.0), (17.0, 11.0), (1.0, 10.0)

Pokud umístíme svitek na souřadnice $[14.178125, 8.58475]$, pokryjeme maximální počet – 5 kamenů.



Mág došel doprostřed jeskyně a rozhlédl se. Ano, tady je nejlepší místo. Vytáhl z vaku svitek a pečlivě si ho ještě jednou prohlédl. Drak otevřel oko.

„Á, návštěva!“ zaburácel drak a udělal krok k mágovi.

Jestli to teď nebude fungovat, tak je s námi konec, pomyslel si mág. Pozvedl svitek a zavolal: „Neprojdeš dál!“ Na ta slova rozlomil pečeť. Mágovi se naježily vousy i vlasy a vzduch se naplnil mazlavým zápachem použité magenergie.

„A kam bych jako neměl projít?“ zeptal se drak a udělal další dva kroky. Chtěl udělat ještě jeden, ale narazil na neviditelnou bariéru a rozplácl se na ní, jako moucha na předním skle závodního létajícího koštěte.

„Už chápu,“ brblal si pro sebe, když si masíroval naražený čumák. „To je zajímavá věcička . . . “

K mágovi zatím přiběhl Vilda a začal sbírat Temné kameny.

„Tak by mě zajímalo,“ nahodil drak konverzačním tónem, „jestli vás to kouzlo taky ochrání před předměty, které bych mohl třeba neopatrně upustit dovnitř . . . “ Mág ustaraně pozoroval, jak drak sebral středně velký kámen do pařátů a ledabyle ho prohodil magickou bariérou.

„Hups,“ usmál se drak. „A taky by mě zajímalo, jestli ten svitek bude fungovat i potom, co shoří,“ řekl a z nosu mu vyšel obláček kouře. Mág odhodil svitek na zem a o několik kroků ustoupil.

„Jen nevím, kam si vás vystavím,“ pokračoval drak.„Mám tu barbary drakobijce, trpaslíky drakobijce, dokonce i pár rytířů . . . ale kouzelník, zombie, kočka a pták . . . na to si budu muset založit samostatnou sbírku.“

Vilda už měl v torně několik Temných kamenů a tak začal ustupovat společně s mágem. Drak nasměroval svůj chřtán přímo na svitek a úzkým kontrolovaným

plamenem ho v mžiku přeměnil na uhel. Ozvalo se slabé zapraskání a magická bariéra povolila.

„Ehm," odkašlal si mág, „my jsme vám nepřišli nijak ... ublížit ... "

„Ale ovšem, že ne. Nechte mě hádat – vy jste přišli na koblihu a šálek čaje, že?" zašklebil se na něj drak.

„No, ve skutečnosti jsme přišli ... jen pro pár Temných kamenů." vypravil ze sebe mág a přitom horečnatě přemýšlel, jak z téhle nepříjemné situace ven.

„Ach tak. Takže vás zajímá pár obyčejných šutrů, zatímco drak a jeho poklad jsou vám úplně lhostejní, že?"

„Jaký poklad?" podíval se na něj mág.

„Hmm. Tady něco nesedí," zarazil se drak. „Jenom abych si to ujasnil: Vy jste sem přišli s nějakou magickou ochranou před draky, ale ve skutečnosti jste neměli v úmyslu mi nijak ublížit a několik bezcenných šutrů vás zajímá víc, než můj poklad," přemýšlel nahlas.

„Jo, přesně tak to bylo," přikyvoval horlivě Vilda.

„Krá," přispěchal mu na pomoc Kiri.

„Já jim hned říkal, že to neprojde," přidal se Felix. „Co na mě všichni tak zíráte? Říkal jsem vám to! Nebo snad ne?!"

„Takže, přišli, seslali a nechtějí poklad," mumlal si pro sebe drak, jako by s tou myšlenkou měl potíže. „Pořád tady jedné věci nerozumím – proč?"

Mág mu začal vysvětlovat, jaké to je, být pánem Temného hvozdu, a co všechno musí dělat, aby si udržel potřebný image. Pak tu byla ta patálie s temnou lucernou a Temné kameny se špatně shánějí. Vyprávěl mu, jak museli projít Temným hvozdem, zjistit, kde vlastně takové kameny hledat, a pak se dotrmácet až sem přes všechny ty močály a druidy ...

Drak se zaujetím poslouchal a občas vypustil proužek dýmu. A protože mág uměl každý příběh podat jako nikdo jiný, začal drak dojetím slzet.

„To je tak dojebdý příběh," řekl drak a popotáhl. „Debáte dáhodou khapesník?"

Mág s Vildou se na sebe podívali. „Máme jen tohle," řekl Vilda, vytáhl z vaku obrovskou deku a podal ji drakovi.

„Ďhekuju," odvětil drak a hlasitě se vysmrkal, až to zatřáslo jeskyní. Deka mu shořela v pařátcích na troud a rozsypala se.

„Víte, já jsem se sem přestěhoval z daleka," navázal drak, když se trochu sebral. „Žil jsem s rodiči ve Velkém pohoří, ale znáte to. Přijde čas, kdy se potřebujete trochu osamostatnit. Vzlétnout na vlastní křídla, jak se říká. Myslel jsem, že tady to bude lepší. Nová země, čerstvá síra ... ale ve skutečnosti je to tady hrozné. V horách jsem měl klid. Jenže sem mi pořád někdo leze. Většinou je to pořád dokola to samé – hrdina sem přijde, vyzve mě na souboj a já pak jen po něm uklidím ohořelé boty a meč s nápisem 'Drakobijec', nebo tak nějak. Problém je, že hora je přímo protkaná velkým množstvím nejrůznějších tunelů

a jeskyní, které tu vytvořila láva. Některé tunely vedou i na povrch a pak mi sem lezou lidé. Chtěl jsem některé tunely zasypat, ale nevím které. A taky si nechci zasypat svůj poklad ...“ dokončil smutně drak.

„S tím bych možná mohl pomoct,“ usmál se mág.

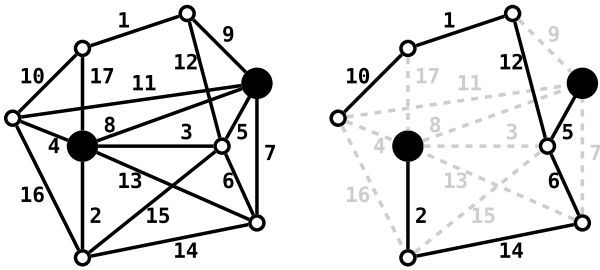
20-5-4 Dračí chodbičky — 11 bodů

Spleť dračích chodeb a jeskyní si představíme jako graf, kde vrcholy jsou jeskyně nebo křižovatky a hrany jsou tunely. Graf nemusí být nutně rovinný, protože hora je velká a některé tunely se mohou křížit mimoúrovňově.

Drak by rád co nejvíc chodeb zasypal, ale zároveň chce, aby se dostal do všech jeskyní (vrcholů). Také vám dává seznam míst, ve kterých má část pokladu. K takovým místům by chtěl nechat pouze jednu přístupovou chodbu (tj. z těchto vrcholů mají být listy). Navrhněte, které chodby by měl drak zachovat, aby součet délek zasypaných chodeb byl největší možný.

Můžete předpokládat, že zadaný problém má řešení (tzn. z vrcholů s pokladem lze udělat listy, aniž by se graf rozpadl na více komponent).

Příklad: Vlevo je obrázek současného stavu tunelů v Ohnivé hoře (místa s pokladem jsou vyznačena černě). Vpravo pak vidíte výsledek (zasypané chodby jsou čárkované).



Celá hora se třásla a všude se ozýval ohlušující zvuk padajícího kamení.

„To byla poslední,“ pohladil si mág spokojeně plnovous, když hluk ustal.

„To je úžasné,“ rozplýval se drak nad provedenými stavebními úpravami. „A tady bych si mohl zřídit konferenční salónek. Až přiletí naši na návštěvu, ti budou koukat ...“

Družinka se rozloučila s drakem a vydala se na dlouhou cestu zpět do Temného hvozdu. Putování jim zabralo hezkých pár týdnů, ale nakonec dorazili všichni ve zdraví domů.

Před Temnou věží se už tísnilo několik rádobydobrodruhů, kteří se zbraní v ruce čekali na mága. Mág jim pokynul na pozdrav a pozval je, jako obvykle, na šálek čaje.

A všechno bylo zase jako dřív. Mág měl svou temnotu, hvozd měl svého pána a dobrodruhové celé Škytánie měli opět kam chodit za hrdinstvím ... a na čaj.

20-5-5 Roztržitý matematik — 15 bodů

Všechno, co má začátek, má i svůj konec. A tak se i nám pomalu blíží konec jubilejního 20. ročníku KSP. Ale ještě než se stane nevyhnutelné, můžete si vyřešit poslední praktickou úložku. Je taková ... ze života.

Způsob odevzdávání a všechny ostatní detaily zůstávají stejné jako v minulých sériích. Takže pokud jste zapomněli, jak to v praktické úložce chodí, nebo jste se do řešení KSP zapojili teprve teď, podívejte se na úložku 20-1-5 z první série, kde naleznete potřebné informace.

Zadání:

Roztržitý matematik tráví většinu svého času ve své malé pracovně na Karlíně. Po stole, po zemi, po stěnách a občas i po stropě se povalují nejrůznější papíry s nedokončenými výpočty, rozečtenými články a sem tam se objeví i seznam s nákupem nebo lísteček z čistírny. Není divu, že se matematikovi těžce pracuje, když neustále něco hledá ...

Všechny matematikovy papíry (včetně obalů od svačiny) jsou očíslované. Matematik má také svůj odkládací systém, ve kterém se sice nevyznáme, ale pro zjednodušení budeme předpokládat, že všechny papíry leží v řadě za sebou. Když matematik nějaký papír použije, vyndá jej z řady, chvíli do něho údivně zírá mumlaje si pod vousy nesrozumitelné věci, načež tento papír položí na začátek řady (ostatní papíry se posunou). Na začátku matematikovy práce to šlo pěkně, neboť všechny papíry byly seřazeny podle čísel $(1, 2, \ldots N)$. Teď už jsou ale hodně přeházené a matematik nemůže najít ani svoji tramvajenku. Naštěstí si ještě pamatuje, kolikátý od začátku řady byl každý papír, se kterým pracoval. A v tomto okamžiku nastupujete do vzniklého chaosu vy, abyste matematika zachránili před jistou smrtí vyčerpáním.

Ve vstupním souboru `papiry.in` jsou na prvním řádku dvě čísla N a K, kde N $(1 \leq N \leq 500000)$ představuje počet papírů a K $(1 \leq K \leq 500000)$ počet operací, které matematik udělal. Na druhém řádku je posloupnost K čísel, kde každé číslo x_i představuje i-tou operaci, při které matematik vzal x_i-tý papír od začátku řady a posunul ho na první místo. Před započetím všech operací byly papíry seřazeny vzestupně od 1 do N.

Výstup uložte do souboru `papiry.out` tak, že na prvním řádku bude N čísel představujících permutaci dokumentů po provedení všech K operací.

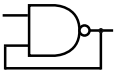
Příklad:

`papiry.in`	8 3 5 1 4
`papiry.out`	3 5 1 2 4 6 7 8

20-5-6 Hrady, hrádky, hradla — 12 bodů

Dnes bude naše povídání tak trošku z jiného soudku. V celém seriálu se naše obvody chovaly kombinačně, což znamená, že nebyly závislé na předchozím stavu a na jeden vstup, který si třeba můžeme představit jako celé číslo zapsané binárně, jsme s jistotou dostali vždy stejný výstup (jiné celé číslo zapsané binárně). Nyní se nám věci začnou komplikovat, neboť naučíme obvody „pamatovat si“ a díky tomu výstup obvodu nebude nutně záležet pouze na vstupu, ale výsledek může ovlivnit i vnitřní stav obvodu. Vnitřní stav obvodu je to, co si obvod pamatuje z minulých vstupů, tedy obvod může vydat jiný výstup na stejný vstup, pokud jsme posloupnost zkoušených vstupů přeházeli. U skutečných obvodů je počáteční stav, tedy stav po zapnutí přístroje nedefinovaný (převážně díky fyzikálním efektům). My si pro jednoduchost počáteční stav, tedy stav na začátku výpočtu, sami nadefinujeme.

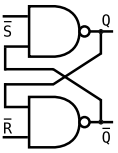
Například by takový obvod mohl počítat průběžnou paritu, na vstupu by byla buď jednička nebo nula, a na výstupu parita aktuální části binárního čísla reprezentovaného posloupností bitů na vstupu. Než se ale do takového obvodu pustíme, musíme vyřešit jeden drobný problém a to, jak má takový obvod poznat dva po sobě jdoucí jedničkové bity (rozmyslete si, že po sobě jdoucí nulové bity v tomto případě nemění výsledek a proto je nemusíme umět rozlišit.) Problém se řeší jednoduše, přidáme si do vstupu další bit, který se bude měnit při každém novém vstupu, tedy tři po sobě jdoucí jedničky budou vypadat třeba jako $10, 11, 10$. V elektronice se podobný signál obvykle označuje jako hodinový (Clock), s tím rozdílem, že reálně se za změnu vstupu považuje chvíle, kdy se hodinový signál přehoupne z nuly na jedničku (náběžná hrana). V následujícím textu budeme hodinový signál považovat za aktivní na náběžné hraně. Problém jsme vyřešili, tedy nechť si obvod na začátku pamatuje, že průběžná parita, tj. parita již načtené části čísla je nula. Pak obvod pro každou nulu na vstupu pošle zapamatovanou paritu na výstup, a pro každou jedničku provede negaci zapamatované dosavadní parity a tuto hodnotu pošle na výstup. Je jistě vidět, že se obvod pro jedničku na vstupu chová různě a rozhoduje se podle vnitřního stavu.



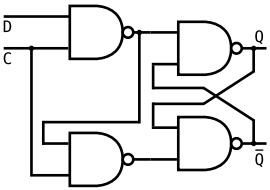
Nebudeme už dále chodit kolem horké kaše a ukážeme, jak se taková paměť vytvoří. Musíme vymyslet, jak uchovat nějakou hodnotu. Když vezmeme hradlo NAND a zapojíme jeho výstup na jeden ze dvou vstupů, dostaneme obvod, který si umí zapamatovat, že na vstupu byla jednička. Nechť je výstup nastaven na jedničku, pak jeden ze vstupů je nastaven také na jedničku a obvod v tomto stavu vydrží, dokud nenastavíme druhý vstup hradla na jedničku. Tím se výstup hradla přepne a bude již mít trvale na výstupu nulu.

Takové hradlo je sice velice zajímavé, i když pramálo užitečné, nám by se hodilo umět jednak výstup nastavit, ale i vynulovat. Vezmeme tedy hradla

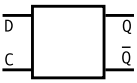
NAND dvě a zapojíme výstup jednoho na vstup druhého a naopak. Takové zapojení se jmenuje klopný obvod RS. Funguje jednoduše, máme dva vstupy. Vstup Set, který nastavuje výstup na jedničku a vstup Reset, který nastavuje Výstup na nulu (odtud se také vzalo RS v pojmenování). Vstupy jsou negované, tedy považujme nezapojený vstup, za vstup na kterém je jednička. Připojením právě jednoho vstupu na nulu se buď obvod přepne, nebo neudělá nic (to záleží na vnitřním stavu). Výstup je na výstupu hradla označen písmenem Q, zatímco Q s pruhem je jeho negace.



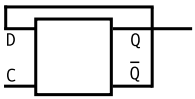
Odtud není daleko ke klopnému obvodu D, který je základem všech paměťových obvodů. Narozdíl od klopného obvodu RS je řízen hodinovým signálem. Funguje tak, že se vstup D „zkopíruje“ na výstup Q, v okamžiku, kdy se na hodinovém vstupu nastaví jednička. V řeči elektroniky bychom řekli, že se vstup zapíše na výstup na náběžné hraně hodinového signálu. Na následujícím obrázku je klopný obvod typu D, který ovšem nereaguje na náběžnou hranu, ale kopíruje vstup D, na výstup Q, když je vstup C nastavený na jedničku.



Obvod který reaguje na náběžnou hranu je o něco složitější a proto ho budeme kreslit následující schématickou značkou:



V sekvenčních obvodech se často využívá zpoždění na hradle, které nás v předchozích úlohách trápilo. Pro nás je teď důležité, že signál projde hradlem pomaleji než drátem (rozumně krátkém, kdybychom natáhli drát kolem země, bude samozřejmě signál hradlem procházet rychleji, nehledě na to, že se v tak dlouhém drátu po cestě ztratí). To znamená, že když za sebe zapojíme dvě hradla NOT, čímž dostaneme původní signál, a vedle natáhneme drát zapojený do téhož vstupu, bude na výstupu těchto hradel jednička ještě chvíli poté, co na vedlejším drátu bude už nula a naopak. S tímto efektem a klopným obvodem D lze vyrobit takzvanou děličku. To je obvod, jenž pro vstupní signál, kde se pravidelně střídají jedničky a nuly (hodinový signál) vyrobí signál, kde se pravidelně střídají dvě jedničky a dvě nuly (dělí frekvenci v původním signálu dvěma).



Vaším úkolem bude vymyslet:

1. zmíněný obvod na průběžnou paritu [5 bodů]
2. čítač, to jest obvod, který má na vstupu hodinový signál a postupně s každou jedničkou na vstupu zvýší hodnotu na výstupu (reprezentovanou binárním N-bitovým číslem) o jedna. [7 bodů]

Programátorské kuchařky

20-2-K Kuchařka druhé série – třídění

I letos Vám kromě úloh budeme servírovat také recepty z programátorské kuchařky. Některé si vypůjčíme z dřívějších ročníků KSP, ale i k těm se budeme snažit připsat něco nového, aby si i zkušenější řešitelé přišli na své. V letošní první kuchařce si povíme o třídících algoritmech. Co to znamená? Pojem *třídění* je možná maličko nepřesný, nehodláme data (čísla, záznamy, řetězce a jiné) rozdělovat do nějakých tříd, ale přerovnat je do správného pořadí, protože se seřazenými údaji se mnohem lépe pracuje, například pokud v nich pak potřebujeme vyhledávat. Takové uspořádávání dat je denním chlebem každého programátora, a tak není divu, že třídící algoritmy jsou jedny z nejstudovanějších. My však nebudeme do nějakých velkých detailů a specialit příliš zabíhat. Zkrátka a dobře – budeme chtít třídit údaje rychle, úsporně a radostně.

Obvykle třídíme exempláře datové struktury typu pascalského záznamu. V takové datové struktuře bývá obsažena jedna význačná položka, *klíč*, podle které se záznamy řadí. Malinko si náš život zjednodušíme a budeme předpokládat, že třídíme záznamy obsahující pouze klíč, který je navíc celočíselný – budeme tedy třídit pole celých čísel. Vzhledem k počtu tříděných čísel N pak budeme vyjadřovat časovou (a paměťovou) složitost jednotlivých algoritmů, které si předvedeme.

Metody třídění můžeme rozdělit do dvou hlavních skupin, a to na *vnitřní třídění,* kdy si můžeme dovolit všechna data načíst do (rychlé) paměti počítače, a na *vnější třídění,* kdy již třídění musíme realizovat opakovaným čtením a vytvářením diskových souborů. V tomto dílu se omezíme pouze na algoritmy vnitřního třídění a tříděné pole si nadeklarujeme takto:

```
const N = 100;
type Pole = array[1..N] of integer;
```

Nejjednodušší třídící algoritmy patří do skupiny *přímých metod.* Všechny mají několik společných rysů: Jsou krátké, jednoduché a třídí přímo v poli (nepotřebujeme pomocné pole). Tyto algoritmy mají většinou časovou složitost $O(N^2)$. Z toho vyplývá, že jsou použitelné tehdy, když tříděných dat není příliš mnoho. Na druhou stranu pokud je dat opravdu málo, je zbytečně složité používat některý z komplikovanějších algoritmů, které si předvedeme později.

Stručně si přiblížíme tři nejznámější přímé algoritmy. **Třídění přímým výběrem (SelectSort)** je založeno na opakovaném vybírání nejmenšího čísla z dosud nesetříděných čísel. Nalezené číslo prohodíme s prvkem na začátku pole a postup opakujeme, tentokrát s nejmenším číslem na indexech $2, \ldots, N$, které prohodíme s druhým prvkem v poli. Poté postup opakujeme s prvky s indexy $3, \ldots, N$, atd. Je snadné si uvědomit, že když takto postupně vybíráme minimum z menších a menších intervalů, setřídíme celé pole (v i-tém kroku nalezneme i-tý nejmenší prvek a zařadíme ho v poli na pozici s indexem i).

```
procedure SelectSort(var A: Pole);
var i,j,k,x: integer;
begin
  for i:=1 to N-1 do
    begin
      k:=i;
      for j:=i+1 to N do
        if A[j]<A[k] then k:=j;
      x:=A[k]; A[k]:=A[i]; A[i]:=x;
    end;
end;
```

Pro úplnost si ještě řekněme pár slov o časové složitosti právě popsaného algoritmu. V i-tém kroku musíme nalézt minimum z $N - i + 1$ čísel, na což spotřebujeme čas $O(N - i + 1)$. Ve všech krocích dohromady tedy spotřebujeme čas $O(N + (N-1) + \ldots + 3 + 2 + 1) = O(N^2)$.

Třídění přímým vkládáním (InsertSort) funguje na podobném principu jako třídění přímým výběrem. Na začátku pole vytváříme správně utříděnou posloupnost, kterou postupně rozšiřujeme. Na začátku i-tého kroku má tato utříděná posloupnost délku $i - 1$. V i-tém kroku určíme pozici i-tého čísla v dosud utříděné posloupnosti a zařadíme ho do utříděné posloupnosti (zbytek utříděné posloupnosti se posune o jednu pozici doprava). Není těžké si rozmyslet, že každý krok lze provést v čase $O(N)$. Protože počet kroků algoritmu je N, celková časová složitost právě popsaného algoritmu je opět $O(N^2)$.

```
procedure InsertSort(var A: Pole);
var i,j,x: integer;
begin
  for i:=2 to N do
    begin
      x:=A[i];
      j:=i-1;
      while (j>0) and (x<A[j]) do
        begin
          A[j+1]:=A[j];
          j:=j-1;
        end;
      A[j+1]:=x;
    end;
end;
```

(Upozornění: v našich příkladech předpokládáme, že máme v překladači zapnuto tzv. zkrácené vyhodnocování logických výrazu, třeba v předchozím while-cyklu se při `j=0` hodnoty `x` a `A[0]` již neporovnávají.)

Bublinkové třídění (BubbleSort) pracuje jinak než dva dříve popsané algoritmy. Algoritmu se říká „bublinkový“, protože podobně jako bublinky v limonádě „stoupají“ vysoká čísla v poli vzhůru. Postupně se porovnávají dvojice sousedních prvků, řekněme zleva doprava, a pokud v porovnávané dvojici následuje menší číslo po větším, tak se tato dvě čísla prohodí. Celý postup opakujeme, dokud probíhají nějaké výměny. Protože algoritmus skončí, když nedojde k žádné výměně, je pole na konci algoritmu setříděné.

```
procedure BubbleSort(var A: Pole);
var i,x: integer;
    zmena: boolean;
begin
  repeat
    zmena:=false;
    for i:=1 to N-1 do
      if A[i] > A[i+1] then
        begin
          x:=A[i]; A[i]:=A[i+1]; A[i+1]:=x;
          zmena:=true;
        end;
  until not zmena;
end;
```

Správnost algoritmu nahlédneme tak, že si uvědomíme, že po i průchodech while-cyklem bude posledních i prvků obsahovat největších i prvků setříděných od nejmenšího po největší (rozmyslete si, proč tomu tak je). Popsaný algoritmus se tedy zastaví po nejvýše N průchodech a jeho celková časová složitost v nejhorším případě je $O(N^2)$, neboť na každý průchod spotřebuje čas $O(N)$. Výhodou tohoto algoritmu oproti předchozím dvěma je, že je tím rychlejší, čím blíže bylo zadané pole k setříděnému stavu – pokud bylo úplně setříděné, tehdy algoritmus spotřebuje jen lineární čas, $O(N)$.

Sofistikovanější třídící algoritmy pracují v čase $O(N \log N)$. Jedním z nich je **Třídění sléváním (MergeSort),** založené na principu slévání (spojování) již setříděných posloupností dohromady. Představme si, že již máme dvě setříděné posloupnosti a chceme je spojit dohromady. Jednoduše stačí porovnávat nejmenší prvky obou posloupností a menší z těchto prvků vždy odstranit a přesunout do nové posloupnosti. Je zřejmé, že ke slití dvou posloupností potřebujeme čas úměrný součtu jejich délek.

My si zde popíšeme a předvedeme modifikaci algoritmu MergeSort, která používá pomocné pole. Algoritmus lze implementovat při zachování časové složitosti i bez pomocného pole, ale je to o dost pracnější. Existuje též modifikace

algoritmu, která má počet fází (viz dále) v nejhorším případě $O(\log N)$, ale pokud je již pole na začátku setříděné, proběhne pouze jediná a v takovém případě má algoritmus časovou složitost $O(N)$. My si však zatajíme i tuto variantu.

Algoritmus pracuje v několika *fázích.* Na začátku první fáze tvoří každý prvek jednoprvkovou setříděnou posloupnost a obecně na začátku i-té fázi budou mít setříděné posloupnosti délky 2^{i-1}. V i-té fázi tedy vždy ze dvou sousedních 2^{i-1}-prvkových posloupností vytvoříme jedinou délky 2^i. Pokud N není násobkem 2^i, bude délka poslední posloupnosti zbytek po dělení N číslem 2^i. Zastavíme se, pokud $2^i \geq N$, tj. po $\lceil \log_2 N \rceil$ fázích. Protože v i-té fázi slijeme $\lceil N/2^i \rceil$ dvojic nejvýše 2^{i-1}-prvkových posloupností, je časová složitost jedné fáze $O(N)$. Celková časová složitost popsaného algoritmu je pak $O(N \log N)$.

```
procedure MergeSort(var A: Pole);
var P: Pole;        { pomocné pole }
    delka:integer;  { délka setříděných posloupností }
    i: integer;     { index do vytvářené posloupnosti }
    i1,i2: integer; { index do slévaných posloupností }
    k1,k2: integer; { konce slévaných posloupností }
begin
  delka:=1;
  while delka<N do
    begin
      i1:=1; i2:=delka+1; i:=1;
      k1:=delka; k2:=2*delka;
      while i<=N do
        begin
          { sléváme A[i1..k1] s A[i2..k2] }
          if k2>N then k2:=N;
          while (i1<=k1) or (i2<=k2) do
            if (i2>k2) or
               ((i1<=k1) and (A[i1]<=A[i2]))
            then
              begin
                P[i]:=A[i1]; i:=i+1; i1:=i1+1;
              end
            else
              begin
                P[i]:=A[i2]; i:=i+1; i2:=i2+1;
              end;
          i1:=k2+1; i2:=i1+delka;
          k1:=k2+delka;
          k2:=k2+2*delka;
        end;
      A:=P;
      delka:=2*delka;
    end;
end;
```

V čase $O(N \log N)$ pracuje také algoritmus jménem **QuickSort.** Tento algoritmus je založen na metodě Rozděl a panuj. Nejprve si zvolíme nějaké číslo,

kterému budeme říkat *pivot.* Více si o jeho volbě povíme později. Poté pole přeuspořádáme a rozdělíme je na dvě části tak, že žádný prvek v první části nebude větší než pivot a žádný prvek v druhé části naopak menší. Prvky v obou částech pak setřídíme rekurzivním zavoláním téhož algoritmu. Musíme ale dát pozor, aby byly v každém kroku obě části neprázdné (a rekurze tedy byla konečná). Je zřejmé, že po skončení algoritmu bude pole setříděné.

Malá zrada spočívá ve volbě pivota. Pro naše účely by se hodilo, aby po přeházení prvků levá i pravá část pole byly přibližně stejně velké. Nejlepší volbou pivota by tedy byl *medián* tříděného úseku, tj. prvek takový, jenž by byl v setříděném poli přesně uprostřed. Přeuspořádání jistě zvládneme v lineárním čase a pokud by pivoty na všech úrovních byly mediány, pak by počet úrovní rekurze byl $O(\log N)$ a celková časová složitost $O(N \log N)$ (na každé úrovni rekurze je součet délek tříděných posloupnosti nejvýše N). Ačkoli existuje algoritmus, který medián pole nalezne v čase $O(N)$, v QuickSortu se obvykle nepoužívá, jelikož konstanta u členu N je příliš velká v porovnání s pravděpodobností, že náhodná volba pivota algoritmus příliš zpomalí. Většinou se pivot volí náhodně z dosud nesetříděného úseku – zkrátka se sáhne někam do pole a nalezený prvek se prohlásí za pivot. Dá se ukázat, že takovýto algoritmus s velmi vysokou pravděpodobností poběží v čase $O(N \log N)$. Důkaz tohoto tvrzení je trošičku trikový a lze jej nalézt např. v knize Kapitoly z diskrétní matematiky od pánů Matouška a Nešetřila. Je však třeba si pamatovat, že pokud se pivot volí náhodně, může rekurze dosáhnout hloubky N a časová složitost algoritmu až $O(N^2)$ – představme si, že se pivot v každém rekurzivním volání nešťastně zvolí jako největší prvek z tříděného úseku. V naší implementaci QuickSortu pro názornost nebudeme pivot volit náhodně, ale vždy jako pivot vybereme prostřední prvek tříděného úseku.

```
procedure QuickSort(var A:Pole; l,r:integer);
var i,j,k,x: integer;
begin
  i:=l; j:=r;
  k:=A[(i+j) div 2];
  repeat
    while A[i]<k do i:=i+1;
    while A[j]>k do j:=j-1;
    if i<=j then
      begin
        x:=A[i]; A[i]:=A[j]; A[j]:=x;
        i:=i+1;
        j:=j-1;
      end;
  until i >= j;
  if j>l then QuickSort(A, l, j);
  if i<r then QuickSort(A, i, r);
end;
```

Ještě si předvedeme dva třídící algoritmy, které jsou vhodné, pokud tříděné objekty mají některé další speciální vlastnosti. Prvním z nich je **třídění počítáním (CountSort).** To lze použít, pokud tříděné objekty obsahují pouze klíče a možných hodnot klíčů je málo. Tehdy si stačí spočítat, kolikrát se který klíč vyskytuje, a místo třídění vytvořit celé pole znovu na základě toho, kolik jednotlivých objektů obsahovalo pole původní. My si tento algoritmus předvedeme na příkladu třídění pole celých čísel z intervalu $\langle D, H \rangle$:

```
const D = 1;
      H = 10;
procedure CountSort(var A: Pole);
var C: array[D..H] of integer;
    i,j,k: integer;
begin
  for i:=D to H do C[i]:=0;
  for i:=1 to N do C[A[i]]:=C[A[i]] + 1;
  k:=1;
  for i:=D to H do
    for j:=1 to C[i] do
      begin
        A[k]:=i;
        k:=k+1;
      end;
end;
```

Časová složitost takovéhoto algoritmu je lineární v N, ale nesmíme zapomenout přičíst ještě velikost intervalu, ve kterém se prvky nacházejí ($K = H - D + 1$), protože nějaký čas spotřebujeme i na inicializaci pole počítadel. Celkem tedy $O(N + K)$.

Pokud by tříděné objekty obsahovaly vedle klíčů i nějaká data, můžeme je místo pouhého počítání rozdělovat do přihrádek podle hodnoty klíče a pak je z přihrádek vysbírat v rostoucím pořadí klíčů. Tomuto algoritmu se říká **přihrádkové třídění (BucketSort)** a my si popíšeme jeho víceprůchodovou variantu **(RadixSort),** která je vhodnější pro větší hodnoty K. V první fázi si čísla rozdělíme do přihrádek (skupin) podle nejméně významné cifry a spojíme do jedné posloupnosti, v druhé fázi čísla roztřídíme podle druhé nejméně významné cifry a opět spojíme do jedné posloupnosti, atd. Je důležité, aby se uvnitř každé přihrádky zachovalo pořadí čísel v posloupnosti na začátku fáze, tj. posloupnost uložená v každé přihrádce je vybranou podposloupností posloupnosti ze začátku fáze. Tvrdíme, že na konci i-té fáze obsahuje výsledná posloupnost čísla utříděná podle i nejméně významných cifer. Zřejmě i-té nejméně významné cifry tvoří neklesající posloupnost, neboť podle nich jsme právě v této fázi rozdělovali čísla do přihrádek, a pokud dvě čísla mají tuto cifru stejnou, jsou uložena v pořadí dle jejich $i - 1$ nejméně významných cifer, neboť v každé přihrádce jsme zachovali pořadí čísel z konce minulé fáze. Na

závěr poznamenejme, že místo čísel podle cifer lze do přihrádek rozdělovat též textové řetězce podle jejich znaků, atp.

Jak je to s časovou složitostí této varianty RadixSortu? Pokud třídíme celá čísla od 1 do K a v každém kroku je rozdělujeme do ℓ přihrádek, potřebujeme $\log_\ell K$ průchodů (tolik je cifer v zápisu čísla K v ℓ-kové soustavě). Každý průchod spotřebuje čas $O(N + \ell)$, takže celý algoritmus běží v čase $O((N + \ell) \log_\ell K)$. To je $O(N)$, pokud K a ℓ jsou konstanty. My si předvedeme implementaci algoritmu pro $K = 255$ a $\ell = 2$ (čísla budeme rozhazovat do přihrádek podle bitů v jejich binárním zápisu).

```
const K=255;
procedure RadixSort(var A: Pole);
var P0,P1: Pole;
    k1,k2: integer;
    i: integer;
    bit: integer;
begin
  bit:=1;
  while bit<=K do
    begin
      k1:=0; k2:=0;
      for i:=1 to N do
        if (A[i] and bit)=0 then
          begin
            k1:=k1+1; P0[k1]:=A[i];
          end
        else
          begin
            k2:=k2+1; P1[k2]:=A[i];
          end;
      for i:=1 to k1 do A[i]:=P0[i];
      for i:=1 to k2 do A[k1+i]:=P1[i];
      bit:=bit shl 1;
    end;
end;
```

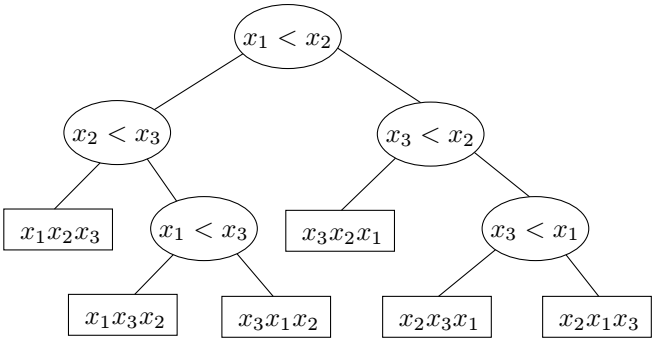
Na závěr našeho povídání o třídících algoritmech si ukážeme, že třídit obecné údaje, se kterými neumíme provádět nic jiného, než je navzájem porovnávat, rychleji než $O(N \log N)$ nejen nikdo neumí, ale také ani umět nemůže. Libovolný třídící algoritmus založený na porovnávání a prohazování prvků totiž musí na některé vstupy vynaložit řádově alespoň $N \log N$ kroků. (RadixSort na první pohled tento výsledek porušuje, na druhý však už ne, když si uvědomíme, o jak speciální druh tříděných dat se jedná.)

Třídící algoritmus v průběhu své činnosti nějak porovnává prvky a nějak je přehazuje. Provedeme myšlenkový experiment. Pozměníme algoritmus tak, že nejdříve bude pouze porovnávat, podle toho zjistí, jak jsou prvky v poli uspořádány, a když už si je jistý správným pořadím, prvky najednou popřehází. Tím se algoritmus zpomalí nejvýše konstanta-krát. Také pro jednoduchost

předpokládejme, že všechny tříděné údaje jsou navzájem různé. Porovnávací činnost algoritmu si pak můžeme popsat *rozhodovacím stromem.* Na obrázku je příklad rozhodovacího stromu pro tříprvkové pole.

Každý vrchol obsahuje porovnání dvou prvků x a y, v levém podstromu daného vrcholu je činnost algoritmu pokud $x < y$, v pravém podstromu činnost při $x \geq y$. V listech je už jisté správné pořadí prvků.

Každému algoritmu odpovídá nějaký rozhodovací strom a každý průběh činnosti algoritmu odpovídá průchodu rozhodovacím stromem od kořene do nějakého listu. Naším cílem bude ukázat, že v libovolném rozhodovacím stromu (a tedy i libovolném odpovídajícím algoritmu) bude existovat cesta z kořene do nějakého listu (neboli výpočet algoritmu) délky $N \log N$.



Kolik maximálně hladin h, a tedy i jaká nejdelší cesta se v takovém stromu může vyskytnout? Náš strom má tolik listů, kolik je možných pořadí tříděných prvků, tedy právě $N!$. Různým pořadím totiž musí odpovídat různé listy, jinak by algoritmus netřídil (předpokládáme přeci, že to, jak má prvky prohazovat, může zjistit jenom jejich porovnáváním), a naopak každé pořadí prvků jednoznačně určuje cestu do příslušného listu. Na nulté hladině je jediný vrchol, na každé další hladině se oproti předchozí počet vrcholů nejvýše zdvojnásobí, takže na i-té hladině se nachází nejvýše 2^i vrcholů. Proto je listů stromu nejvýše 2^h (některé listy mohou být i výše, ale za každý takový určitě chybí jeden vrchol na h-té hladině). Z toho víme, že platí:

$$2^h \geq \textit{počet listů} \geq N!, \quad \text{a proto} \quad h \geq \log_2(N!).$$

Logaritmus faktoriálu se těžko počítá přesně, ale můžeme si ho zdola odhadnout pomocí následujícího pozorování:

$$n! = \underbrace{n \cdot (n-1) \cdot \ldots \cdot (n/2)}_{n/2 \text{ členů, každý} \geq n/2} \cdot \ldots \geq$$

$$\geq (n/2)^{(n/2)}.$$

Dosazením získáme:

$$h \geq \log_2(N!) \geq \log_2((N/2)^{N/2}) =$$

$$= \frac{N}{2} \log_2(N/2) = \frac{1}{2} \cdot N(\log_2 N - 1) \geq \frac{1}{4} \cdot N \log_2 N.$$

Vidíme tedy, že pro každý třídící algoritmus existuje vstup, na kterém se bude muset provést alespoň $c \cdot N \log N$ kroků, kde $c > 0$ je nějaká konstanta.

Poznámky na okraj:

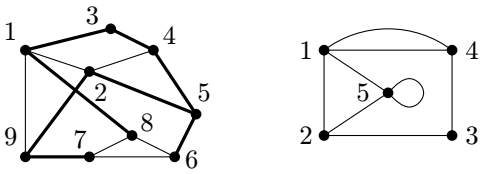
- Zkuste si též rozmyslet (drobnou modifikací předchozího důkazu), že ani *průměrná* časová složitost třídění nemůže být lepší než $N \log N$.
- Odvodit průměrnou složitost QuickSortu vlastně není zase tak  těžké. Zkusme následující úvahu: Pokud by pivot nebyl přesně medián, ale alespoň se nacházel v prostřední třetině setříděného úseku, byla by složitost stále $O(N \log N)$, jen by se zvýšila konstanta v O-čku. Kdybychom pivot volili náhodně, ale po rozdělení prvků si zkontrolovali, jestli pivot padl do prostřední třetiny, a pokud ne (jeden z úseků by byl moc velký a druhý moc malý), volbu bychom opakovali, v průměru by nás to stálo konstantní počet pokusů (pozorování z řešení úlohy 16-1-5: pokud čekáme na událost, která nastává náhodně s pravděpodobností p, stojí nás to v průměru $1/p$ pokusů; zde je $p = 1/3$), takže celková složitost by v průměru vzrostla jen konstantně. Původní QuickSort sice žádné takové opakování volby neprovádí a rovnou se zavolá rekurzivně na velký i malý úsek, ale opět se po v průměru konstantně mnoha iteracích velký úsek zredukuje na nejvýše 2/3 původní velikosti a třídění malých úseků jednotlivě nezabere víc času, než kdyby se třídily dohromady.
- Kdybychom u QuickSortu použili rekurzivní volání jen na menší interval, zatímco ten větší bychom obsloužili přenastavením proměnných a skokem na začátek právě prováděné procedury, zredukovali bychom paměťovou složitost na $O(\log N)$, jelikož každé další rekurzivní volání zpracovává alespoň dvakrát menší úsek než to předchozí. Časové složitosti tím však nepomůžeme.
- Počet přihrádek u RadixSortu vůbec nemusí být konstanta – pokud např. chcete třídit N čísel v rozsahu $1 \ldots N^k$, stačí si zvolit $\ell = N$ a fází bude jenom k. Pro pevné k tak dosáhneme lineární časové složitosti.

20-3-K Kuchařka třetí série – grafy

V dnešním vydání známého bestselleru budeme péci grafy souvislé i nesouvislé, orientované i neorientované. Řekneme si o základním procházení grafem, komponentách souvislosti, topologickém uspořádání a dalších grafových algoritmech. Abychom ale mohli začít, musíme si nejprve říci, s čím budeme pracovat.

Ingredience

Neorientovaný graf je určen množinou vrcholů V a množinou hran E, což jsou neuspořádané dvojice vrcholů. Hrana $e = \{x, y\}$ spojuje vrcholy x a y. Většinou požadujeme, aby hrany nespojovaly vrchol se sebou samým (takovým hranám říkáme *smyčky*) a aby mezi dvěma vrcholy nevedla více než jedna hrana (pokud toto neplatí, mluvíme o *multigrafech*). Obvykle také předpokládáme, že vrcholů je konečně mnoho. Neorientovaný graf většinou zobrazujeme jako body pospojované čarami.



Neorientovaný graf a multigraf

Podgrafem grafu G rozumíme graf G', který vznikl z grafu G vynecháním některých (a nebo žádných) hran a vrcholů.

Často nás zajímá, zda se dá z vrcholu x dojít po hranách do vrcholu y. Ovšem slovo „dojít“ by mohlo být trochu zavádějící, proto si zavedeme pár pojmů:

- *sled* budeme říkat takové posloupnosti vrcholů a hran tvaru $v_1, e_1, v_2, e_2, \ldots, e_{n-1}, v_n$, že $e_i = \{v_i, v_{i+1}\}$ pro každé i. Sled je tedy nějaká procházka po grafu. Délku sledu měříme počtem hran v této posloupnosti.
- *tah* je sled, ve kterém se neopakují hrany, tedy $e_i \neq e_j$ pro $i \neq j$.
- *cesta* je sled, ve kterém se neopakují vrcholy, čili $v_i \neq v_j$ pro $i \neq j$. Všimněte si, že se nemohou opakovat ani hrany.

Lehce nahlédneme, že pokud existuje sled z vrcholu x do y ($v_1 = x$, $v_n = y$), pak také existuje cesta z vrcholu x do vrcholu y. Každý sled, který není cestou, totiž obsahuje nějaký vrchol u dvakrát. Existuje tedy $i < j$ takové, že $u = v_i = v_j$. Pak ale můžeme z našeho sledu vypustit posloupnost $e_i, v_{i+1}, \ldots, e_{j-1}, v_j$ a dostaneme také sled spojující v_1 a v_n, který je určitě kratší než původní sled.

Tak můžeme po konečném počtu úprav dospět až ke sledu, který neobsahuje žádný vrchol dvakrát, tedy k cestě.

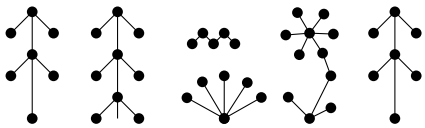
Kružnicí neboli *cyklem* nazýváme cestu délky alespoň 3, ve které oproti definici cesty platí $v_1 = v_n$. Někdy se na cesty, tahy a kružnice v grafu také díváme jako na podgrafy, které získáme tak, že z grafu vypustíme všechny ostatní vrcholy a hrany.

Ještě si ukážeme, že pokud existuje cesta z vrcholu a do vrcholu b a z vrcholu b do vrcholu c, pak také existuje cesta z vrcholu a do vrcholu c. To vyplývá z faktu, že existuje sled z vrcholu a do vrcholu c, který můžeme dostat například tak, že spojíme za sebe cesty z a do b a z b do c. A jak jsme si ukázali, když existuje sled z a do c, existuje i cesta z a do c.

V mnoha grafech (například v těch na předchozím obrázku) je každý vrchol dosažitelný cestou z každého. Takovým grafům budeme říkat *souvislé.* Pokud je graf nesouvislý, můžeme ho rozložit na části, které již souvislé jsou a mezi kterými nevedou žádné další hrany. Takové podgrafy nazýváme *komponentami souvislosti.*

Teď se podívejme na pár pojmů z přírody: *Strom* je souvislý graf, který neobsahuje kružnici. *List* je vrchol, ze kterého vede pouze jedna hrana. Ukážeme, že každý strom s alespoň dvěma vrcholy má nejméně dva listy. Proč to? Stačí si najít nejdelší cestu (pokud je takových cest více, zvolíme libovolnou z nich). Oba koncové vrcholy této cesty musí být nutně listy: kdyby z některého z nich vedla hrana, musela by vést do vrcholu, který na cestě ještě neleží (jinak by ve stromu byla kružnice), ale o takovou hranu bychom cestu mohli prodloužit, takže by původní cesta nebyla nejdelší.

Grafům bez kružnic budeme obecně říkat *lesy,* jelikož každá komponenta souvislosti takového grafu je strom.



Les, jak ho vidí matematici

Někdy se hodí jeden z vrcholů stromu prohlásit za *kořen,* čímž jsme si v každém vrcholu určili směr nahoru (ke kořeni – je to zvláštní, ale matematici obvykle kreslí stromy kořenem vzhůru) a dolů (od kořene). Souseda vrcholu směrem nahoru pak nazýváme jeho *otcem,* sousedy směrem dolů jeho *syny.*

Kostra souvislého grafu říkáme každému jeho podgrafu, který je stromem a spojuje všechny vrcholy grafu. Můžeme ji například získat tak, že dokud jsou v grafu kružnice, odebíráme hrany ležící na nějaké kružnici. Pro nesouvislé gra-

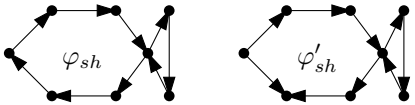
fy nazveme kostrou les tvořený kostrami jednotlivých komponent. Na prvním obrázku je jedna z koster levého grafu znázorněna silnými hranami.

Cvičení: Zkuste si dokázat, že stromy jsou právě grafy, které jsou souvislé a mají o jedna méně hran než vrcholů.

Orientované grafy

Často potřebujeme, aby hrany byly pouze jednosměrné. Takovému grafu říkáme *orientovaný graf*. Hrany jsou nyní uspořádané dvojice vrcholů (x, y) a říkáme, že hrana vede z vrcholu x do vrcholu y. Hrany (x, y) a (y, x) jsou tedy dvě různé hrany. Orientovaný graf většinou zobrazujeme jako body spojené šipkami. Většina pojmů, které jsme definovali pro neorientované grafy, dává smysl i pro grafy orientované, jen si musíme dát pozor na směr hran.

Se souvislostí orientovaných grafů je to trochu složitější. Rozlišujeme slabou a silnou souvislost: *slabě souvislý* je graf tehdy, pokud se z něj zapomenutím orientace hran stane souvislý neorientovaný graf. *Silně souvislým* ho nazveme tehdy, vede-li mezi každými dvěma vrcholy x, y orientovaná cesta v obou směrech. Pokud je graf silně souvislý, je i slabě souvislý, ale jak ukazuje náš obrázek, opačně to platit nemusí.



Silně a slabě souvislý orientovaný graf

Komponenta silné souvislosti orientovaného grafu G je takový podgraf G', který je silně souvislý a není podgrafem žádného většího silně souvislého podgrafu grafu G. Komponenty silné souvislosti tedy mohou být mezi sebou propojeny, ale žádné dvě nemohou ležet na společném cyklu.

Ohodnocené grafy

Další možností, jak si graf „vyzdobit“, je ohodnotit jeho hrany čísly. Například v grafu silniční sítě (vrcholy jsou města, hrany silnice mezi nimi) je zcela přirozené ohodnotit hrany délkami silnic nebo třeba mýtným vybíraným za průjezd silnicí. Přiřazeným číslům se proto často říká *délky* hran nebo jejich *ceny*. Pojmy, které jsme si před chvílí nadefinovali pro obyčejné grafy, můžeme opět snadno rozšířit pro grafy ohodnocené – např. délku sledu budeme namísto počtu hran sledu počítat jako součet jejich ohodnocení. Neohodnocený graf pak odpovídá grafu, v němž mají všechny hrany jednotkovou délku.

Podobně můžeme přiřazovat ohodnocení i vrcholům, ale raději si všechny operace s ohodnocenými grafy necháme na některé z dalších dílů Kuchařky. I tak budeme mít práce dost a dost.

Reprezentace grafů

Nyní už víme o grafech hodně, ale ještě jsme si neřekli, jak graf reprezentovat v paměti počítače. To můžeme udělat například tak, že vrcholy očíslujeme přirozenými čísly od 1 do N, hrany od 1 do M a odkud kam vedou hrany, popíšeme jedním z následujících tří způsobů:

- *matice sousednosti* – to je pole A velikosti $N \times N$. Na pozici $A[i,j]$ uložíme hodnotu 0 nebo 1 podle toho, zda z vrcholu i do vrcholu j vede hrana (1) nebo nevede (0). S maticí sousednosti se zachází velmi snadno, ale má tu nevýhodu, že je vždy kvadraticky velká bez ohledu na to, kolik je hran. Výhodou naopak je, že místo jedniček můžeme ukládat nějaké další informace o hranách, třeba jejich délky. Vpravo od tohoto odstavce najdete matici sousednosti grafu z prvního obrázku.

```
  123456789
1 011000011
2 100110001
3 100100000
4 011010000
5 010101000
6 000010110
7 000001011
8 100001100
9 110000100
```

- *seznam sousedů* je tvořen dvěma poli: polem sousedů $S[1 \ldots M]$ obsahujícím postupně čísla všech vrcholů, do kterých vede hrana z vrcholu 1, pak z vrcholu 2 atd., a polem začátků $Z[1 \ldots N]$, v němž se pro každý vrchol dozvíme začátek odpovídajícího úseku v poli S. Pokud navíc do $Z[N+1]$ uložíme $M+1$, bude platit, že sousedé vrcholu i jsou uloženi v $S[Z[i]], \ldots, S[Z[i+1]-1]$. Tato reprezentace má tu výhodu, že zabírá pouze prostor $O(N+M)$ a sousedy každého vrcholu máme pěkně pohromadě a nemusíme je hledat. Pro graf z 1. obrázku:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
$S[i]$	2	3	8	9	1	4	5	9	1	4	2	3	5	2	4	6	5	7	8	6	8	9	1	6	7	1	2	7

i	1	2	3	4	5	6	7	8	9	10
$Z[i]$	1	5	9	11	14	17	20	23	26	29

Reprezentace grafu seznamem sousedů

- *půlhranami* – tato reprezentace se používá tehdy, pokud potřebujeme během výpočtu graf složitě upravovat. Je univerzální, ale dost pracná na naprogramování. Spočívá v tom, že si každou hranu uložíme jako dvě půlhrany (začátek a konec hrany), každý vrchol bude obsahovat spojové seznamy přicházejících a odcházejících půlhran a každá půlhrana bude ukazovat na svou druhou polovici a na vrchol, ze kterého vychází.

V následujících receptech budeme vždy používat seznamy sousedů, poli S budeme říkat `Sousedi`, poli Z `Zacatky` a nadeklarujeme si je takto:

```
var N, M: Integer;  { počet vrcholů a hran }
    Zacatky: array[1..MaxN+1] of Integer;
    Sousedi: array[1..MaxM] of Integer;
```

Prohledávání do hloubky

Naše povídání o grafových algoritmech začneme dvěma základními způsoby procházení grafem. K tomu budeme potřebovat dvě podobné jednoduché datové struktury: *Fronta* je konečná posloupnost prvků, která má označený začátek a konec. Když do ní přidáváme nový prvek, přidáme ho na konec posloupnosti. Když z ní prvek odebíráme, odebereme ten na začátku. Proto se tato struktura anglicky nazývá *first in, first out*, zkráceně *FIFO*. *Zásobník* je také konečná posloupnost prvků se začátkem a koncem, ale zatímco prvky přidáváme také na konec, odebíráme je z téhož konce. Anglický název je (překvapivě) *last in, first out*, čili *LIFO*.

Algoritmus prohledávání grafu do hloubky:

1. Na začátku máme v zásobníku pouze vstupní vrchol w. Dále si u každého vrcholu v pamatujeme značku z_v, která říká, zda jsme vrchol již navštívili. Vstupní vrchol je označený, ostatní vrcholy nikoliv.
2. Odebereme vrchol ze zásobníku, nazvěme ho u.
3. Každý neoznačený vrchol, do kterého vede hrana z u, přidáme do zásobníku a označíme.
4. Kroky 2 a 3 opakujeme, dokud není zásobník prázdný.

Na konci algoritmu budou označeny všechny vrcholy dosažitelné z vrcholu w, tedy v případě neorientovaného grafu celá komponenta souvislosti obsahující w. To můžeme snadno dokázat sporem: Předpokládáme, že existuje vrchol x, který není označen, ale do kterého vede cesta z w. Pokud je takových vrcholů více, vezmeme si ten nejbližší k w. Označme si y předchůdce vrcholu x na nejkratší cestě z w; y je určitě označený (jinak by x nebyl nejbližší neoznačený). Vrchol y se tedy musel někdy objevit na zásobníku, tím pádem jsme ho také museli ze zásobníku odebrat a v kroku 3 označit všechny jeho sousedy, tedy i vrchol x, což je ovšem spor.

To, že algoritmus někdy skončí, nahlédneme snadno: v kroku 3 na zásobník přidáváme pouze vrcholy, které dosud nejsou označeny, a hned je značíme. Proto se každý vrchol může na zásobníku objevit nejvýše jednou, a jelikož ve 2. kroku pokaždé odebereme jeden vrchol ze zásobníku, musí vrcholy někdy (konkrétně po nejvýše N opakováních cyklu) dojít. Ve 3. kroku probereme každou hranu grafu nejvýše dvakrát (v každém směru jednou). Časová složitost celého algoritmu je tedy lineární v počtu vrcholů N a počtu hran M, čili

$O(N+M)$. Paměťová složitost je stejná, protože si tak jako tak musíme hrany a vrcholy pamatovat a zásobník není větší než paměť na vrcholy.

Prohledávání do hloubky implementujeme nejsnáze rekurzivní funkcí. Jako zásobník v tom případě používáme přímo zásobník programu, kde si program ukládá návratové adresy funkcí. Může to vypadat třeba následovně:

```
var Oznacen: array[1..MaxN] of Boolean;
procedure Projdi(V: Integer);
var I: Integer;
begin
  Oznacen[V] := True;
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if not Oznacen[Sousedi[I]] then
      Projdi(Sousedi[I]);
end;
```

Rozdělit neorientovaný graf na komponenty souvislosti je pak už jednoduché. Projdeme postupně všechny vrcholy grafu a pokud nejsou v žádné z dosud označených komponent grafu, přidáme novou komponentu tak, že graf z tohoto vrcholu prohledáme do hloubky. Vrcholy značíme přímo číslem komponenty, do které patří. Protože prohledáváme do hloubky několik oddělených částí grafu, každou se složitostí $O(N_i + M_i)$, kde N_i a M_i je počet vrcholů a hran komponenty, vyjde dohromady složitost $O(N+M)$. Nic nového si ukládat nemusíme, a proto je paměťová složitost stále $O(N+M)$.

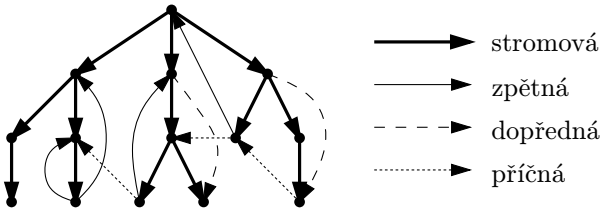
```
var Komponenta: array[1..MaxN] of Integer;
    NovaKomponenta: Integer;
procedure Projdi(V: Integer);
var I: Integer;
begin
  Komponenta[V] := NovaKomponenta;
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if Komponenta[Sousedi[I]] = -1 then
      Projdi(Sousedi[I]);
end;

var I: Integer;
begin
  ...
  for I := 1 to N do Komponenta[I] := -1;
  NovaKomponenta := 1;
  for I := 1 to N do
    if Komponenta[I] = -1 then
      begin
        Projdi(I);
        Inc(NovaKomponenta);
      end;
  ...
end.
```

Průběh prohledávání grafu do hloubky můžeme znázornit stromem (říká se mu DFS strom (podle anglického názvu Depth-First Search pro prohledávání do hloubky)). Z počátečního vrcholu w učiníme kořen. Pak budeme graf procházet do hloubky a vrcholy zakreslovat jako syny vrcholů, ze kterých jsme přišli. Syny každého vrcholu si uspořádáme v pořadí, v němž jsme je navštívili; tomuto pořadí budeme říkat *zleva doprava* a také ho tak budeme kreslit. Hranám mezi otci a syny budeme říkat *stromové hrany*. Protože jsme do žádného vrcholu nešli dvakrát, budou opravdu tvořit strom. Hrany, které vedou do již navštívených vrcholů na cestě, kterou jsme přišli z kořene, nazveme *zpětné hrany. Dopředné hrany* vedou naopak z vrcholu blíže kořeni do už označeného vrcholu dále od kořene. A konečně *příčné hrany* vedou mezi dvěma různými podstromy grafu.

Všimněte si, že při prohledávání neorientovaného grafu objevíme každou hranu dvakrát: buďto poprvé jako stromovou a podruhé jako zpětnou, a nebo jednou jako zpětnou a podruhé jako dopřednou. Příčné hrany se objevit nemohou – pokud by příčná hrana vedla doprava, vedla by do dosud neoznačeného vrcholu, takže by se prohledávání vydalo touto hranou a nevznikl by oddělený podstrom; doleva rovněž vést nemůže: představme si stav prohledávání v okamžiku, kdy jsme opouštěli levý vrchol této hrany. Tehdy by naše hrana musela být příčnou vedoucí doprava, ale o té už víme, že neexistuje.

Prohledávání do hloubky lze tedy také využít k nalezení kostry neorientovaného grafu, což je strom, který jsme prošli. Rovnou při tom také zjistíme, zda graf neobsahuje cyklus: to poznáme tak, že nalezneme zpětnou hranu různou od té stromové, po níž jsme do vrcholu přišli.

Pro orientované grafy je situace opět trochu složitější: stromové a dopředné hrany jsou orientované vždy ve stromě shora dolů, zpětné zdola nahoru a příčné hrany mohou existovat, ovšem vždy vedou zprava doleva, čili pouze do podstromů, které jsme již prošli (nahlédneme opět stejně).



Strom prohledávání do hloubky a typy hran

Prohledávání do šířky

Prohledávání do šířky je založené na podobné myšlence jako prohledávání do hloubky, pouze místo zásobníku používá frontu:

1. Na začátku máme ve frontě pouze jeden prvek, a to zadaný vrchol w. Dále si u každého vrcholu x pamatujeme číslo $H[x]$. Všechny vrcholy budou mít na začátku $H[x] = -1$, jen $H[w] = 0$.
2. Odebereme vrchol z fronty, označme ho u.
3. Každý vrchol v, do kterého vede hrana z u a jeho $H[v] = -1$, přidáme do fronty a nastavíme jeho $H[v]$ na $H[u] + 1$.
4. Kroky 2 a 3 opakujeme, dokud není fronta prázdná.

Podobně jako u prohledávání do hloubky jsme se dostali právě do těch vrcholů, do kterých vede cesta z w (a označili jsme je nezápornými čísly). Rovněž je každému vrcholu přiřazeno nezáporné číslo maximálně jednou. To vše se dokazuje podobně, jako jsme dokázali správnost prohledávání do hloubky.

Vrcholy se stejným číslem tvoří ve frontě jeden souvislý celek, protože nejprve odebereme z fronty všechny vrcholy s číslem n, než začneme odebírat vrcholy s číslem $n+1$. Navíc platí, že $H[v]$ udává délku nejkratší cesty z vrcholu w do v. Že neexistuje kratší cesta, dokážeme sporem: Pokud existuje nějaký vrchol v, pro který $H[v]$ neodpovídá délce nejkratší cesty z w do v, čili vzdálenosti $D[v]$, vybereme si z takových v to, jehož $D[v]$ je nejmenší. Pak nalezneme nejkratší cestu z w do v a její předposlední vrchol z. Vrchol z je bližší než v, takže pro něj už musí být $D[z] = H[z]$. Ovšem když jsme z fronty vrchol z odebírali, museli jsme objevit i jeho souseda v, který ještě nemohl být označený, tudíž jsme mu museli přidělit $H[v] = H[z] + 1 = D[v]$, a to je spor.

Prohledávání do šířky má časovou složitost taktéž lineární s počtem hran a vrcholů. Na každou hranu se také ptáme dvakrát. Fronta má lineární velikost k počtu vrcholů, takže jsme si oproti prohledávání do hloubky nepohoršili a i paměťová složitost je $O(N+M)$. Algoritmus implementujeme nejsnáze cyklem, který bude pracovat s vrcholy v poli představujícím frontu.

```
var Fronta, H: array[1..MaxN] of Integer;
    I, V, Prvni, Posledni: Integer;
    PocatecniVrchol: Integer;
begin
  ...
  for I := 1 to N do H[I] := -1;
  Prvni := 1;
  Posledni := 1;
  Fronta[Prvni] := PocatecniVrchol;
  H[PocatecniVrchol] := 0;

  repeat
    V := Fronta[Prvni];
    for I := Zacatky[V] to Zacatky[V+1]-1 do
      if H[Sousedi[I]] < 0 then begin
        H[Sousedi[I]] := H[V]+1;
        Inc(Posledni);
        Fronta[Posledni] := Sousedi[I];
```

```
      end;
    Inc(Prvni);
  until Prvni > Posledni;  { Fronta je prázdná }
  ...
end.
```

Prohledávání do šířky lze také použít na hledání komponent souvislosti a hledání kostry grafu.

Topologické uspořádání

Teď si vysvětlíme, co je *topologické uspořádání* grafu. Máme orientovaný graf G s N vrcholy a chceme očíslovat vrcholy čísly 1 až N tak, aby všechny hrany vedly z vrcholu s větším číslem do vrcholu s menším číslem, tedy aby pro každou hranu $e = (v_i, v_j)$ bylo $i > j$. Představme si to jako srovnání vrcholů grafu na přímku tak, aby „šipky“ vedly pouze zprava doleva.

Nejprve si ukážeme, že pro žádný orientovaný graf, který obsahuje cyklus, nelze takovéto topologické pořadí vytvořit. Označme vrcholy cyklu $v_1, \ldots, v_n$, takže hrana vede z vrcholu v_i do vrcholu v_{i-1}, resp. z v_1 do v_n. Pak vrchol v_2 musí dostat vyšší číslo než vrchol v_1, v_3 než $v_2, \ldots, v_n$ než v_{n-1}. Ale vrchol v_1 musí mít zároveň vyšší číslo než v_n, což nelze splnit.

Cyklus je ovšem to jediné, co muže existenci topologického uspořádání zabránit. Libovolný acyklický graf lze uspořádat následujícím algoritmem:

1. Na začátku máme orientovaný graf G a proměnnou $p = 1$.
2. Najdeme takový vrchol v, ze kterého nevede žádná hrana (budeme mu říkat *stok*). Pokud v grafu žádný stok není, výpočet končí, protože jsme našli cyklus.
3. Odebereme z grafu vrchol v a všechny hrany, které do něj vedou.
4. Přiřadíme vrcholu v číslo p.
5. Proměnnou p zvýšíme o 1.
6. Opakujeme kroky 2 až 5, dokud graf obsahuje alespoň jeden vrchol.

Proč tento algoritmus funguje? Pokud v grafu nalezneme stok, můžeme mu určitě přiřadit číslo menší než všem ostatním vrcholům, protože překážet by nám v tom mohly pouze hrany vedoucí ze stoku ven a ty neexistují. Jakmile stok očíslujeme, můžeme jej z grafu odstranit a pokračovat číslováním ostatních vrcholů. Tento postup musí někdy skončit, jelikož v grafu je pouze konečně mnoho vrcholů.

Zbývá si uvědomit, že v neprázdném grafu, který neobsahuje cyklus, vždy existuje alespoň jeden stok: Vezměme libovolný vrchol v_1. Pokud z něj vede nějaká hrana, pokračujme po ní do nějakého vrcholu v_2, z něj do v_3 atd. Co se při tom může stát?

- Dostaneme se do vrcholu v_i, ze kterého nevede žádná hrana. Vyhráli jsme, máme stok.

- Narazíme na v_i, ve kterém jsme už jednou byli. To by ale znamenalo, že graf obsahuje cyklus, což, jak víme, není pravda.
- Budeme objevovat stále a nové a nové vrcholy. V konečném grafu nemožno.

Algoritmus můžeme navíc snadno upravit tak, aby netratil příliš času hledáním vrcholů, z nichž nic nevede – stačí si takové vrcholy pamatovat ve frontě a kdykoliv nějaký takový vrchol odstraňujeme, zkontrolovat si, zda jsme nějakému jinému vrcholu nezrušili poslední hranu, která z něj vedla, a pokud ano, přidat takový vrchol na konec fronty. Celé topologické třídění pak zvládneme v čase $O(N + M)$.

Jiná možnost je prohledat graf do hloubky a všimnout si, že pořadí, ve kterém jsme se z vrcholů vraceli, je právě topologické pořadí. Pokud zrovna opouštíme nějaký vrchol a číslujeme ho dalším číslem v pořadí, rozmysleme si, jaké druhy hran z něj mohou vést: stromová nebo dopředná hrana vede do vrcholu, kterému jsme již přiřadili nižší číslo, zpětná existovat nemůže (v grafu by byl cyklus) a příčné hrany vedou pouze zprava doleva, takže také do již očíslovaných vrcholů. Časová složitost je opět $O(N + M)$.

```
var Ocislovani: array[1..MaxN] of Integer;
    Posledni: Integer;
    I: Integer;

procedure Projdi(V: Integer);
var I: Integer;
begin
  Ocislovani[V] := 0; { zatím V jen označíme }
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if Ocislovani[Sousedi[I]] = -1 then
      Projdi(Sousedi[I]);
  Inc(Posledni);
  Ocislovani[V] := Posledni;
end;

begin
  ...
  for I := 1 to N do
    Ocislovani[I] := -1;
  Posledni := 0;
  for I := 1 to N do
    if Ocislovani[I] = -1 then Projdi(I);
  ...
end.
```

Hranová a vrcholová 2-souvislost

Nyní se podíváme na trochu komplikovanější formu souvislosti. Říkáme, že neorientovaný graf je *hranově 2-souvislý*, když platí, že:

- má alespoň 3 vrcholy,

- je souvislý,
- zůstane souvislý po odebrání libovolné hrany.

Hranu, jejíž odebrání by způsobilo zvýšení počtu komponent souvislosti grafu, nazýváme *most.*

Na hledání mostů nám poslouží opět upravené prohledávání do hloubky a DFS strom. Všimněme si, že mostem může být jedině stromová hrana – každá jiná hrana totiž leží na nějaké kružnici. Odebráním mostu se graf rozpadne na část obsahující kořen DFS stromu a podstrom „visící“ pod touto hranou. Jediné, co tomu může zabránit, je existence nějaké další hrany mezi podstromem a hlavní částí, což musí být zpětná hrana, navíc taková, která není jenom stromovou hranou viděnou z druhé strany. Takovým hranám budeme říkat *ryzí zpětné hrany.*

Proto si pro každý vrchol spočítáme hladinu, ve které se nachází (kořen je na hladině 0, jeho synové na hladině 1, jejich synové 2, ...). Dále si pro každý vrchol v spočítáme, do jaké nejvyšší hladiny (s nejmenším číslem) vedou ryzí zpětné hrany z podstromu s kořenem v. To můžeme udělat přímo při procházení do hloubky, protože než se vrátíme z v, projdeme celý podstrom pod v. Pokud všechny zpětné hrany vedou do hladiny stejné nebo větší než té, na které je v, pak odebráním hrany vedoucí do v z jeho otce vzniknou dvě komponenty souvislosti, čili tato hrana je mostem. V opačném případě jsme nalezli kružnici, na níž tato hrana leží, takže to most být nemůže. Výjimku tvoří kořen, který žádného otce nemá a nemusíme se o něj proto starat.

Algoritmus je tedy pouhou modifikací procházení do hloubky a má i stejnou časovou a paměťovou složitost $O(N + M)$. Zde jsou důležité části programu:

```
var Hladina, Spojeno: array[1..MaxN] of Integer;
    DvojSouvisle: Boolean;
    I: Integer;

procedure Projdi(V, NovaHladina: Integer);
var I, W: Integer;
begin
  Hladina[V] := NovaHladina;
  Spojeno[V] := Hladina[V];

  for I := Zacatky[V] to Zacatky[V+1]-1 do
  begin
    W := Sousedi[I];
    if Hladina[W] = -1 then
    begin { stromová hrana }
      Projdi(W, NovaHladina + 1);
      if Spojeno[W] < Spojeno[V] then
        Spojeno[V] := Spojeno[W];
      if Spojeno[W] > Hladina[V] then
        DvojSouvisle := False; { máme most }
    end else { zpětná nebo dopředná hrana }
```

```
      if (Hladina[W] < NovaHladina-1) and
         (Hladina[W] < Spojeno[V]) then
        Spojeno[V] := Hladina[W];
    end;
  end;

  begin
    ...
    for I := 1 to N do
      Hladina[I] := -1;
    DvojSouvisle := True;
    Projdi(1, 0);
    ...
  end.
```

Další formou souvislosti je *vrcholová souvislost*. Graf je *vrcholově 2-souvislý*, právě když:

- má alespoň 3 vrcholy,
- je souvislý,
- zůstane souvislý po odebrání libovolného vrcholu.

Artikulace je takový vrchol, který když odebereme, zvýší se počet komponent souvislosti grafu.

Algoritmus pro zjištění vrcholové 2-souvislosti grafu je velmi podobný algoritmu na zjišťování hranové 2-souvislosti. Jen si musíme uvědomit, že odebíráme celý vrchol. Ze stromu procházení do hloubky může odebráním vrcholu vzniknout až několik podstromů, které všechny musí být spojeny zpětnou hranou s hlavním stromem. Proto musí zpětné hrany z podstromu určeného vrcholem v vést až *nad* vrchol v. Speciálně pro kořen nám vychází, že může mít pouze jednoho syna, jinak bychom ho mohli odebrat a vytvořit tak dvě nebo více komponent souvislosti. Algoritmus se od hledání hranové 2-souvislosti liší jedinou změnou ostré nerovnosti na neostrou, sami zkuste najít, které nerovnosti.

20-4-K Kuchařka čtvrté série – halda a Dijsktrův algoritmus

Dnešní povídání o algoritmech a datových strukturách se bude zabývat jedním z nejznámějších algoritmů: Dijkstrovým algoritmem pro hledání nejkratších cest v grafech. K tomu (a nejenom k tomu) se nám bude hodit šikovná datová strukturka zvaná halda, tak si předvedeme nejdříve ji.

Halda

Halda je datová struktura pro uchovávání množiny čísel (či jakýchkoliv jiných prvků, na kterých máme definováno uspořádání, tj. umíme pro každou dvojici prvků říci, který z nich je menší). Tato datová struktura obvykle podporuje následující operace: Přidání nového prvku, nalezení nejmenšího prvku a

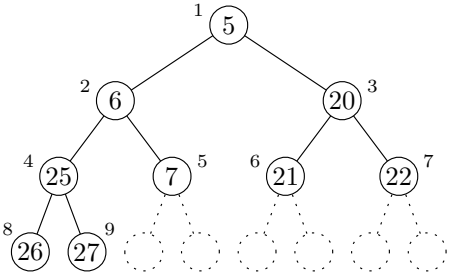
odebrání nejmenšího prvku. My si ukážeme jednoduchou implementaci haldy, která bude při uložení N prvků potřebovat čas $O(\log N)$ na přidání či odebrání jednoho prvku a $O(1)$ (tj. konstantní) na zjištění hodnoty nejmenšího prvku.

Naše implementace bude vypadat následovně: Pokud halda obsahuje N prvků, uložíme její prvky do pole na pozice 1 až N. Prvek na pozici k bude mít dva *následníky,* a to prvky na pozicích $2k$ a $2k + 1$; samozřejmě, pokud je k velké, a tedy např. $2k + 1 > N$, má takový prvek jen jednoho či dokonce žádného následníka. Naopak prvek na pozici $\lfloor k/2 \rfloor$ nazveme *předchůdcem* prvku na pozici k. Ti z vás, kteří znají binární stromy, v tomto jistě rozpoznali způsob, jak v poli uchovávat úplné binární stromy (následníci jsou synové a předchůdci otcové v obvyklé stromové terminologii, prvek č. 1 je kořen stromu).

Prvky haldy však v poli neuchováváme v úplně libovolném pořadí. Chceme, aby platilo, že každý prvek je menší nebo roven všem svým následníkům. Naše halda tedy může vypadat např. takto:

1	2	3	4	5	6	7	8	9
5	6	20	25	7	21	22	26	27

Tomu odpovídá tento strom:



Z toho, co jsme si právě popsali, je jasné, že nejmenší prvek je uložen na pozici s indexem 1, a tedy můžeme snadno v konstantním čase zjistit jeho hodnotu. Ještě prozradíme, jak lze prvky do haldy rychle přidávat a odebírat:

Jestliže halda obsahuje N prvků, pak nový prvek, říkejme mu třeba x, přidáme na konec pole, tj. na pozici s indexem N. Nyní x porovnáme s jeho předchůdcem. Pokud je jeho předchůdce menší, je vše v pořádku a jsme hotovi. V opačném případě x s jeho předchůdcem prohodíme. Tím jsme problém napravili, ale nyní může být x menší než jeho nový předchůdce. To lze napravit dalším prohozením a tak budeme pokračovat dále, než se buďto dostaneme do situace, kdy už je x větší nebo rovno svému předchůdci, nebo „vybublá“ až do kořene haldy, kde už žádného předchůdce nemá. Protože se v každém kroku pozice, na níž se prvek x právě nachází, zmenší alespoň na polovinu, provedeme dohromady nejvýše $O(\log N)$ výměn, a tedy spotřebujeme čas $O(\log N)$.

Odebírání nejmenšího prvku probíhá podobně: Prvek z poslední pozice (tj. z pozice N) přesuneme na pozici 1, tedy místo minima. Místo s předchůdci jej však porovnáme s jeho následníky a v případě, že je větší než některý z jeho následníků, opět je prohodíme (pokud je větší než oba následníci, prohodíme ho s menším z nich). A protože se nám v každém kroku index „bublajícího“ prvku v poli alespoň zdvojnásobí, opět spotřebujeme čas $O(\log N)$.

Jako cvičení si rozmyslete, že v čase $O(\log N)$ lze z haldy smazat dokonce libovolný prvek, pokud si ovšem pamatujeme, kde se v haldě nachází. Také můžeme prvek ponechat a jen změnit jeho hodnotu.

Ještě si předvedeme program:

```
var halda: array[1..MAX] of integer;
    N: integer;            { počet prvků v haldě }

function nejmensi: integer;
begin
  nejmensi:=halda[1]
end;

procedure vloz(prvek: integer);
var i, x: integer;
begin
  i:=N; N:=N+1;
  halda[i]:=prvek;
  while (i>1) and (halda[i div 2]>halda[i]) do begin
    x:=halda[i div 2];
    halda[i div 2]:=halda[i];
    halda[i]:=x;
    i:=i div 2
  end
end;

procedure smaz_nejmensi;
var i, j, x: integer;
begin
  halda[1]:=halda[N];
  N:=N-1; i:=1;
  while 2*i<=N do begin
    j:=i;
    if halda[j]>halda[2*i] then j:=2*i;
    if (2*i+1<=N) and (halda[j]>halda[2*i+1]) then j:=2*i+1;
    if i=j then break;
    x:=halda[i]; halda[i]:=halda[j]; halda[j]:=x;
    i:=j
  end
end;
```

HeapSort

Když už máme k dispozici haldu, můžeme pomocí ní například snadno třídit čísla. Máme-li N čísel, která chceme setřídit, vytvoříme si z nich nejprve haldu o N prvcích (například postupným vkládáním do prázdné haldy), načež z ní budeme postupně N-krát odebírat nejmenší prvek. Tím získáme prvky původního pole v rostoucím pořadí. Celkově provedeme N vložení, N nalezení minima a N smazání. To vše dohromady stihneme v čase $O(N \log N)$.

Než si ukážeme program, přidáme ještě dva triky, které nám implementaci značně usnadní. Předně si vše uložíme do jednoho pole – to bude při plnění haldy obsahovat na svém začátku haldu a na konci zbytek vstupního pole, přitom zbytek pole se bude postupně zmenšovat a uvolňovat tak místo haldě; naopak v druhé polovině algoritmu budeme zmenšovat haldu a do volného prostoru ukládat setříděné prvky. K tomu se nám bude hodit získávat prvky v opačném pořadí, proto si upravíme haldu tak, aby udržovala nikoliv minimum, nýbrž maximum.

Druhý trik spočívá v tom, že nebudeme haldu vytvářet postupným vkládáním, nýbrž naopak zabubláváním prvků (podobným, jako děláme při mazání minima) od konce. Všimněte si, že takto také získáme správné nerovnosti mezi prvky a jejich následníky, a dokonce tak zvládneme celou haldu vytvořit v lineárním čase [proč to tak je, si zkuste dokázat sami, stačí si uvědomit, kolikrát zabubláváme které prvky]. Zbytek třídění bohužel nadále zůstává $O(N \log N)$.

Tomuto algoritmu se obvykle říká *HeapSort* (čili třídění haldou) a je jedním z mála známých rychlých třídících algoritmů, které nepotřebují pomocnou paměť.

```
type Pole = array[1..MAXN] of Integer;
procedure HeapSort(var A: Pole);
var i, x: integer;
   procedure bublej(m, i: integer); { "zabublání" prvku }
   { m je velikost haldy, i je index zabublávaného prvku }
   var j, x: integer;
   begin
      while 2*i<=m do begin
         j:=2*i;
         if (j<m) and (A[j+1]>A[j]) then j:=j+1;
         if A[i]>=A[j] then break;
         x:=A[i]; A[i]:=A[j]; A[j]:=x;
         i:=j;
      end;
   end;
begin
   for i:=N div 2 downto 1 do bublej(N,i); { bublej }
   for i:=N downto 2 do begin { vybírej maximum }
      x:=A[1]; A[1]:=A[i]; A[i]:=x;
      bublej(i-1, 1);
end; end;
```

Dijkstrův algoritmus

Nyní již konečně k slíbenému *Dijkstrovu algoritmu.* Tento algoritmus dostane orientovaný graf s hranami ohodnocenými nezápornými čísly (viz kuchařka v minulé sérii) a nalezne v něm nejkratší cestu mezi dvěma zadanými vrcholy. Ve skutečnosti tento algoritmus dělá o malinko více: Najde totiž nejkratší cesty z jednoho zadaného vrcholu do všech ostatních.

Nechť v_0 je vrchol grafu, ze kterého chceme určit délky nejkratších cest. Budeme si udržovat pole délek zatím nalezených cest z vrcholu v_0 do všech ostatních vrcholů grafu. Navíc u některých vrcholů budeme mít poznamenáno, že cesta nalezená do nich je už ta nejkratší možná. Takovým vrcholům budeme říkat *definitivní.* Na začátku inicializujeme v poli všechny hodnoty na ∞ kromě hodnoty odpovídající vrcholu v_0, kterou inicializujeme na 0 (délka nejkratší cesty z v_0 do v_0 je 0). V každém *kroku* algoritmu pak provedeme následující: Vybereme vrchol w, který ještě není definitivní, a mezi všemi takovými vrcholy je délka zatím nalezené cesty do něj nejkratší možná. Vrchol w prohlásíme za definitivní. Dále otestujeme, zda pro nějaký vrchol v cesta z vrcholu v_0 do w a pak po hraně z w do v není kratší, než zatím nalezená cesta z v_0 do v, a je-li tomu tak, upravíme délku zatím nalezené cesty do v. Toto provedeme pro všechny takové vrcholy v. Celý algoritmus skončí, pokud jsou už všechny vrcholy definitivní nebo všechny vrcholy, co nejsou definitivní, mají délku cesty rovnou ∞ (v takovém případě se graf skládá z více nesouvislých částí).

Předtím než dokážeme, že právě představený algoritmus opravdu nalezne délky nejkratších cest z vrcholu v_0, se zamysleme nad jeho časovou složitostí.

Pro každý z N vrcholů si délku dosud nalezené cesty uchováme v poli. Celý algoritmus má nejvýše N kroků, protože v každém kroku nám přibyde jeden definitivní vrchol. Ten vybíráme z jako minimum z délky aktuální cesty přes všechny dosud nedefinitivní vrcholy, kterých je $O(N)$. V každému kroku musíme zkontrolovat tolik vrcholů v, kolik hran vede z vrcholu w. Počet takových změn pro všechny kroky dohromady je pak nejvýše $O(M)$, kde M je počet hran vstupního grafu. Z toho vyjde časová složitost $O(N^2 + M)$, čili $O(N^2)$, jelikož M je nejvýše N^2. Tuto implementaci Dijkstrova algoritmu najdete na konci naší kuchařky.

K uchovávání délek dosud nalezených nejkratších cest můžeme ovšem použít haldu. Ta bude na začátku obsahovat N prvků a v každém kroku se počet jejích prvků sníží o jeden: Nalezneme a smažeme nejmenší prvek, to zvládneme v čase $O(\log N)$, a případně upravíme délky nejkratších cest do sousedů právě zpracovávaného vrcholu. To pro každou hranu trvá rovněž $O(\log N)$, celkově za všechny hrany tedy $O(M \log N)$. Z toho vyjde celková časová složitost algoritmu $O((N + M) \log N)$, a to je pro „řídké“ grafy (tedy grafy s $M \ll N^2$) výrazně lepší.

Vraťme se nyní k důkazu správnosti Dijkstrova algoritmu. Ukážeme, že po každém kroku algoritmu platí následující tvrzení: Nechť A je množina definitivních vrcholů. Pak délka dosud nalezené cesty z v_0 do v (v je libovolný vrchol grafu) je délka nejkratší cesty $v_0v_1 \ldots v_kv$ takové, že všechny vrcholy $v_0, v_1, \ldots, v_k$ jsou v množině A. Tvrzení dokážeme indukcí dle počtu kroků algoritmu, které již proběhly. Tvrzení zřejmě platí před a po prvním kroku algoritmu. Nechť w je vrchol, který byl v předchozím kroku prohlášen za definitivní. Uvažme nejprve nějaký vrchol v, který je definitivní. Pokud $v = w$, tvrzení je triviální. V opačném případě ukážeme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w. Označme D délku cesty z v_0 do v přes vrcholy A bez vrcholu w. Protože v každém kroku vybíráme vrchol s nejmenším ohodnocením a ohodnocení vybraných vrcholů v jednotlivých krocích tvoří neklesající posloupnost (váhy hran jsou nezáporné!), tak délka cesty z v_0 do w přes vrcholy z A je alespoň D. Ale potom délka libovolné cesty z v_0 do v přes w používající vrcholy z A je alespoň D. Z volby D pak víme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w.

Nyní uvažme takový vrchol v, který není definitivní. Nechť $v_0v_1 \ldots v_kv$ je nejkratší cesta z v_0 do v taková, že všechny vrcholy $v_0, v_1, \ldots, v_k$ jsou v množině A. Pokud $v_k = w$, pak jsme ohodnocení v změnili na délku této cesty v právě proběhlém kroku. Pokud $v_k \neq w$, pak $v_0v_1, \ldots, v_k$ je nejkratší cesta z v_0 do v_k přes vrcholy z množiny A a tedy můžeme předpokládat, že žádný z vrcholů $v_1, \ldots, v_k$ není w (podle toho, co jsme si rozmysleli na konci minulého odstavce). Potom se ale délka cesty do v rovnala správné hodnotě už před právě proběhlým krokem.

Vzhledem k tomu, že po posledním kroku množina A obsahuje právě ty vrcholy, do kterých existuje cesta z vrcholu v_0, dokázali jsme, že náš algoritmus funguje správně.

Na závěr ještě poznamenejme, že Dijkstrův algoritmus funguje je možné snadno upravit tak, aby nám kromě délky nejkratší cesty i takovou cestu našel: U každého vrcholu si v okamžiku, kdy mu měníme ohodnocení, poznamenáme, ze kterého vrcholu do něj přicházíme. Nejkratší cestu do nějakého vrcholu pak zrekonstruujeme tak, že u posledního vrcholu této cesty zjistíme, který vrchol je předposlední, u předposledního, který je předpředposlední, atd.

Poznámka pro zvídavé: existují i jiné druhy hald, například k-regulární haldy, v nichž má každý prvek k následníků (rozmyslete si, jaká je v takové haldě časová složitost operací a jak nastavit k v závislosti na M a N, aby byl Dijkstrův algoritmus co nejrychlejší), nebo tzv. Fibonacciho halda, která dokáže upravit hodnotu prvku v konstantním čase. S tou pak umíme hledat nejkratší cesty v čase $O(M + N \log N)$.

Implementace Dijkstrova algoritmu

```
var N: word; { počet vrcholů }
vahy: array[1..MAX, 1..MAX] of integer; { váhy hran, -1=hrana neex.}
delky: array[1..MAX] of integer;     { délky zatím nalezených cest,}
                                              { -1 = nekonečno }
def: array[1..MAX] of boolean;        { definitivní? }

procedure Dijkstra(odkud: word);
var i, w, v: word;
begin
  for i:=1 to N do begin
    def[i]:=false;  delky[i]:=-1;
  end;
  def[odkud]:=true;
  delky[odkud]:=0;
  repeat
    w:=0;
    for i:=1 to N do
      if not def[i] and ((w=0) or (delky[i]<delky[w])) then w:=i;
    if w<>0 then begin
      def[w]:=true;
      for i:=1 to N do
        if (vahy[w][i]<>-1) and (delky[w]+vahy[w][i]<delky[i]) then
        delky[i]:=delky[w]+vahy[w][i]
    end
  until w=0;
end;
```

20-5-K Kuchařka páté série – vyhledávací stromy

V nedávném vydání programátorské kuchařky jsme se zabývali tříděním dat, tentokráte si povíme, jak v uspořádaných datech něco efektivně najít a jak si data udržovat stále uspořádaná. K tomu se nám bude hodit zejména binární vyhledávání a různé druhy vyhledávacích stromů.

Binární vyhledávání. Představte si, že jste k narozeninám dostali obrovské pole setříděných záznamů (to je, pravda, trochu netradiční dárek, ale proč ne – může to být třeba telefonní seznam). Záznamy mohou vypadat libovolně a to, že jsou setříděné, znamená jen a pouze, že $x_1 < x_2 < \ldots < x_N$, kde $<$ je nějaká relace, která nám řekne, který ze dvou záznamů je menší (pro jednoduchost předpokládáme, že žádné dva záznamy nejsou stejné).

Co si ale s takovým polem počneme? Zkusíme si v něm najít nějaký konkrétní záznam z. To můžeme udělat třeba tak, že si nalistujeme prostřední záznam (označíme si ho x_m) a porovnáme s ním naše z. Pokud $z < x_m$, víme, že se z nemůže vyskytovat „napravo“ od x_m, protože tam jsou všechny záznamy větší než x_m a tím spíše než z. Analogicky pokud $z > x_m$, nemůže se z vyskytovat v první polovině pole. V obou případech nám zbude jedna polovina a v ní budeme pokračovat stejným způsobem. Tak budeme postupně zmenšovat

interval, ve kterém se z může nacházet, až buďto z najdeme nebo vyloučíme všechny prvky, kde by mohlo být.

Tomuto principu se obvykle říká *binární vyhledávání* nebo také *hledání půlením intervalu* a snadno ho naprogramujeme buďto rekursivně nebo pomocí cyklu, v němž si budeme udržovat interval $\langle l, r\rangle$, ve kterém se hledaný prvek může nacházet:

```
function BinSearch(z : integer):integer;
var l,r,m : integer;
begin
  l := 1;   { interval, ve kterém hledáme }
  r := N;
  while l <= r do begin { ještě není prázdný }
    m := (l+r) div 2;   { střed intervalu }
    if z < x[m] then
      r := m-1          { je vlevo }
    else if z > x[m] then
      l := m+1          { je vpravo }
    else begin          { Bingo! }
      hledej := m;
      exit;
    end;
  end;
  hledej := -1;         { nebyl nikde }
end;
```

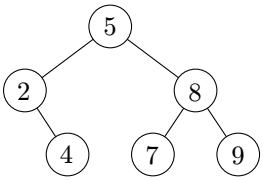
Všimněte si, že průchodů cyklem `while` může být nejvýše $\lceil \log_2 N \rceil$, protože interval $\langle l, r\rangle$ na počátku obsahuje N prvků a v každém průchodu jej zmenšíme na polovinu (ve skutečnosti ještě o jedničku, ale tím lépe pro nás). Proto po k průchodech bude interval obsahovat nejvýše $N/2^k$ prvků a jelikož pro $N/2^k < 1$ se algoritmus zastaví, může být k nejvýše $\log_2 N$. Proto je časová složitost binárního vyhledávání $O(\log N)$. [Základ logaritmu nemusíme psát, protože $\log_a b = \log_c b / \log_c a$, čili logaritmy o různých základech se liší jen konstantou, která se „schová do O-čka.“]

Hledání půlením intervalu je tedy velmi rychlé, pokud máme možnost si data předem setřídit. Jakmile ale potřebujeme za běhu programu přidávat a odebírat záznamy, potážeme se se zlou: buďto budeme mít záznamy uložené v poli, a pak nezbývá než při zatřiďování nového prvku ostatní „rozhrnout“, což může trvat až N kroků, a nebo si je budeme udržovat v nějakém seznamu, do kterého dokážeme přidávat v konstantním čase, jenže pak pro změnu nebudeme při vyhledávání schopni najít tolik potřebnou polovinu.

Zkusme ale provést jednoduchý myšlenkový pokus:

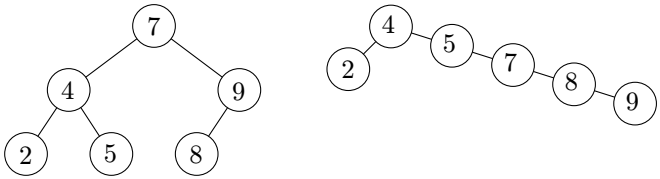
Vyhledávací stromy. Představme si, jakými všemi možnými cestami se může v našem poli binární vyhledávání ubírat. Na začátku porovnáváme s prostředním prvkem a podle výsledku se vydáme jednou ze dvou možných cest (nebo rovnou zjistíme, že se jedná o hledaný prvek, ale to není moc zajímavý

případ). Na každé cestě nás zase čeká porovnání se středem příslušného intervalu a to nás opět pošle jednou ze dvou dalších cest atd. To si můžeme přehledně popsat pomocí stromu:



Jeden vrchol stromu prohlásíme za kořen a ten bude odpovídat celému poli (a jeho prostřednímu prvku). K němu budou připojené vrcholy obou polovin pole (opět obsahující příslušné prostřední prvky) a tak dále. Ovšem jakmile známe tento strom, můžeme náš půlící algoritmus provádět přímo podle stromu (ani k tomu nepotřebujeme vidět původní pole a umět v něm hledat poloviny): začneme v kořeni, porovnáme a podle výsledku se buďto přesuneme do levého nebo pravého podstromu a tak dále. Každý průběh algoritmu bude tedy odpovídat nějaké cestě z kořene stromu do hledaného vrcholu.

Teď si ale všimněte, že aby hledání hodnoty podle stromu fungovalo, strom vůbec nemusel vzniknout půlením intervalu – stačilo, aby v každém vrcholu platilo, že všechny hodnoty v levém podstromu jsou menší než tento vrchol a naopak hodnoty v pravém podstromu větší. Hledání v témže poli by také popisovaly následující stromy (např.):



Hledací algoritmus podle jiných stromů samozřejmě už nemusí mít pěknou logaritmickou složitost (když bychom hledali podle „degenerovaného“ stromu z pravého obrázku, trvalo by to dokonce lineárně). Důležité ale je, že takovéto stromy se dají poměrně snadno modifikovat a že je při troše šikovnosti můžeme udržet dostatečně podobné ideálnímu půlení intervalu. Pak bude hloubka stromu stále $O(\log N)$, tím pádem i časová složitost hledání a, jak za chvilku uvidíme, mnohých dalších operací.

Definice. Zkusme si tedy pořádně nadefinovat to, co jsme právě vymysleli:

Binární vyhledávací strom (po domácku BVS) je buďto prázdná množina nebo *kořen* obsahující jednu hodnotu a mající dva *podstromy* (levý a pravý), což jsou opět BVS, ovšem takové, že všechny hodnoty uložené v levém podstromu

jsou menší než hodnota v kořeni, a ta je naopak menší než všechny hodnoty uložené v pravém podstromu.

Úmluva: Pokud x je kořen a L_x a R_x jeho levý a pravý podstrom, pak kořenům těchto podstromů (pokud nejsou prázdné) budeme říkat *levý a pravý syn* vrcholu x a naopak vrcholu x budeme říkat *otec* těchto synů. Pokud je některý z podstromů prázdný, pak vrchol x příslušného syna nemá. Vrcholu, který nemá žádné syny, budeme říkat *list* vyhledávacího stromu. Všimněte si, že pokud x má jen jediného syna, musíme stále rozlišovat, je-li to syn levý nebo pravý, protože potřebujeme udržet správné uspořádání hodnot. Také si všimněte, že pokud známe syny každého vrcholu, můžeme již rekonstruovat všechny podstromy.

Každý BVS také můžeme popsat velmi jednoduchou strukturou v paměti:

```
type pvrchol = ^vrchol;
     vrchol = record
       l, r : pvrchol;  { levý a pravý syn }
       x    : integer;  { hodnota }
     end;
```

Pokud některý ze synů neexistuje, zapíšeme do příslušné položky hodnotu *nil*.

Find. V řeči BVS můžeme přeformulovat náš vyhledávací algoritmus takto:

```
function TreeFind(v:pvrchol; x:integer):pvrchol;
{ Dostane kořen stromu a hodnotu. Vrátí vrchol,
  kde se hodnota nachází, nebo nil, není-li. }
begin
  while (v<>nil) and (v^.x<>x) do begin
    if x<v^.x then
      v := v^.l
    else
      v := v^.r
  end;
  TreeFind := v;
end;
```

Funkce `TreeFind` bude pracovat v čase $O(h)$, kde h je hloubka stromu, protože začíná v kořeni a v každém průchodu cyklem postoupí o jednu hladinu níže.

Insert. Co kdybychom chtěli do stromu vložit novou hodnotu (aniž bychom se teď starali o to, zda tím strom nemůže degenerovat)? Stačí zkusit hodnotu najít a pokud tam ještě nebyla, určitě při hledání narazíme na odbočku, která je *nil*. A přesně na toto místo připojíme nově vytvořený vrchol, aby byl správně uspořádán vzhledem k ostatním vrcholům (že tomu tak je, plyne z toho, že při hledání jsme postupně vyloučili všechna ostatní místa, kde nová hodnota být nemohla). Naprogramujeme opět snadno, tentokráte si ukážeme rekurzivní zacházení se stromy:

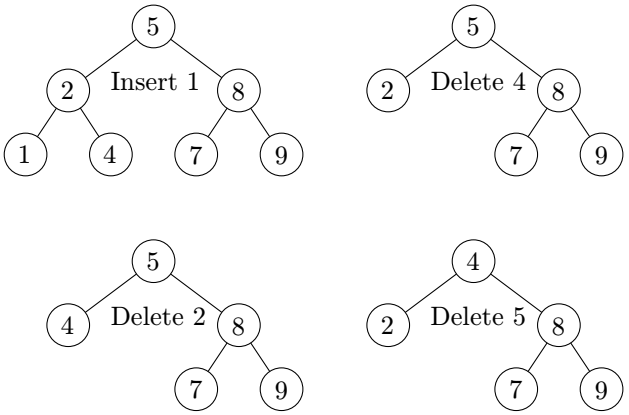
```
function TreeIns(v:pvrchol; x:integer):pvrchol;
{ Dostane kořen stromu a hodnotu ke vložení,
  vrátí nový kořen. }
begin
  if v=nil then begin
    { prázdný strom => založíme nový kořen }
    new(v);
    v^.l := nil;
    v^.r := nil;
    v^.x := x;
  end else if x<v^.x then  { vkládáme vlevo }
    v^.l := TreeIns(v^.l, x)
  else if x>v^.x then      { vkládáme vpravo }
    v^.r := TreeIns(v^.r, x);
  TreeIns := v;
end;
```

Delete. Mazání bude o kousíček pracnější, musíme totiž rozlišit tři případy: Pokud je mazaný vrchol list, stačí ho vyměnit za *nil*. Pokud má právě jednoho syna, stačí náš vrchol v ze stromu odstranit a syna přepojit k otci v. A pokud má syny dva, najdeme největší hodnotu v levém podstromu (tu najdeme tak, že půjdeme jednou doleva a pak pořád doprava), umístíme ji do stromu namísto mazaného vrcholu a v levém podstromu ji pak smažeme (což už umíme, protože má 1 nebo 0 synů). Program následuje:

```
function TreeDel(v:pvrchol; x:integer):pvrchol;
{ Parametry stejně jako TreeIns }
var w:pvrchol;
begin
  TreeDel := v;
  if v=nil then exit         { prázdný strom }
  else if x<v^.x then
    v^.l := TreeDel(v^.l, x) { ještě hledáme x }
  else if x>v^.x then
    v^.r := TreeDel(v^.r, x)
  else begin                 { našli jsme }
    if (v^.l=nil) and (v^.r=nil) then begin
      TreeDel := nil;        { mažeme list }
      dispose(v);
    end else if v^.l=nil then begin
      TreeDel := v^.r;       { jen pravý syn }
      dispose(v);
    end else if v^.r=nil then begin
      TreeDel := v^.l;       { jen levý }
      dispose(v);
    end else begin           { má oba syny }
      w := v^.l;             { hledáme max(L) }
      while w^.r<>nil do w := w^.r;
      v^.x := w^.x;          { prohazujeme }
      { a mažeme původní max(L) }
      v^.l := TreeDel(v^.l, w^.x);
```

```
    end;
  end;
end;
```

Když do stromu z našeho prvního obrázku zkusíme přidávat nebo z něj odebírat prvky, dopadne to takto:



Jak vkládání, tak mazání opět budou trvat $O(h)$. Ale pozor, jejich používáním může h nekontrolovatelně růst – sami zkuste najít nějaký příklad, kdy h dosáhne až N.

Procházení stromu. Pokud bychom chtěli všechny hodnoty ve stromu vypsat, stačí strom rekursivně projít a sama definice uspořádání hodnot ve stromu nám zajistí, že hodnoty vypíšeme ve vzestupném pořadí: nejdříve levý podstrom, pak kořen a pak podstrom pravý. Časová složitost je, jak se snadno nahlédne, lineární, protože strávíme konstantní čas vypisováním každého prvku a prvků je právě N. Program bude opět přímočarý:

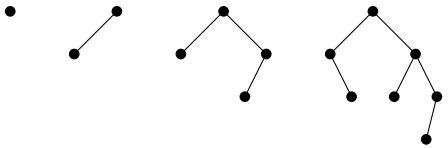
```
procedure TreeShow(v:pvrchol);
begin
  if v=nil then exit;  { není co dělat }
  TreeShow(v^.l);
  writeln(v^.x);
  TreeShow(v^.r);
end;
```

Vyvážené stromy. S binárními stromy lze dělat všelijaká kouzla a prakticky všechny stromové algoritmy mají společné to, že jejich časová složitost je lineární v hloubce stromu. (Pravda, právě ten poslední byl výjimka, leč všechny prvky rychleji než lineárně s N opravdu nevypíšeme.) Jenže jak jsme viděli, neopatrným insertováním a deletováním prvků mohou snadno vznikat všelijaké degenerované stromy, které mají lineární hloubku. Abychom tomu zabránili, musíme stromy *vyvažovat.* To znamená definovat si nějaké šikovné omezení

na tvar stromu, aby hloubka byla vždy $O(\log N)$. Možností je mnoho, my uvedeme jen ty nejdůležitější:

Dokonale vyvážený budeme říkat takovému stromu, ve kterém pro každý vrchol platí, že počet vrcholů v jeho levém a pravém podstromu se liší nejvýše o jedničku. Takové stromy kopírují dělení na poloviny při binárním vyhledávání, a proto (jak jsme již dokázali) mají vždy logaritmickou hloubku. Jediné, čím se liší, je, že mohou zaokrouhlovat na obě strany, zatímco náš půlící algoritmus zaokrouhloval polovinu vždy dolů, takže levý podstrom nemohl být nikdy větší než pravý. Z toho také plyne, že se snadnou modifikací půlícího algoritmu dá dokonale vyvážený BVS v lineárním čase vytvořit ze setříděného pole. Bohužel se ale při Insertu a Deletu nedá v logaritmickém čase strom znovu vyvážit.

AVL stromy. Zkusíme tedy vyvažovací podmínku trochu uvolnit a vyžadovat, aby se u každého vrcholu lišily o jedničku nikoliv velikosti podstromů, nýbrž pouze jejich hloubky. Takovým stromům se říká *AVL stromy* a mohou vypadat třeba takto:



Každý dokonale vyvážený strom je také AVL stromem, ale jak je vidět na předchozím obrázku, opačně to platit nemusí. To, že hloubka AVL stromů je také logaritmická, proto není úplně zřejmé a zaslouží si to trochu dokazování:

Věta: AVL strom o N vrcholech má hloubku $O(\log N)$.

Důkaz: Označme A_d nejmenší možný počet vrcholů, jaký může být v AVL stromu hloubky d. Snadno vyzkoušíme, že $A_1 = 1$, $A_2 = 2$, $A_3 = 4$ a $A_4 = 7$ (příslušné minimální stromy najdete na předchozím obrázku). Navíc platí, že $A_d = 1 + A_{d-1} + A_{d-2}$, protože každý minimální strom hloubky d musí mít kořen a 2 podstromy, které budou opět minimální, protože jinak bychom je mohli vyměnit za minimální a tím snížit počet vrcholů celého stromu. Navíc alespoň jeden z podstromů musí mít hloubku $d-1$ (protože jinak by hloubka celého stromu nebyla d) a druhý hloubku $d-2$ (podle definice AVL stromu může mít $d-1$ nebo $d-2$, ale s menší hloubkou bude mít evidentně méně vrcholů).

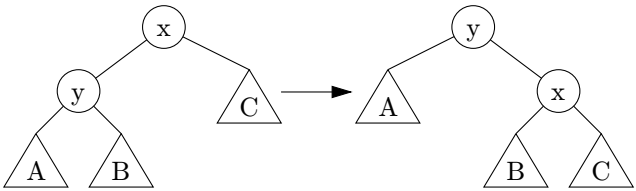
Spočítat, kolik přesně je A_d, není úplně snadné. Nám však postačí dokázat, že $A_d \geq 2^{d/2}$. To provedeme indukcí: Pro $d < 4$ to plyne z ručně spočítaných hodnot. Pro $d \geq 4$ je $A_d = 1 + A_{d-1} + A_{d-2} > 2^{(d-1)/2} + 2^{(d-2)/2} = 2^{d/2} \cdot (2^{-1/2} + 2^{-1}) > 2^{d/2}$ (součet čísel v závorce je ≈ 1.207).

Jakmile už víme, že A_d roste s d alespoň exponenciálně, tedy že $\exists c : A_d \geq c^d$, důkaz je u konce: Máme-li AVL strom T na N vrcholech, najdeme si nejmenší d takové, že $A_d \leq N$. Hloubka stromu T může být maximálně d, protože jinak

by T musel mít alespoň A_{d+1} vrcholů, ale to je více než N. A jelikož A_d rostou exponenciálně, je $d \leq \log_c N$, čili $d = O(\log N)$. *Q.E.D.*

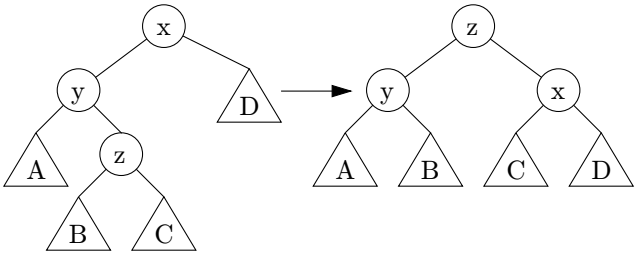
AVL stromy tedy vypadají nadějně, jen stále nevíme, jak provádět Insert a Delete tak, strom zůstal vyvážený. Nemůžeme si totiž dovolit strukturu stromu měnit libovolně – stále musíme dodržovat správné uspořádání hodnot. K tomu se nám bude hodit zavést si nějakou množinu operací, o kterých dokážeme, že jsou korektní, a pak strukturu stromu měnit vždy jen pomocí těchto operací. Budou to:

Rotace. Rotací binárního stromu (respektive nějakého podstromu) nazveme jeho „překořenění“ za některého ze synů kořene. Místo formální definice ukažme raději obrázek:



Strom jsme překořenili za vrchol y a přepojili jednotlivé podstromy tak, aby byly vzhledem k x a y opět uspořádané správně (všimněte si, že je jen jediný způsob, jak to udělat). Jelikož se tím okolí vrcholu y „otočilo“ po směru hodinových ručiček, říká se takové operaci *rotace doprava.* Inverzní operaci (tj. překořenění za pravého syna kořene) se říká *rotace doleva* a na našem obrázku odpovídá přechodu zprava doleva.

Dvojrotace. Také si nakreslíme, jak to dopadne, když provedeme dvě rotace nad sebou lišící se směrem (tj. jednu levou a jednu pravou nebo opačně). Tomu se říká *dvojrotace* a jejím výsledkem je překořenění podstromu za vnuka kořene připojeného „cikcak“. Raději opět předvedeme na obrázku:



Znaménka. Při vyvažování se nám bude hodit pamatovat si u každého vrcholu, v jakém vztahu jsou hloubky jeho podstromů. Tomu budeme říkat *znaménko* vrcholu a bude buďto 0, jsou-li oba stejně hluboké, − pro levý podstrom hlubší a + pro pravý hlubší. V textu budeme znaménka, respektive vrcholy se znaménky značit $\oplus$, $\ominus$ a $\odot$.

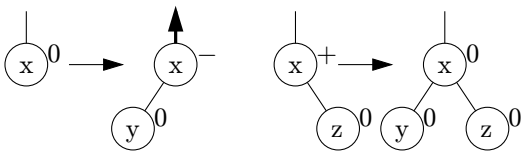
Pokud celý strom zrcadlově obrátíme (prohodíme levou a pravou stranu), znaménka se změní na opačná ($\oplus$ a $\ominus$ se prohodí, $\odot$ zůstane). Toho budeme často využívat a ze dvou zrcadlově symetrických situací popíšeme jenom jednu s tím, že druhá se v algoritmu zpracuje symetricky.

Často také budeme potřebovat nalézt otce nějakého vrcholu. To můžeme zařídit buďto tak, že si do záznamů popisujících vrcholy stromu přidáme ještě ukazatele na otce a budeme ho ve všech operacích poctivě aktualizovat, a nebo využijeme toho, že jsme do daného vrcholu museli někudy přijít z kořene, a celou cestu z kořene si zapamatujeme v nějakém zásobníku a postupně se budeme vracet.

Tím jsme si připravili všechny potřebné ingredience, tož s chutí do toho:

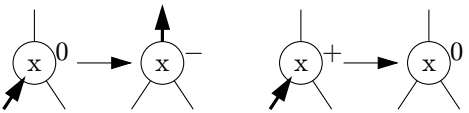
Vyvažování po Insertu. Když provedeme Insert tak, jak jsme ho popisovali u obecných vyhledávacích stromů, přibude nám ve stromu list. Pokud se tím AVL vyváženost neporušila, stačí pouze opravit znaménka na cestě z nového listu do kořene (všude jinde zůstala zachována). Pakliže porušila, musíme s tím něco provést, konkrétně ji šikovně zvolenými rotacemi opravit. Popíšeme algoritmus, který bude postupovat od listu ke kořeni a vše potřebné zařídí:

Nejprve přidání listu samotné:

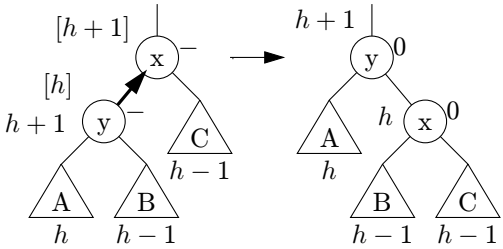


Pokud jsme přidali list (bez újmy na obecnosti levý, jinak vyřešíme zrcadlově) vrcholu se znaménkem $\odot$, změníme znaménko na $\ominus$ a pošleme o patro výš informaci o tom, že hloubka podstromu se zvýšila (to budeme značit šipkou). Přidali-li jsme list k $\oplus$, změní se na $\odot$ a hloubka podstromu se nemění, takže můžeme skončit.

Nyní rozebereme případy, které mohou nastat na vyšších hladinách, když nám z nějakého podstromu přijde šipka. Opět budeme předpokládat, že přišla zleva; pokud zprava, vyřešíme zrcadlově. Pokud přišla do $\oplus$ nebo $\odot$, ošetříme to stejně jako při přidání listu:

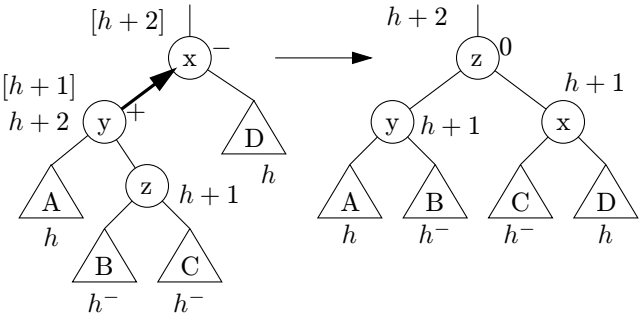


Pokud ale vrchol x má znaménko $\ominus$, nastanou potíže: levý podstrom má teď hloubku o 2 vyšší než pravý, takže musíme rotovat. Proto se podíváme o patro níž, jaké je znaménko vrcholu y pod šipkou, abychom věděli, jakou rotaci provést. Jedna možnost je tato (y je $\ominus$):



Tehdy provedeme jednoduchou rotaci vpravo. Jak to dopadne s hloubkami jsme přikreslili do obrázku – pokud si hloubku podstromu A označíme jako h, B musí mít hloubku $h-1$, protože y je $\ominus$, atd. Jen nesmíme zapomenout, že v x jsme ještě $\ominus$ nepřepočítali (vede tam přeci šipka), takže ve skutečnosti je jeho levý podstrom o 2 hladiny hlubší než pravý (původní hloubky jsme na obrázku naznačili [v závorkách]). Po zrotování vyjdou u x i y znaménka $\odot$ a celková hloubka se nezmění, takže jsme hotovi.

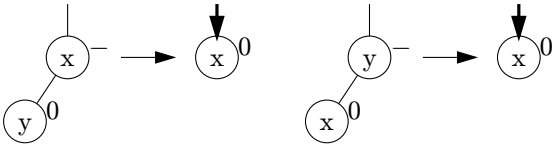
Další možnost je y jako $\oplus$:



Tehdy se podíváme ještě o hladinu níž a provedeme dvojrotaci. (Nemůže se nám stát, že by z neexistovalo, protože jinak by v y nebylo $\oplus$.) Hloubky opět najdete na obrázku. Jelikož z může mít libovolné znaménko, jsou hloubky podstromů B a C buďto h nebo $h-1$, což značíme h^-. Podle toho pak vyjdou znaménka vrcholů x a y po rotaci. Každopádně vrchol z vždy obdrží $\odot$ a celková hloubka se nemění, takže končíme.

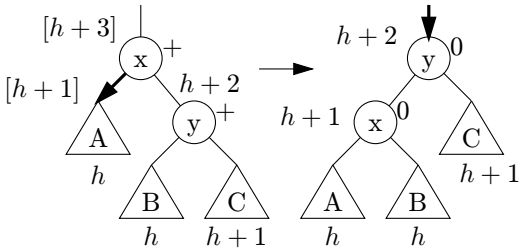
Poslední možnost je, že by y byl $\odot$, ale tu vyřešíme velmi snadno: všimneme si, že nemůže nastat. Kdykoliv totiž posíláme šipku nahoru, není pod ní $\odot$. (Kontrolní otázka: jak to, že $\oplus$ může nastat?)

Vyvažování po Deletu. Vyvažování po Deletu je trochu obtížnější, ale také se dá popsat pár obrázky. Nejdříve opět rozebereme základní situace: odebíráme list (BÚNO levý) nebo vnitřní vrchol stupně 2 (tehdy ale musí být jeho jediný syn listem, jinak by to nebyl AVL strom):



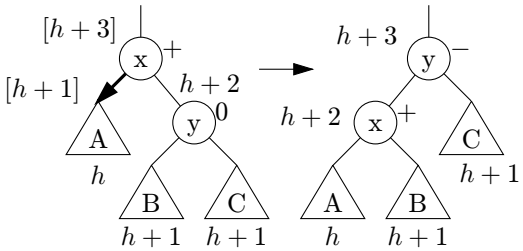
Šipkou dolů značíme, že o patro výš posíláme informaci o tom, že se hloubka podstromu snížila o 1. Pokud šipku dostane vrchol typu $\ominus$ nebo $\odot$, vyřešíme to snadno:

Problematické jsou tentokráte ty případy, kdy šipku dostane $\oplus$. Tehdy se musíme podívat na znaménko *opačného* syna a podle toho rotovat. První možnost je, že opačný syn má $\oplus$:



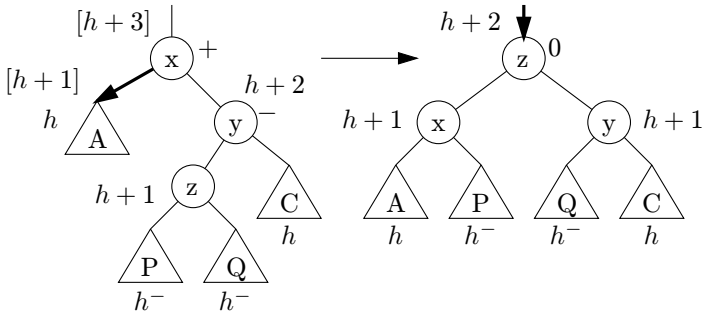
Tehdy provedeme rotaci vlevo, x i y získají nuly, ale celková hloubka stromu se sníží o hladinu, takže nezbývá, než poslat šipku o patro výš.

Pokud by y byl $\odot$:



Opět rotace vlevo, ale tentokráte se zastavíme, protože celková hloubka se nezměnila.

Poslední, nejkomplikovanější možnost je, že by y byl $\ominus$:



V tomto případě provedeme dvojrotaci (z určitě existuje, jelikož y je typu $\ominus$), vrcholy x a y obdrží znaménka v závislosti na původním znaménku vrcholu z a celý strom se snížil, takže pokračujeme o patro výš.

Happy end. Jak při Insertu, tak při Deletu se nám podařilo strom upravit tak, aby byl opět AVL stromem, a trvalo nám to lineárně s hloubkou stromu (konáme konstantní práci na každé hladině), čili stejně jako trvá Insert a Delete samotný. Jenže o AVL stromech jsme již dokázali, že mají hloubku vždy logaritmickou, takže jak hledání, tak Insert a Delete zvládneme v logaritmickém čase (vzhledem k aktuálnímu počtu prvků ve stromu).

Další typy stromů. AVL stromy samozřejmě nejsou jediný způsob, jak zavést stromovou datovou strukturu s logaritmicky rychlými operacemi. Další jsou třeba:

- *Červeno-černé stromy* – ty si místo znamének vrcholy barví, každý je buďto červený nebo černý a platí, že nikdy nejsou dva červené vrcholy pod sebou a že na každé cestě z kořene do listu je stejný počet černých vrcholů. Opět je hloubka stromu logaritmická, po Insertu a Deletu barvy opravujeme přebarvováním na cestě do kořene a rotováním, jen je potřeba rozebrat podstatně více případů než u AVL stromů. (Za to jsme ale odměněni tím, že nikdy neděláme více než 2 rotace.)
- *2-3-stromy* – v jednom vrcholu nemáme uloženu jednu hodnotu, nýbrž jednu nebo dvě (a synové jsou pak 2 nebo 3, odtud název), a navíc přidáme pravidlo, že všechny listy jsou na téže hladině. Hloubka vyjde opět logaritmická, vyvažování řešíme pomocí spojování a rozdělování vrcholů.
- *Splay stromy* – nezavádíme žádnou vyvažovací podmínku, nýbrž definujeme, že kdykoliv pracujeme s nějakým vrcholem, vždy si jej vyrotujeme do kořene a pokud to jde, preferujeme dvojrotace. Takové operaci se říká Splay a dají se pomocí ní definovat operace ostatní: Find hodnotu najde a poté na ni zavolá Splay. Insert si nechá vysplayovat předchůdce nové hodnoty a vloží nový vrchol mezi předchůdce a jeho pravého syna. Delete vysplayuje mazaný prvek, pak

uvnitř pravého podstromu vysplayuje minimum, takže bude mít jen jednoho syna a můžeme jím tedy nahradit mazaný prvek v kořeni.

Jednotlivé operace samozřejmě mohou trvat až lineárně dlouho, ale dá se o nich dokázat, že jejich *amortizovaná* složitost je vždy $O(\log N)$. Tím chceme říci, že provést t po sobě jdoucích operací začínajících prázdným stromem trvá $O(t \cdot \log N)$ (některé operace mohou být pomalejší, ale to je vykoupeno větší rychlostí jiných). To u většiny použití stačí – datovou strukturu obvykle používáte uvnitř nějakého algoritmu a zajímá vás, jak dlouho běží všechny operace dohromady – a navíc je Splay stromy daleko snazší naprogramovat než nějaké vyvažované stromy. Mimo to mají Splay stromy i jiné krásné vlastnosti: přizpůsobují svůj tvar četnostem hledání, takže často hledané prvky jsou pak blíž ke kořeni, snadno se dají rozdělovat a spojovat atd.

- *Treapy* – randomizovaně vyvažované stromy: něco mezi stromem (tree) a haldou (heap). Každému prvku přiřadíme *váhu,* což je náhodné číslo z intervalu $\langle 0, 1\rangle$. Strom pak udržujeme uspořádaný stromově podle hodnot a haldově podle vah (všimněte si, že tím je jeho tvar určen jednoznačně, pokud tedy jsou všechny váhy navzájem různé, což skoro jistě jsou). Insert a Delete opravují haldové uspořádání velmi jednoduše pomocí rotací. Časová složitost v průměrném případě je $O(\log N)$.
- *BB-α stromy* – zobecnění dokonalé vyváženosti jiným směrem: zvolíme si vhodné číslo α a vyžadujeme, aby se velikost podstromů každého vrcholu lišila maximálně α-krát (prázdné podstromy nějak ošetříme, abychom nedělili nulou; dokonalá vyváženost odpovídá $\alpha = 1$ [až na zaokrouhlování]). V každém vrcholu si budeme pamatovat, kolik vrcholů obsahuje podstrom, jehož je kořenem, a po Insertu a Deletu přepočítáme tyto hodnoty na cestě zpět do kořene a zkontrolujeme, jestli je strom ještě stále α-vyvážený. Pokud ne, najdeme nejvyšší místo, ve kterém se velikosti podstromů příliš liší, a vše od tohoto místa dolů znovu vytvoříme algoritmem na výrobu dokonale vyvážených stromů. Ten, pravda, běží v lineárním čase, ale čím větší podstrom přebudováváme, tím to děláme méně často, takže vyjde opět amortizovaně $O(\log N)$ na operaci.

Cvičení. Několik věcí, které se do kuchařky už nevešly, ale můžete si je zkusit vymyslet:

1. jak konstruovat dokonale vyvážené stromy
2. jak pomocí toho naprogramovat BB-α stromy

3. algoritmus, který k prvku ve stromu najde jeho následníka, což je prvek s nejbližší vyšší hodnotou (zde předpokládejte, že ke každému prvku máte uložený ukazatel na jeho otce)
4. jak vypsat celý strom tak, že začnete v minimu a budete postupně hledat následníky (i když nalezení následníka může trvat až $O(h)$, všimněte si, že projití celého stromu přes následníky bude lineární)
5. jak do vrcholů stromu ukládat různé pomocné informace, jako třeba počet vrcholů v podstromu každého vrcholu, a jak tyto informace při operacích se stromem udržovat (při Insertu, Deletu, rotaci)
6. že libovolný interval $\langle a, b\rangle$ lze rozložit na logaritmicky mnoho intervalů odpovídajících podstromům
7. a že zkombinováním předchozích dvou cvičení lze odpovídat i na otázky typu „kolik si strom pamatuje hodnot ze zadaného intervalu“ v logaritmickém čase ...

Několik poznámek na závěr.

- Pokud záznamy můžeme jenom porovnávat, je binární vyhledávání nejlepší možné. Libovolné hledání založené na porovnávání lze totiž popsat binárním stromem a binární strom s N vrcholy musí mít vždy hloubku alespoň $\lfloor \log_2 N \rfloor$.
- Pokud bychom ale předpokládali, že se záznamy můžeme zacházet i jinak, dají se některé operace provádět i v konstantním čase (alespoň průměrně). K tomu se hodí například *hashování,* a to si popíšeme v některé z kuchařek v příštím ročníku KSP (nebo se můžete podívat do kuchařky u 4. série 19. ročníku). Jeho nevýhodou ovšem je, že udržuje jenom množinu prvků, nikoliv uspořádání na ní, takže například nelze najít k zadanému prvku nejbližší vyšší.
- Pokud bychom připustili, že se mohou vyskytnout dva stejné záznamy, budou stromy stále fungovat, jen si musíme dát o něco větší pozor na to, co všechno při operacích se stromem může nastat.
- Jakpak přišly AVL stromy ke svému jménu? Inu, podle svých objevitelů pánů Adelsona-Velského a Landise.
- Rekurenci $A_d = 1 + A_{d-1} + A_{d-2}$, $A_1 = 1$, $A_2 = 2$ pro velikosti minimálních AVL stromů je samozřejmě možné vyřešit i přesně. Žádné překvapení se nekoná, objeví se totiž stará známá Fibonacciho čísla: $A_n = F_{n+2} - 1$.

Vzorová řešení

20-1-1 Temná šachovnice — Mária Vámošová

„Ach jo, co si to ti mágové dnes nevymyslí," povzdechl si Vilda a začal vysvětlovat temnému pánovi, proč jeho příkazu nemůže vyhovět:

Na šachovnici 4×4, a vůbec na všech šachovnicích o sudém rozměru, se při výchozím obarvení nachází sudý počet jak černých, tak bílých políček. Zaměřme se třeba na černá. („No proto!" pochvaloval si mág.)

Dokážu, že když je na takové šachovnici před tahem počet černých políček sudý, po tahu bude sudý také. Při libovolném tahu mohu změnit počet černých políček následovně: V daném sloupci/řádku byl počet černých políček sudý, což znamená, že bílých bylo také sudě. Takže když se po tahu tyto počty prohodí, zůstanu pořád na sudém počtu černých políček v tomhle sloupci/řádku. Protože byl ale celkový počet černých políček před tahem sudý, tak když odečtu sudý počet původně černých políček a přičtu sudý počet nově černých políček, dostanu opět sudé číslo.

Naopak, mohlo se mi stát, že počet černých políček v daném sloupci/řádku byl lichý, což také znamená, že bílých políček bylo také liše. Když se po tahu tyto počty prohodí, celkem dostanu opět sudý počet černých políček (od původního počtu odečtu liché číslo, ale pak zase jiné liché číslo přičtu).

„Z toho bohužel vyplývá, že šachovnici na požadovaný tvar upravit nedokážu – ve výsledku mám dostat lichý počet černých políček, což opravdu nejde," dokončil svůj výklad Vilda. „Hm," zamyslel se mág, „chápu, že pro šachovnice o sudém rozměru to nejde, nepůjde to ale pro šachovnice s lichým počtem políček na straně?" snažil se přesvědčit Vildu a začal si připravovat pilku pro případné ořezání šachovnice.

„No, to bohužel nepůjde také," zklamal ho Vilda a pokračoval ve vysvětlování:

Každá šachovnice o rozměru větším než 2 má v levém horním rohu *podšachovnici* o rozměru 2×2. Tuto podšachovnici potřebujeme obarvit tak, aby levé horní políčko bylo bíle a zbylá tři černá. Když se na chvíli zamyslíme, zjistíme, že na změnu libovolného políčka podšachovnice mají vliv jenom inverze prvních dvou sloupců/řádků. Odmysleme si zbytek těchto sloupců/řádků, jsme opět tam kde jsme byli – snažíme se přebarvit šachovnici o straně 2 a o té už víme, že přebarvit nejde.

„Mám však taky jednu dobrou zprávu," rozjasnil se Vilda, „šachovnici 1×1 lze obarvit velice snadno!" načež si od mága vypůjčil pilku, vyřezal levé horní políčko a obarvil ho bíle. Mág chvíli na šachovnici hleděl silně podezřívavým

pohledem, pak sáhl do kapsy, vytáhl figurku dámy a položil ji na políčko. „Tak, zahrajem si?“

20-1-2 Příprava na cestu — Jana Kravalová

Každého jistě napadne, že bychom v KSP nevypisovali úlohu za 12 bodů, kdyby se měla řešit prostým prohledáváním všech možností neboli obyčejným backtrackem. Nezachrání nás ani různé heuristiky a pokusy sem tam odřezat neperspektivní větev výpočtu. My si ukážeme, že pomocí šikovného ukládání již spočítaných výsledků dokážeme najít řešení v polynomiálním čase.

Celá myšlenka spočívá v následující úvaze: Představme si, že bychom věděli, jak nejlépe vyřešit tuto úlohu pro $N-1$ úkolů a zadaný počet minut. Tedy že pro $N-1$ úkolů a daný počet minut víme, kolik minut věnovat kterému úkolu a jaká bude výsledná pravděpodobnost, že bude mág s Vildovou prací spokojen. (Zatím nemusíme moc přemýšlet nad tím, jak jsme k této znalosti přišli, prostě ji máme.) Kdyby nám teď někdo přidal N-tý, poslední úkol a dal nám na vyřešení všech úkolů M minut, stačí udělat jednoduchou věc: zkusíme poslednímu N-tému úkolu postupně přidělit všechny možné počty minut, které přichází v úvahu (tedy $k = 0..M$ minut) a na zbylých $N-1$ úkolů použít zbylých $M-k$ minut. Pravděpodobnost, že bude mág spokojen, bude součinem čísla a_{Nk} (na úkol N jsme použili k minut) a nejlepší pravděpodobnosti, kterou můžeme na $N-1$ úkolech při $M-k$ minutách dosáhnout (toto číslo máme dopředu spočítané). Pak nám stačí ze všech možných k vybrat to nejlepší.

Dobře, ale jak spočítáme nejlepší pravděpodobnost, které můžeme dosáhnout na $N-1$ úkolech pro nějaký zadaný počet minut? No, představme si, že toto číslo už známe pro $N-2$ úkolů ...

A teď si algoritmus představíme pořádně:

Máme zadánu matici $A = aij$ kde prvek a_{ij} udává, jaká je pravděpodobnost, že bude mág s Vildovým řešením úkolu i spokojen, pokud mu Vilda věnuje j minut. Zavedeme si matici $P = pij$, kde prvek p_{ij} bude znamenat nejlepší možnou pravděpodobnost, jakou můžeme dosáhnout na úkolech $1..i$ při přidělených minutách j. Hodnota p_{23} tedy například udává, jaká je nejlepší možná pravděpodobnost mágovy spokojenosti pro úkoly 1 a 2 s přiděleným počtem minut 3.

Matici P budeme zaplňovat po řádcích od levého horního rohu, který odpovídá nejlepší pravděpodobnosti pro 1 úkol a 0 minut, až do pravého dolního rohu, který odpovídá nejlepší pravděpodobnosti pro N úkolů a M minut a je vlastně řešením naší úlohy.

Chceme zaplnit pole p_{ij}, které by mělo odpovídat nejlepší možné pravděpodobnosti pro úkoly $1..i$ a přidělené minuty j. Trošku jsme to naznačili už v úvodní myšlence. Vyzkoušíme všechna možná přidělení minut $k = 0..j$ pro poslední i-tou úlohu a do hodnoty p_{ij} dosadíme maximum z hodnot $a_{ik} \cdot p_{i-1,j-k}$.

Toto nejlepší k si také uložíme, protože pro nás znamená nejlepší přidělené minuty pro tento konkrétní úkol a na konci ho budeme potřebovat vypsat.

Ještě zbývá vymyslet, jak zaplníme první řádek matice P. Můžeme si pomoct tím, že si na začátek přidáme nultý řádek, který bude na pozicích p_{00} až p_{0M} zaplněn samými jedničkami, což si také můžeme vyložit tak, že hodnoty označují mágovu spokojenost, když nebude žádný úkol a bude pro něj k dispozici 0–M minut. Potom můžeme uvedený algoritmus nastartovat od prvního řádku matice (od prvního úkolu).

Protože musíme zaplnit matici P, která má N řádků a $M+1$ sloupců a protože v každém políčku musíme udělat $O(M)$ testů, má celý algoritmus časovou složitost $O(NM^2)$.

```
#include <stdio.h>

int main() {
int N, M;
scanf("%d %d",&N,&M);
double a[N+1][M+1], p[N+1][M+1];
int m[N+1][M+1];

for (int i = 0; i < N; i++) // trik - obrátíme pořadí pro jednodušší vypisování
        for (int j = 0; j <= M; j++) scanf("%lf ",&a[N-i][j]);

for (int i = 0; i <= M; i++) p[0][i] = 1;        // neexistují žádné úkoly
for (int i = 1; i <= N; i++)                     // dynamika
        for (int j = 0; j <= M; j++) {
                int maxk = 0;                    // přidělíme úkolu i 0 minut
                for (int k = 1; k <= j; k++)
                        // zkusíme úkolu i přidělit k=1..j minut
                        if (a[i][k]*p[i-1][j-k] >= a[i][maxk]*p[i-1][j-maxk])
                                maxk = k;
                p[i][j] = a[i][maxk] * p[i-1][j-maxk];  // nejlepší uložíme
                m[i][j] = maxk;                         // zapamatujeme si,
        }                                // kolik jsme úkolu přidělili minut

int zbyva = M;
for (int i = N; i > 0; i--) {           // kolik minut přidelím úkolů i,
        printf("Úkol %d: %d minut\n",N-i+1,m[i][zbyva]);
        zbyva -= m[i][zbyva];           //když zbylo "zbývá" minut
}
printf("Celková pravděpodobnost spokojenosti: %f\n",p[N][M]);
return 0;
}
```

20-1-3 Oprava lodi — Josef Pihera

V naprosté většině případů měl Vilda naději, že nebude spolu s lodí muset prozkoumávat dno, což mu udělalo velkou radost. Proto si všechno o záplatování trupu pečlivě zaznamenal:

K řešení využijeme toho, že žádné dvě záplaty se nebudou překrývat, pokud mají nejmenší možnou velikost. Rovněž víme, že záplata musí pokrývat celou souvislou oblast všech děr. Nemusíme tedy příliš hloubat, abychom objevili, že rozměr i poloha záplaty jsou dány nejlevější, nejhořejší, nejpravější a nejspodnější dírou celé souvislé oblasti, ty v tomto pořadí přesně určují levý, horní, pravý a dolní okraj záplaty. Cokoliv menšího je málo, cokoliv většího by bylo zbytečné. Ke spočtení velikosti záplat nám tedy stačí projít všechny oblasti a určit jejich horní, dolní, levé a pravé okraje.

Teď se musíme vypořádat s tím, jak hledat souvislé oblasti děr. Začněme tím, že všechny díry jsou v dvojrozměrném poli *diry*. Prvek $diry[x, y]$ je logická hodnota odpovídající tomu, zda na souřadnicích $[x, y]$ díra je nebo není.

Ještě než začneme se samotným hledáním, připomeneme si jednu datovou strukturu – *frontu*. Fronta není nic jiného než nějaký seznam prvků, avšak dvě operace jsou pro ni specifické – přidání prvku na konec a odebrání prvku ze začátku. Prvky jsou tedy z fronty odebírány ve stejném pořadí, v jakém do ní byly přidány a programátorova fronta se od fronty v obchodě liší pouze tím, že se v ní nesmí předbíhat.

Nyní zpět k původnímu problému. Souvislé oblasti budeme hledat *hledáním do šířky*. Těm z vás, kteří o tomto algoritmu slyší poprvé, vřele doporučuji se s ním blíže obeznámit (je popsán v aktuální kuchařce). Na počátku si vybereme nějakou libovolnou dosud nezazáplatovanou díru a uložíme si ji do fronty. Nyní budeme postupně odebírat díry z fronty a pro každou z nich se podíváme, zda nemá vedle sebe sousedku. Sousedku hledáme prostě tak, že se v poli s dírami podíváme na políčka kolem sebe. Pokud takovou díru nalezneme, přidáme si ji na konec fronty a v poli děr si ji poznačíme jako zpracovanou, abychom ji nevkládali do fronty vícekrát. Povšimněme si, že záplatu můžeme určovat už během procházení frontou. Na počátku řekneme, že záplata zakrývá právě jednu díru, a to tu první, kterou jsme nalezli a přidali do fronty. S každou další dírou nám stačí záplatu „roztáhnout“, pokud z ní díra vybočuje. Při dosažení konce fronty jsme určitě nalezli všechny díry v dané oblasti a velikost záplaty pro tuto oblast přičteme k celkovému výsledku. Zbývá už jen po nalezení všech oblastí vypsat celkový součet.

Náš algoritmus potřeboval pole popisující celkem $M \cdot N$ políček, tedy paměťová složitost je $O(M \cdot N)$. Každé políčko zpracujeme pouze konstantně-krát, navíc přinejmenším na počátku muselo být inicializováno, takže s každým jsme pracovali alespoň jednou. Tedy časová složitost je $O(M \cdot N)$. Povšimněte si, že $D \leq M \cdot N$, proto ho můžeme „schovat“ do $O(M \cdot N)$.

Ještě se zamyslíme, jestli neexistuje vylepšení algoritmu pro D, která jsou mnohem menší než $M \cdot N$. Pokusme se zbavit závislosti na M a N, která mohou být neúměrně větší než D i bez toho, že by se mezi sebou vynásobila. Odstraňme proto ono dvojrozměrné pole pro díry a ponechme si jen posloupnost děr takovou, jakou jsme ji načetli ze vstupu. Pole, kterého jsme se zbavili, jsme používali k tomu, abychom uměli rychle naleznout sousední díry v průběhu hledání do šířky. Jak ale teď nalezneme sousední díry? Abychom to uměli rychle, uspořádáme si na počátku všechny díry podle souřadnice X a pokud se v ní shodují, tak podle Y. Nyní řekněme, že máme díru se souřadnicemi $[x, y]$. Pak její sousedky mohou být pouze $[x-1, y]$, $[x+1, y]$, $[x, y-1]$ nebo $[x, y+1]$. Jsou tedy celkem čtyři možnosti, které musíme vyzkoušet. Tedy hledáme čtyři hodnoty v uspořádané posloupnosti, což umíme půlením intervalu v logaritmickém počtu kroků vzhledem k délce takové posloupnosti. Pokud taková díra existuje, pak jsme k ní zřejmě připojeni a to už algoritmu stačí.

Celý algoritmus potřebuje pouze množinu všech děr, kterých je D, a frontu pro průchod do šířky, která může rovněž obsahovat až D prvků. Paměťová složitost je tedy $O(D)$. Časovou složitost nejvíce ovlivňuje setřídění děr a prohledávání souvislých oblastí, kde pro každý prvek provádíme půlení intervalu v logaritmickém čase. Obě hlavní fáze a tedy celý algoritmus běží v čase $O(D \cdot \log D)$. Mezi zdrojovými kódy vzorových řešení naleznete pouze tuto druhou variantu algoritmu, která je ve skutečnosti poměrně jednoduchou úpravou varianty první.

Ještě existuje jedno řešení, kterého si správně všiml Peter Ondrúška. Úlohu lze řešit v čase $O(M + N + D)$. Zvídavější z vás si nejspíše uvědomili, že prohledáváme graf a hledáme slabě souvislé komponenty. To samo o sobě umíme rychle, ale v tomto případě nám nejdéle trvá konstrukce grafu. Na počátku si přihrádkově setřídíme všechny díry podle souřadnice X, tedy je rozdělíme do sloupečků (zvládneme v čase $O(N + D)$). Pomocí tohoto rozdělení dokážeme později lépe určit vodorovně sousedící díry. Začneme postupně zpracovávat všechny sloupečky x a $x + 1$ pro x od 1 do $M - 1$. Díry těchto dvou sloupečků si vždy setřídíme podle řádků - opět přihrádkově a oba sloupečky zvlášť. Zde musíme dát pozor, abychom netřídili pro každou dvojici sloupečků v čase $O(M)$. Budeme opakovaně používat dvě přihrádková pole velikosti M (tedy co řádek, to přihrádka). Před prvním tříděním podle řádků si je musíme vynulovat ($O(M)$). V dalších krocích (pro další dvojice sloupečků) předpokládáme, že pole jsou vynulována, a my proto děláme zápisy jen pro každou díru, nikoliv pro každou přihrádku – to je zřejmé, protože používáme jen ty přihrádky, které potřebujeme. Až nebudeme obsah polí potřebovat, opět vymažeme pouze ty díry, které jsme přidali, tedy nebudeme nulovat všechny přihrádky v poli – v důsledku třídění pro všechny dvojice sloupečků a všechny díry na plánu stihneme v $O(M + D)$.

Mějme tedy přihrádkově setříděné dva sousední sloupečky děr podle řádků a seznam děr pro každý z obou sloupečků (seznamy vznikly při prvním setřídění děr podle souřadnice X). Nyní se stačí pro každou z děr v seznamech podívat do přihrádek, jestli má sousedku nad sebou nebo pod sebou (pro $x \geq 2$ to stačí pro díry ze sloupce $x+1$, protože sloupec x jsme vyřešili v předchozím kroku). Takto jsme doplnili svislé hrany do grafu (hrany odpovídající sousedství dvou děr ve svislém směru). Dále budeme hledat sousedící díry z právě zpracovávaných dvou sloupečků ve vodorovném směru. Pro každou díru ze sloupce x na řádku y (a tedy v přihrádce y pro tento sloupec) se podíváme, zda je nějaká díra v přihrádce y sloupečku $x+1$. Tím, že projdeme všechny sousední dvojice sloupečků, máme zajištěno, že nalezneme všechny vodorovné hrany našeho grafu. Pro každou díru si budeme v celku pamatovat max. 4 odkazy na sousedy (paměť $O(D)$).

Všimněte si, že se tento postup opět podobá prvnímu, ale s tím rozdílem, že nemáme „napřihrádkovaný“ úplně celý trup, ale jen tu část, kterou právě potřebujeme, a zbytek – ostatní sloupečky – jen částečně – tj. neřešíme jejich rozmístění děr na řádcích. Celková paměťová složitost odpovídá časové, tj. $O(M+N+D)$

```
program Zaplaty;
type  TDira = record x,y: LongInt; patch: boolean; end;
const MaxD=1000000;
      movex:array[1..4] of integer=(-1,1,0,0); {doleva, doprava, ..., ...}
      movey:array[1..4] of integer=(0,0,-1,1); { ..., ..., nahoru, dolů}
var diry: array[1..MaxD] of TDira;
    m,n,d: LongInt; {rozměry m n, počet děr}
    i,j,k:LongInt;
    vysledek: LongInt;
    f:array[1..MaxD] of LongInt; {fronta děr}
    fi,fcnt:LongInt; {index ve frontě, počet děr ve frontě}
    minx,miny,maxx,maxy:LongInt;

procedure ReadInp;
var x,y:LongInt;
begin
  readln(m,n,d);
  for i:=1 to d do
  begin
    readln(y, x);
    diry[i].x:= x;   diry[i].y:= y;   diry[i].patch:= false;
  end;
end;

function Compare(a,b:TDira):LongInt;
begin
  if a.x>b.x then Compare:=1 else
   if a.x<b.x then Compare:=-1 else
```

```
    if a.y>b.y then Compare:=1 else
     if a.y<b.y then Compare:=-1 else
      Compare:=0;
end;

procedure sort_diry;
 procedure qsort(l,p:LongInt);
 var i,j,m:LongInt;
     t:TDira;
 begin
   i:=l-1;
   m:=(l+p) div 2;
   t:=diry[m]; diry[m]:=diry[p]; diry[p]:=t;
   for j:=l to p-1 do
    if Compare(diry[j],diry[p])<0 then
     begin inc(i); t:=diry[i]; diry[i]:=diry[j]; diry[j]:=t; end;
   t:=diry[p]; diry[p]:=diry[i+1]; diry[i+1]:=t;
   if i>l then qsort(l,i);
   if i+2<p then qsort(i+2,p);
 end;
begin
  qsort(1,d);
end;

function find(x,y:LongInt):LongInt;
var pt:TDira;
    l,p,m:LongInt;
begin
  pt.x:=x; pt.y:=y;
  l:=1; p:=d;
  repeat
   m:=(l+p) div 2;
   if Compare(pt,diry[m])=1 then
    l:=m+1 else p:=m-1;
  until (Compare(pt,diry[m])=0) or (l>p);
  if Compare(pt,diry[m])=0 then find:=m else find:=-1;
end;

begin
  ReadInp;
  sort_diry;                          {Roztřídíme díry}
  vysledek:=0;
  for i:=1 to d do                    {Nahlédneme do každé díry a zalepíme ji}
   if not diry[i].patch then begin    {díra není zazáplatována?}
     diry[i].patch:=true;
     minx:=diry[i].x; miny:=diry[i].y; maxx:=minx; maxy:=miny;
     fcnt:=1;                         {Díru dáme do fronty}
     f[1]:=i; fi:=1;
     while fi<=fcnt do begin          {Projdeme díry ve frontě}
        for k:=1 to 4 do begin        {Zkontrolujeme, jestli nemá díra sousedky}
          j:=find(diry[f[fi]].x+movex[k],diry[f[fi]].y+movey[k]);
```

```
          if j<>-1 then                          {našli jsme nějakou vedle sebe?}
           if not diry[j].patch then begin {přidej do fronty a rozšiř záplatu}
              inc(fcnt);
              f[fcnt]:=j;
              diry[j].patch:=true;
              if minx>diry[j].x then minx:=diry[j].x;
              if maxx<diry[j].x then maxx:=diry[j].x;
              if miny>diry[j].y then miny:=diry[j].y;
              if maxy<diry[j].y then maxy:=diry[j].y;
            end;
        end;
        inc(fi);
      end;
     vysledek:=vysledek+(maxx-minx+1)*(maxy-miny+1);
   end;
  writeln(vysledek);                                  {A je to}
end.
```

20-1-4 Kormidlo — Michal „vorner“ Vaner

Zdá se, že tato úloha byla těžší, než se z počátku zdálo. Správných řešení přišlo pomálu, ty rychlé v podstatě žádné, takže Vildovi nezbylo než přibýt místo kormidla jeden obdélníkový kus dřeva, který mu zbyl z opravy.

Úkolem je vlastně spočítat počet koster na daném grafu. Co je to kostra a graf se dočtete kupříkladu v kuchařce ke třetí sérii, kterou jste právě dostali.

Vzorec na výpočet počtu koster na úplném grafu nám nepomůže, protože kormidlo není úplný graf. Stejně tak postupy pro obecné grafy jsou trochu jako nukleární bomba na vrabce. Jde to jednodušeji.

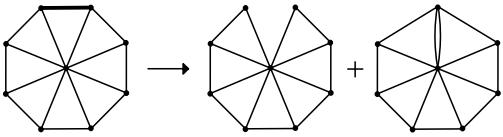
Tedy, naší úlohou je najít počet koster určeného grafu. Úlohu si mírně zobecníme. V grafu je obvykle zakázané mít násobné hrany (více hran spojující stejnou dvojici vrcholů). My toto zakazovat nebudeme, čímž dostaneme multigraf. K čemu nám to bude dobré si povíme později.

Máme tedy multigraf M. Vyberme si jednu multihranu (multihrana jsou všechny „normální“ hrany, které spojují stejné 2 vrcholy). Rozdělíme si množinu koster grafu M podle této multihraně na dvě (disjunktní) podmnožiny.

První podmnožina bude obsahovat všechny kostry, které neobsahují žádnou hranu z této multihrany. Velikost takové množiny je zjevně stejná, jako velikost množiny všech koster grafu M^-, který vznikne z M odebráním celé této multihrany.

Druhá podmnožina je ten zbytek, tedy všechny kostry, kde použijeme právě jednu hranu z této multihrany (více jich vést nemůže, to by nebyla kostra). Kdyby naše multihrana nebyla násobná (byla by to jen obyčejná hrana), velikost této podmnožiny by byla stejná jako počet koster na grafu $M^{\Rightarrow\Leftarrow}$, který

z M vznikne odstraněním této multihrany a sloučením vrcholů touto multihranou spojených do jednoho (toto je proč celou dobu pracujeme s multigrafy – tady mohou vznikat multihrany).



Jak to ale bude vypadat, když námi vybraná hrana bude h-násobná? Úplně stejně, jako s jednoduchou, jen použitou hranu můžeme vybrat h způsoby, tedy výsledkem bude $h \cdot M^{\Rightarrow\Leftarrow}$.

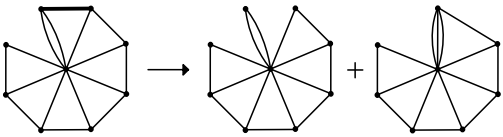
Protože jsou tyto dvě podmnožiny disjunktní a dohromady dávají celou množinu koster (nic jiného, než že tam hrana je a že tam není, se stát nemůže), můžeme velikosti těchto dvou podmnožin jednoduše sečíst.

Tímto převedeme problém počtu koster na multigrafu na dva stejné problémy, ale na menších multigrafech (čímž jsme mimochodem dokázali, že algoritmus je konečný, neboť počet koster na jednovrcholovém grafu je roven jedné a počet koster na nesouvislém grafu je nula). Nyní stačí už jen využít toho, že vstupní graf není jen tak ledajaký, ale že je to naše pěkné kormidlo.

Podívejme se, na co se rozloží kormidlo velikosti N. Vybereme si jednu hranu na jeho obvodu. Když hranu vynecháme, vznikne něco, co by se dalo nazvat vějířem (viz obrázek). Když hranu použijeme, vznikne skoro totéž, jako kormidlo velikosti $N-1$, jen s tím rozdílem, že jedna hrana do středu je dvojitá.

Kormidlo velikosti N s jednou k-násobnou hranou se rozloží na vějíř velikosti N s jednou $(k+1)$-násobnou hranou na kraji (vybereme si opět hranu sousedící s onou k-násobnou hranou) a jedno kormidlo velikosti $N-1$ s jednou $(k+1)$-násobnou hranou.

Co uděláme s vějířem velikosti N a k-násobnou krajní hranou? Vybereme si vnější hranu, která sousedí s tou k-násobnou. Když ji použijeme, dostaneme vějíř velikosti $N-1$ s jednou $k+1$-násobnou hranou. Když ji nepoužijeme, dostaneme vějíř velikosti $N-1$ na násobné stopce. Protože do toho vrcholu na konci stopky vede už jen tato multihrana, musíme ji použít a počet koster takové kostry bude stejný jako počet koster vějíře velikosti N vynásobeným k (máme k způsobů, jak připojit stopkový vrchol).



Nyní, kdy toho necháme? Vějíře se jednou stáhnou až do jedné k-násobné hrany (na které najdeme k různých koster). Když nám nebude vadit myšlenka

existence kormidla velikosti 1 s jednou k-násobnou hranou, všimneme si, že je to opět hrana samotná (spojující „krajní“ se „středovým“ vrcholem).

Nyní trocha počtů. Označme μ_N^k počet koster korμdla o velikosti N s jednou k-násobnou hranou. Stejně tak V_N^k budiž počet koster vějíře velikosti N s jednou k-násobnou hranou. Pomocí našeho rozkládacího pravidla si vyjádříme, že $V_N^k = V_{N-1}^{k+1} + k \cdot V_{N-1}^1$. Stejně tak $\mu_N^k = V_N^k + \mu_{N-1}^{k+1}$. Toto je jen přepis výše zmíněných rozkladů na menší podproblémy.

Kdybychom nyní iterovali přes všechna potřebná N a k (všimněme si, že k bude nejvýše N – až na nějaké malé konstanty okolo), tak se zajisté dobereme k výsledku. Když si budeme mezivýsledky ukládat (některé budeme potřebovat vícekrát), tak se dostaneme na časovou složitost $O(N^2)$.

Mohlo by se stát, že se nám taková časová složitost nelíbí. V takovém případě se pokusíme zbavit počítání multigrafů s násobnými hranami tím, že přepíšeme vzorečky, aby používaly pouze V_N^1 a μ_N^1. Postupně budeme rozkládat vše, co má horní index různý od 1. Tedy, $V_N^k = k \cdot V_{N-1}^1 + V_{N-1}^{k+1} = k \cdot V_{N-1}^1 + (k+1) \cdot V_{N-2}^1 + V_{N-3}^{k+2} = \ldots$ Zastavíme se, až budeme mít V_1^{k+N-1}, což je, jak jsme si rozmysleli výše, $k+N-1$. Obdobně to uděláme pro μ_N^k. Doporučuji si to napřed rozepsat pro třeba $N = 4$, je z toho hezky vidět, co vyjde.

Protože již k nepotřebujeme, pro zkrácení si označme V_N jako ekvivalent V_N^1. Obdobně pro μ_N a μ_N^k. Až práci s tužkou a papírem dokončíme, vyjde nám, že $V_N = 1 \cdot V_{N-1} + 2 \cdot V_{N-2} + \ldots + (N-1) \cdot V_1 + N$. Pro celá kormidla to vyjde $\mu_N = 1^2 \cdot V_{N-1} + 2^2 \cdot V_{N-2} + \ldots + (N-1)^2 \cdot V_1 + N^2$.

Kdybychom nám někdo dal všechny $V_1, \ldots, V_{N-1}$, není problém v lineárním čase spočítat μ_N sečtením všech sčítanců.

Zbývá tedy spočítat všechny vějíře, pokud možno také v lineárním čase. Kdybychom měli čísla $S_l = 1 + \sum_{i=1}^{l} V_i$ a $V_l = l + \sum_{i=1}^{l-1}(l-i) \cdot V_i$, jejich sečtením získáme V_{l+1} (čtenář si může ověřit sečtením). S_{l+1} získáme tak, že k S_l přičteme V_{l+1} (které již nyní máme také). Stačí doplnit startovní hodnoty. V_1 je jedna (vějířek s jedním krajním bodem je jen hrana), S_1 spočteme na 2. Všechny tedy zvládneme spočítat v $O(N)$.

Nyní si už stačí jen všimnout, že každé V_l potřebujeme jen k přičtení k celkovému výsledku (samozřejmě vynásobené správným číslem). Toto přičtení můžeme udělat okamžitě, tudíž ho již příště nepotřebujeme a není třeba uchovávat pole se všemi. Tím k lineární časové složitosti získáme jako bonus konstantní paměťovou.

Program si můžeme zjednodušit dopočítáním V_0 a S_0 (na 0 a 1), čímž zjednodušíme chování cyklu a celkový součet můžeme přepočítat už po spočtení V_1.

```
program kormidlo;

var N, V, S, Total, i: longint;
```

```
begin
        writeLn( 'Jak velké je kormidlo?' );
        readLn( N );
        V := 0;
        S := 1;
        Total := N*N;
        for i := 1 to N - 1 do begin
                inc( V, S );
                inc( S, V );
                inc( Total, ( N - i ) * ( N - i ) * V );
        end;
        writeLn( 'Kormidlo je možné sestavit ', Total, ' způsoby' );
end.
```

Poznámka M.M.: Každý pravověrný matematik samozřejmě věří, že na libovolný „počítací“ problém existuje chytrý vzoreček. Někdy je i hezký :) Pokud na formulky pro μ_N z našeho vzorového řešení použijete techniku zvanou metoda vytvořujících funkcí (ta je moc pěkně popsaná ve starých dobrých Kapitolách z diskrétní matematiky), dostanete následující pěkný vztah (časem – ono dá docela dost práce se tím vším propočítat, takže detaily si pro tentokrát odpustíme):

$$\mu_N = \alpha^N + \beta^N - 2,$$

kde α a β jsou konstanty definované takto:

$$\alpha = \frac{3+\sqrt{5}}{2}, \quad \beta = \frac{3-\sqrt{5}}{2}.$$

Pro počítání v programu to žádná velká výhra není, protože stěží dovedeme iracionální odmocniny z pěti reprezentovat dost přesně. Můžeme si ale pomoci drobným úskokem: Podobně jako se počítá s komplexními čísly jako s výrazy typu $a + b\sqrt{-1}$, my budeme počítat s dvousložkovými čísly ve tvaru $a + b\sqrt{5}$, kde a a b jsou racionální. Jelikož součet, rozdíl i součin takových čísel je opět číslo v tomto tvaru, můžeme vše počítat v nich a na konci pouze vypsat první složku. (Víme totiž, že výsledek je přirozené číslo, a tak musí být druhá složka nulová. Navíc díky symetrii bude první složka u α^N stejné jako u β^N, takže stačí počítat jen jednu z nich). Ještě si vzpomeneme na trik na rychlé umocňování (viz třeba řešení úlohy 18-4-1) a vyloupne se následující program, který μ_N spočítá v čase $O(\log N)$.

```
/* Dvojsložková čísla a jejich násobení */
typedef struct { int i, j; } num;
num mul(num x, num y)
{ return (num){ x.i*y.i + 5*x.j*y.j,
                x.i*y.j + x.j*y.i }; }
```

```
int M(int n)
{
  num x={3,1}, y={1,0};    // x=2*alfa
  for (int i=n; i; i/=2)   // počítáme y=x^n
    {
      if (i%2)
       y = mul(y,x);
      x = mul(x,x);
    }
  return ((2*y.i) >> n) - 2;
}
```

20-1-5 Praktická – Studentův rozvrh — Martin „Bobřík“ Kruliš

K řešení tohoto problému použijeme jednoduchý *„hladový“* (*„greedy“*) algoritmus. Na začátku si nejprve všechny přednášky setřídíme vzestupně podle času konce f_i (pozor, pokud je setřídíme jinak - např. podle času začátku s_i, tak to nebude fungovat). V průběhu algoritmu si budeme navíc držet čas konce poslední dosud vybrané přednášky (označíme ho F a na počátku bude nastaven na hodnotu mínus nekonečno).

Samotný algoritmus je vlastně pouze jedním cyklem přes setříděné přednášky, při kterém si hladově vybereme, které přednášky vezmeme, a které ne. Na každou přednášku i se podíváme a porovnáme její začátek s číslem F. Pokud je $F \leq s_i$, pak si přednášku vybereme a aktualizujeme hodnotu $F = f_i$. V opačném případě víme, že se přednáška nevejde do rozvrhu a tak se s ní nemusíme dále zdržovat.

Jak vidíte, algoritmus je v zásadě lehký. Zbývá ukázat, že také funguje. Množina přednášek, kterou náš program vydá, je určitě nezávislá (žádné přednášky se v ní nepřekrývají). To je vidět na první pohled přímo z definice hladového výběru. Ovšem není úplně jisté, jestli je tato množina také maximální možná (tzn. jestli neexistuje jiná nezávislá množina, ve které by bylo víc přednášek).

Abychom měli jistotu, že je naše množina také maximální, musíme vědět, že nám výběr některé přednášky nezablokuje lepší řešení. Řekněme, že jsme právě vybrali do našeho rozvrhu nějakou přednášku a chceme ukázat, že tato přednáška nezablokuje výběr dvou (nebo více) přednášek, které bychom mohli zapsat místo ní.

Budeme postupovat sporem. Nechť jsme vybrali přednášku x, která nám zablokovala výběr přednášek y a z (pro nástin důkazu nám dvě stačí, ale počet by šel libovolně rozšířit). Protože přednášky vybíráme setříděné podle času konce, musí rozhodně platit: $f_x \leq f_y$ a $f_x \leq f_z$. Navíc přednáška x blokuje přednášky y a z, což lze zapsat jako $s_y \leq f_x$ a $s_z \leq f_x$. Nemusíme mít doktorát z matematiky, abychom si všimli, že přednášky y a z spolu musí také kolidovat

(minimálně prochází obě časovým bodem f_x). To je ovšem spor s předpoklady, protože přednášky y a z nelze obě zařadit do rozvrhu místo x.

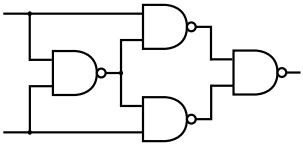
Tím jsme ukázali, že rozvrh spočítaný naším algoritmem bude nejen korektní z hlediska nezávislosti, ale také největší možný.

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

struct SCourse {        /* Struktura udržující informace o jedné přednášce. */
        int idx;        // index přednášky
        int start;      // čas začátku
        int finish;     // čas konce
};

/* Funkce načítající přednášky ze souboru do pole courses. */
int loadCourses(struct SCourse **courses) {
        int count;
        FILE *fp = fopen("prednasky.in", "r");
        fscanf(fp, "%d\n", &count);
        *courses = (struct SCourse*)malloc(count * sizeof(struct SCourse));
        for(int i = 0; i < count; i++) {
                fscanf(fp, "%d %d\n", &((*courses)[i].start),
                                              &((*courses)[i].finish));
                (*courses)[i].idx = i+1;
        }
        fclose(fp);
        return count;
}

/* Funkce porovnávající přednášky dle času konce. */
int courseCompare(const void *course1, const void *course2) {
return ((struct SCourse*)course1)->finish - ((struct SCourse*)course2)->finish;
}

/* Implementace hladového algoritmu nad přednáškami.
 * Vybrané přednášky rovnou ukládá do výstupního souboru. */
void greedy(struct SCourse *courses, int count) {
        if (count < 1) return;
        int i = 0, resCount = 0, last = -1;
        while(i < count) {
                if (i != resCount) courses[resCount] = courses[i];
                last = courses[resCount].finish;
                resCount++;  i++;
                // hladově vybereme další přednášku
                while((i < count) && (courses[i].start <= last)) i++;
        }
        FILE *fp = fopen("rozvrh.out", "w");
        fprintf(fp, "%d\n", resCount);
        for(i = 0; i < resCount; i++) fprintf(fp, "%d ", courses[i].idx);
        fclose(fp);
}
```

```
int main() {
        struct SCourse *courses;
        int count;
        count = loadCourses(&courses);
        qsort(courses, count, sizeof(struct SCourse), courseCompare);
        greedy(courses, count);
        return 0;
}
```

20-1-6 Hrady, hrádky, hradla — Cyril Hrubiš

Prvním úkolem bylo vymyslet hradlo XOR z hradel NAND a nakreslit takový obvod. Asi nejjednodušší je tento obvod:



Úkolem druhým bylo najít všechny dvouvstupové funkce. Je několik způsobu, jak si spočítat, kolik jich vlastně je. Vstupem funkce jsou čtyři různé dvojice (00, 10, 01, 11). Máme dvě možnosti jako odpovědět na 00, dvě jak odpovědět na 10 ..., celkem nám vychází $2 \cdot 2 \cdot 2 \cdot 2 = 2^4$ možných funkcí. K tomuto číslu se dá také dobrat jinou úvahou: Budeme sčítat počty čtyřprvkových posloupností složených z nul a jedniček a to tak, že si je roztřídíme do skupin podle toho, kolik obsahují jedniček. Zavedeme k od 0 do 4, které označuje počet jedniček v posloupnosti. Pro $k = 0$ je to právě jedna možnost, a to čtyři nuly. Pro $k = 1$ jsou to 4 možnosti: představme si, že jednička postupně prochází všechny pozice. Pro $k = 2$ je to 6 možností, přijít se na to dá třeba takhle: pokud je levá z obou jedniček na první pozici, jsou 3 možnosti, kam dát druhou, pokud na druhé pozici, tak 2, a pro třetí pozici jen jedna. Pro $k = 3$ a $k = 4$ je situace stejná jako pro $k = 1$ a $k = 0$, stačí zaměnit jedničky a nuly. Tedy celkem $1 + 4 + 6 + 4 + 1$. Což je překvapivě součet pátého řádku v Pascalově trojúhelníku.

Dle obou výpočtů nám vyšlo, že máme celkem 16 dvouvstupových funkcí. Najdete je v následující tabulce, kde jsme si je roztřídili do několika skupin: nejprve dvě funkce konstantní, pak čtyři s jednou jedničkou, čtyři s jednou nulou a konečně šest funkcí se dvěma jedničkami a dvěma nulami:

X	Y	'0'	'1'	AND	$>$	$<$	NOR	OR	$\geq$	$\leq$	NAND
1	1	0	1	1	0	0	0	1	1	1	0
1	0	0	1	0	1	0	0	1	1	0	1
0	1	0	1	0	0	1	0	1	0	1	1
0	0	0	1	0	0	0	1	0	1	1	1

X	Y	X	Y	XNOR	XOR	$\neg Y$	$\neg X$
1	1	1	1	1	0	0	0
1	0	1	0	0	1	1	0
0	1	0	1	0	1	0	1
0	0	0	0	1	0	1	1

Úkolem třetím bylo dokázat, že všechny dvouvstupové funkce jdou postavit z hradel NAND.

Nejprve si všimneme, že všechny funkce postavíme z hradel AND, OR a NOT. Konstantní nula je $x \text{ AND } \neg x$. Funkce obsahující právě jednu jedničku jsou (v pořadí podle naši tabulky) $x \text{ AND } y$, $x \text{ AND } \neg y$, $\neg x \text{ AND } y$ a $\neg x \text{ AND } \neg y$. Funkce s více jedničkami dostaneme jako OR funkcí pro jednotlivé jedničky. Například $x \text{ XOR } y = (x \text{ AND } \neg y) \text{ OR} (\neg x \text{ AND } y)$.

Teď už stačí dokázat, že z hradel NAND postavíme AND, OR a NOT. To je snadné: $\neg x = x \text{ NAND } x$, $x \text{ AND } y = \neg(x \text{ NAND } y)$ a $x \text{ OR } y = (\neg x) \text{ NAND} (\neg y)$.

Totéž by se dalo dokázat i jinak: Všimneme si, že každá funkce má svůj negovaný protějšek, takže stačí z hradel NAND postavit NOT a 8 vhodně vybraných funkcí, s nimiž rovnou dostaneme i jejich negace. Hradlo NOT opět složíme jako $x \text{ NAND } x$, NAND už máme (a získáme AND), 0 je $x \text{ NAND } x$ (z toho 1), OR jest $\neg x \text{ NAND } \neg y$ (z toho NOR), XOR jest $(x \text{ OR } y) \text{ AND} (x \text{ NAND } y)$ (z toho XNOR), $>$ jest $x \text{ AND } \neg y$ (z toho $\leq$) a $<$ jest $y \text{ AND } \neg x$ (a máme i $\geq$). Tím je důkaz hotov.

Mimochodem, všechny dvouvstupové funkce by také šly vyrobit jen z hradel NOR. Vskutku: $\neg x = x \text{ NOR } x$, OR získáme jako $\neg(x \text{ NOR } y)$ a AND coby $(\neg x) \text{ NOR} (\neg y)$. Z těchto funkcí už umíme získat všechny zbylé.


Kdybychom počítali n-vstupové funkce, měla by naše tabulka 2^n řádků, takže by existovalo 2^{2^n} způsobů, jak ji vyplnit. Všimněte si, že trik vyjádřením všech funkcí pomocí AND, OR a NOT by stále fungoval, takže by stále stačil samotný NAND nebo NOR, jen by obvody byly trochu složitější.


Zkuste si dokázat, že hradla NAND a NOR jsou jediná takto univerzální. Třeba AND takový není proto, že každý obvod, ve kterém jsou jenom ANDy, na vstup ze samých nul odpoví zase nulou, takže nelze postavit (třeba) negaci.

20-2-1 Volba šamana — Petr Kratochvíl

Nejjednodušší způsob, jak najít šamana, je nejspíš tento: V prvním mezičase mezi údery bubnu každý elf vytvoří zprávu obsahující jeho jméno a věk a sdělí ji svému sousedovi vlevo. V dalších mezičasech elfové pouze přeposílají poslední přijaté zprávy, a to tak dlouho, než jim přijde jejich původní zpráva (poznají ji podle svého jména). To se stane u všech elfů najednou, protože každá zpráva obejde celé kolo elfů za stejnou dobu. Takže skončí všichni současně. Když si každý elf bude ještě pamatovat zprávu nejstaršího dosud známého elfa,

bude na konci rituálu znát jméno a věk šamana, protože každá zpráva (včetně té od budoucího šamana) obešla všechny elfy.

Tímto postupem bude celý rituál s N elfy trvat N úderů bubnu. Jak si mnozí řešitelé všimli, je možné rituál dvakrát zrychlit, když budou elfové svoje zprávy posílat oběma směry. V takovém případě stačí, když zpráva obejde půl kola, a rituál bude trvat jenom $N/2$ úderů. Zbývá vyřešit ukončení rituálu. Pokud je počet elfů v kruhu sudý, pozná se konec jednoduše tak, že ke každému elfovi dojde z obou stran stejná zpráva. Pokud je elfů lichý počet, sejdou se obě kopie zprávy jakoby na půl cesty mezi dvěma elfy. Elf v jednom kroku dostane stejnou zprávu, jakou právě odeslal. Elfové sice neví, jestli jich je sudý nebo lichý počet, ale skončí jednoduše tehdy, když nastane jeden z těchto případů.

20-2-2 Setříděné stromy — Pavel Čížek

Temný mág by z Vás měl radost. Většina došlých řešení potřebovala $O(\log K)$ porovnání. Ukážeme si zde jedno, které je sice trochu delší na rozbor, ale porovnává stromy opravdu nejméně a dokážeme si to o něm. Nejdříve tři drobná pozorování.

Pokud je délka první resp. druhé řady stromů N_1, resp. N_2 větší než K, můžeme prvky, které jsou v ní více než K-té, zahodit, jelikož určitě nebudou K-té celkově.

Dále K-tý nejvyšší strom existuje právě tehdy když $N_1 + N_2 \geq K$. Pokud tedy $N_1 + N_2 < K$ vypíšeme, že strom neexistuje a jsme hotovi. Zřejmě jsme tohle byli schopni udělat bez porovnávání výšek stromů a tedy lépe to nešlo.

Podobně triviální případ nastane, když jedna z posloupností bude mít nulovou délku. Pak K-tý strom celkově bude samozřejmě K-tý v řadě stromů s nenulovou délkou. Tím jsme odstranili speciální případy a dále budeme pokračovat s posloupnostmi, pro jejichž délky platí $1 \leq N_1 \leq K$, $1 \leq N_2 \leq K$ a $N1 + N2 \geq K$.

Nyní se dostáváme k vlastnímu algoritmu. Ten je založen na metodě půlení intervalů. Bohužel, pokud chceme mít opravdu optimální řešení, tak se nám toto půlení intervalů rozpadne na 3 případy, které sice budou skoro stejné na naprogramování, nicméně, zvláště u třetího, se budou lišit interpretací toho, co které proměnná znamená. Pokud Vás tedy bude zajímat jen idea programu, stačí si přečíst jen první z nich.

Nejdříve případ, kdy $N1 = N2 = K$. Tady si budeme udržovat v paměti o kterých stromech Z_1, resp. Z_2 z první, resp. druhé posloupnosti budeme vědět, že jsou celkově méně než K-té v pořadí. Dále budeme mít v paměti délku intervalu D, kde ještě K-tý nejvyšší strom může být (budou v úvahu připadat stromy $Z_1+1, Z_1+2, \ldots, Z_1+D$ z první, resp. $Z_2+1, Z_2+2, \ldots, Z_2+D$ z druhé řady). Na D je také možno se dívat jako na počet stromů, které jsou celkově nejvýše K-té, ale které jsme zatím ještě nenašli (rozumíme které jsou více než

Z_1., resp. Z_2. v první, resp. druhé řadě). Z tohoto úhlu pohledu je zřejmá interpretace součtu $Z_1 + Z_2 + D$, který bude po celou dobu běhu programu roven K. Na počátku zřejmě $Z_1 = Z_2 = 0$ (a nula tady znamená, že i první strom v příslušné posloupnosti může být K-tý nejvyšší) a $D = K$.

Nyní k vlastnímu půlení intervalů. Vezměme z první, resp. druhé řady strom s pořadím $S_1 = Z_1 + \lfloor D/2 \rfloor$, resp. $S_2 = Z_2 + \lfloor D/2 \rfloor$. Tyhle dva stromy budeme muset porovnat. BÚNO nechť je vyšší první z nich. To znamená, že z druhé řady mohou být vyšší než strom S_1 jen stromy $1, 2, \ldots, S_2 - 1$. Nicméně jak víme, tak $S_1 + S_2 - 1 \leq K - 1$ a tedy strom S_1 může být celkově nejvýše $(K-1)$-ní. Tím jsme o něm dokázali, že musí být před tím, který hledáme a tak můžeme prohlásit, že $Z_1 = S_1$. Tím jsme vyloučili další stromy a tak musíme zkrátit i interval D, ve kterém hledáme, konkrétně na $\lceil D/2 \rceil$. Jak se můžeme snadno přesvědčit, bude opět platit $Z_1 + Z_2 + D = K$.

Předchozí odstavec budeme opakovat dokud $D > 1$. Pak budeme mít o $Z_1 + Z_2 = K - 1$ stromech dokázáno, že jsou nejvýše $(K - 1)$-ní a tedy K-tý strom v pořadí bude vyšší z dvojice stromů $Z_1 + 1$ a $Z_2 + 1$.

Nyní k důkazu optimality tohoto případu. X dotazy na porovnání výšek stromů můžeme rozlišit mezi 2^X možnostmi (máme jen 2 možné odpovědí - je vyšší první, resp. druhý strom). Rozhodujeme se mezi $2K$ stromy a tak potřebujeme alespoň $\lceil \log_2(2K) \rceil = \lceil (\log_2 K) \rceil + 1$ porovnání. Náš program bude $\lceil \log_2 K \rceil$-krát porovnávat během cyklu a pak bude jedno porovnání na rozhodnutí, zda je K-tý nejvyšší strom v první, resp. druhé řadě.

Druhý uvažovaný případ nastane, když $N_1 = K$ a $N_2 < K$ (nebo obráceně). Tady můžeme vyloučit na začátku stromy $1, 2, \ldots, K - N_2 - 1$ z první řady aniž bychom potřebovali porovnávat - v druhé řadě prostě není dost stromů na to, aby byly celkově K-té. Tím máme v první řadě $N_2 + 1$ stromů, které mají naději stát se K-tým nejvyšším, zatímco v druhé řadě je jich jen N_2. Zavedeme si tedy v druhé řadě virtuální $(N_2 + 1)$-ní strom, který bude nižší než jakýkoliv jiný (v programu je tohle implementováno ve funkci porovnávající stromy), čímž délky obou intervalů srovnáme a můžeme použít výše uvedené půlení intervalů, kde ale počáteční podmínky budou $Z_1 = K - N_2 - 1$, $Z_2 = 0$ a $D = N_2 + 1$.

Důkaz optimality bude analogický. Vybíráme z $2N_2 + 1$ stromů, budeme tedy potřebovat $\lceil \log_2(2N_2 + 1) \rceil$ dotazů. Program bude chtít porovnat $\lceil \log_2(N_2 + 1) \rceil$-krát během cyklu a jednou na rozhodnutí, ve které ze zadaných dvou řad hledaný strom je. Tato dvě čísla jsou shodná, ač to na první pohled není vidět. Pro naše N_2 je $\lceil \log_2(2N_2+1) \rceil = \lceil \log_2(2N_2+2) \rceil = \lceil \log_2(N_2+1) \rceil + 1$. První rovnost neplatí obecně, nicméně my víme, že N_2 je přirozené číslo. Tedy $2N_2 + 1$ je číslo liché větší než 2 proto je horní celá část binárního logaritmu stejná jako horní celá část binárního logaritmu čísla o 1 vyššího. (Kdybychom si nakreslili graf funkce $f(x) = \lceil \log_2(x) \rceil$, tak by vypadal jako konstantní funk-

ce až na body, které jsou mocninou dvojky, kde $f(x)$ zvyšuje skokově svou hodnotu o 1. Nicméně mocniny dvojky větší než dva jsou sudé.)

Zbývá poslední případ, kdy $N_1 < K$ a $N_2 < K$. Tady budeme moci i bez porovnávání tvrdit, že stromy $1, 2, \ldots, K-1-N_2$ v první, resp. $1, 2, \ldots, K-1-N_1$ v druhé řadě budou v celkovém pořadí maximálně $(K-1)$-ní a tedy je můžeme vyloučit rovnou. Analogicky k předchozímu případu bychom tedy dosadili $Z_1 = K-1-N_2$, $Z_1 = K-1-N_1$ a $D = N_1+N_2-K+2$. Tohle samozřejmě vede k řešení s logaritmickým počtem dotazů, bohužel v některých případech bude chtít měřit jednou navíc. Vybíráme totiž z $2 \cdot (N_1+N_2-K+1)$ stromů (a tedy optimální počet porovnání je $\lceil \log_2(2 \cdot (N_1+N_2-K+1)) \rceil$) a přímočaře zobecněný postup použitý v předchozích dvou případech vede na $\lceil \log_2(N_1+N_2-K+2) \rceil + 1$ porovnání.

Vylepšení na optimální počet porovnání je trochu trik. Místo toho, abychom hledali K-tý nejvyšší prvek, budeme hledat $(K+1)$-ní. Zřejmě budeme tedy mít u výše uvedeného algoritmu počáteční podmínky $Z_1 = K-N_1$, $Z_2 = K-N_2$ a $D = N_1+N_2-K+1$. Po skončení cyklu bude platit $Z_1+Z_2 = K$ víme, že $(K+1)$-ní nejvyšší strom je vyšší ze stromů Z_1+1 a Z_2+1. Co si ale můžeme dovolit tvrdit dále je, že K-tý nejvyšší je nižší ze stromů Z_1 a Z_2. Důkaz toho je jednoduchý - víme, že všechny stromy, které jsou nejvýše K-té nejvyšší jsou $1, 2, \ldots Z_1$ v první, resp. $1, 2, \ldots Z_2$ v druhé řadě. K-tý nejvyšší musí tedy i K-tý nejvyšší být mezi nimi a kvůli setřízenosti vstupních posloupností bude na konci jedné z nich.

Tenhle „trikový“ postup potřebuje $\lceil \log_2(N_1+N_2-K+1) \rceil$ dotazů během cyklu a jeden dotaz na rozhodnutí se mezi stromy Z_1 a Z_2. To je ale přesně, jak již bylo uvedeno, minimální počet dotazů potřebných k určení K-tého nejvyššího stromu.

```
const MaxN = 10000;

var Posloupnost1:array[1..MaxN] of real;
    Posloupnost2:array[1..MaxN] of real;
    DelkaPosloupnosti1:integer;
    DelkaPosloupnosti2:integer;
    K:integer;
    Zacatek1,Zacatek2:integer;
    DelkaIntervalu:integer;
    Stred1,Stred2:integer;

procedure NactiVstup();
var index:integer;
begin
{V praktické implementaci v lese by tohle již bylo hotovo ...}
   read(DelkaPosloupnosti1);
   for index:=1 to DelkaPosloupnosti1 do
      read(Posloupnost1[index]);
```

```
   read(DelkaPosloupnosti2);
   for index:=1 to DelkaPosloupnosti2 do
      read(Posloupnost2[index]);
   read(K);
end;

function PorovnejStromy(Strom1,Strom2:integer):boolean;
{Vrací true, pokud je strom v první posloupnosti vyšší}
begin
   if (Strom1 > DelkaPosloupnosti1) then PorovnejStromy:=false
   else if (Strom2 > DelkaPosloupnosti2) then PorovnejStromy:=true
   else PorovnejStromy:=(Posloupnost1[Strom1] > Posloupnost2[Strom2]);
{Nebo jiná implementace porovnávání stromů ...}
end;

begin
   NactiVstup();
   if (DelkaPosloupnosti1 + DelkaPosloupnosti2 < K) then begin
      writeln('Takový strom neexistuje.');      {Nemáme ani K stromů ...}
   end else if DelkaPosloupnosti1 = 0 then
      writeln(K,'. nejvyšší strom je ',K,'. v druhé posloupnosti.');
   else if DelkaPosloupnosti2 = 0 then
      writeln(K,'. nejvyšší strom je ',K,'. v první posloupnosti.');
      {Tím jsme ošetřili speciální případy}
   else begin
      if DelkaPosloupnosti1 > K then DelkaPosloupnosti1:=K;
      if DelkaPosloupnosti2 > K then DelkaPosloupnosti2:=K;
      {Dál než K-tý prvek, který nás zajímá, nebude.
      A teď nás čeká nastavení počátečních podmínek}
      if DelkaPosloupnosti1 = K then begin
         if DelkaPosloupnosti2 = K then begin
            Zacatek1:=0; Zacatek2:=0;
            DelkaIntervalu:=K;
         end else begin
            Zacatek1:=K - DelkaPosloupnosti2 - 1; Zacatek2:=0;
            DelkaIntervalu:=DelkaPosloupnosti2 + 1;
         end;
      end else begin
         if DelkaPosloupnosti2 = K then begin
            Zacatek1:=0; Zacatek2:=K - DelkaPosloupnosti1 - 1;
            DelkaIntervalu:=DelkaPosloupnosti1 + 1;
         end else begin
            Zacatek1:=K - DelkaPosloupnosti2; Zacatek2:=K - DelkaPosloupnosti1;
            DelkaIntervalu:=DelkaPosloupnosti1 + DelkaPosloupnosti2 - K + 1;
         end;
      end;
      while DelkaIntervalu > 1 do begin {Nyní začne binární vyhledávání}
         Stred1:=Zacatek1 + (DelkaIntervalu div 2);
         Stred2:=Zacatek2 + (DelkaIntervalu div 2);
         if PorovnejStromy(Stred1,Stred2) then
            Zacatek1:=Stred1
```

```
         else Zacatek2:=Stred2;
         DelkaIntervalu:=(DelkaIntervalu+1) div 2;
      end;
      if (Zacatek1 + Zacatek2 = K - 1) then begin
         {Máme nalezeno K-1 stromů, které jsou nejvýše (K-1)-ní}
         if PorovnejStromy(Zacatek1+1,Zacatek2+1) then
            writeln(K,'. nejvyšší strom je ',Zacatek1+1,'. v první posl.')
         else writeln(K,'. nejvyšší strom je ',Zacatek2+1,'. v druhé posl.');
      end else begin    {Nebo K stromů, které jsou nejvýše K-té}
         if PorovnejStromy(Zacatek1,Zacatek2) then
            writeln(K,'. nejvyšší strom je ',Zacatek2,'. v druhé posl.')
         else writeln(K,'. nejvyšší strom je ',Zacatek1,'. v první posl.');
      end;
   end;
end.
```

20-2-3 Morseovkabezoddělovačů — Martin Mareš

Telegraficky: `-..-.---..---..-.-.-.--.-....` :-)

Dobrá, tak trochu podrobněji ...

První pokus: Kdybychom chtěli zjistit jen to, zda se zadaný řetězec teček a čárek dá rozložit na slova (tedy aniž bychom uvažovali nad tím, jaká možnost je nejkratší), stačilo by vyzkoušet všechna slova, která pasují na začátek řetězce, a pro každou z těchto možností si zavolat tutéž funkci rekurzivně na zbytek řetězce.

To samozřejmě funguje, jenže někdy až exponenciálně pomalu: pokud budeme mít ve slovníku slova e (`.`) a i (`..`) a vstupní řetězec bude obsahovat spoustu teček a nakonec čárku, náš algoritmus postupně vyzkouší všechny možnosti, jak tečky rozložit na e-čka a i-čka, a při každé z nich se zarazí o koncovou čárku a se svěšeným ocasem se vrátí zpět. Spočítat, kolik přesně těch možností pro n teček bude, není úplně jednoduché (mimochodem, je to $(n+1)$-ní Fibonacciho číslo), ale je jich určitě aspoň $2^{n/2}$. To proto, že když si vezmeme nějakou posloupnost $n/2$ znaků složenou libovolně z e-ček a i-ček, bude její morseovkový zápis tvořen nejvýše n tečkami. Všechny takové posloupnosti (a těch je $2^{n/2}$) náš algoritmus určitě vyzkouší a ještě i mnohé další. Nic moc.

Druhý pokus: Použijeme *dynamické programování* (trochu netradičně odzadu, brzy bude jasné, proč). Nechť vstupní řetězec obsahuje n morseovkových značek $z_1, \ldots, z_n$. My svou úlohu zkusíme vyřešit postupně pro všechny jeho konce: Budeme počítat hodnoty b_i, které budou říkat, na kolik nejméně slov se dá rozložit úsek $z_i, \ldots, z_n$, případně nějaké ∞ (tedy chci říci hrozně velké číslo), pokud se úsek rozložit nedá.

Všimneme si, že tato b_i se dají docela snadno spočítat pozpátku: Začneme trochu šalamounsky u b_{n+1} – to je vždy nula, protože prázdnou posloupnost značek můžeme určitě rozložit na prázdnou posloupnost slov. Dále b_n je buďto 1 (to pokud poslední značka odpovídá nějakému jednopísmenkovému slovu

ve slovníku) nebo ∞ (pokud ne). Pokud už máme spočítaná $b_{i+1}, \ldots, b_{n+1}$, snadno dopočítáme b_i: Každý rozklad značek od z_i do konce musí začínat nějakým slovem, řekněme délky ℓ značek, a pokračovat rozkladem značek od $z_{i+\ell}$ do konce. Stačí tedy prozkoumat všechna slova, která do vstupu pasují od i-té pozice, a pro každou z nich se podívat, jestli umíme rozložit zbytek (to nám řekne $b_{i+\ell}$). Pokud je možností víc, vybereme si samozřejmě tu s nejmenším počtem slov (a tedy nejmenším $b_{i+\ell}$). Jinými slovy (ehm, znaky ...):

$$b_i = 1 + \min\{b_{i+\ell} : z_i, \ldots, z_{i+\ell-1} \text{ je známé slovo}\}.$$

Takto se postupně dopočítáme až k b_1, což je minimální počet slov, ze kterých se dá poskládat celý vstup, a to je skoro řešení naší úlohy. Chybí jen tato slova najít. K tomu stačí, abychom si u každého (konečného) b_i ještě zapamatovali, které bylo to slovo, jež jsme na tomto místě napasovali do textu. Tomu říkejme třeba s_i a jeho délce ℓ_i.

S těmito informacemi už můžeme rekonstruovat celé řešení (pro změnu odpředu). Našli jsme rozklad s b_1 slovy a jeho první slovo je s_1. Zbytek řetězce tedy musí být rozložen na $b_{1+\ell_1}$ slov a první z nich je $s_{1+\ell_1}$. A tak dále.

Časová složitost se nahlédne snadno. Počítáme celkem n hodnot b_i, pro každou z nich zkoušíme nejvýše tolik pasujících slov, kolik je maximální délka L slova ve slovníku (mohlo by sice existovat více slov, která by pan Morse psal stejně, ale z těch nám jistě stačí vyzkoušet jedno jediné). Pokud budeme odvážně předpokládat, že jedno slovo otestujeme v konstantním čase, stojí nás to celkem čas $O(nL)$ a paměť $O(n)$ bez slovníku. Zpětný průchod pak zabere pouze čas $O(n)$ a konstantní paměť.

Jak ale reprezentovat slovník, abychom s ním dokázali pracovat tak rychle, jak jsme slíbili? Pomůže nám datová struktura zvaná *trie.* To je úplně obyčejný strom, do kterého postupně uložíme všechna známá slova v morseovce tak, že v kořeni se budeme rozhodovat podle první značky, v dalším vrcholu podle druhé značky a tak dále. V každém vrcholu si pak zapamatujeme, která slova tam končí (nezapomeňte, že jich může být víc se stejným zápisem). Každé slovo dokážeme do trie přidat v čase lineárním s jeho délkou, celou strukturu tedy vybudujeme lineárně s velikostí slovníku (čili součtem délek všech slov).

Pro libovolnou posloupnost ℓ značek pak samozřejmě umíme v čase $O(\ell)$ zjistit, jestli ji máme ve slovníku, ale my jsme přeci slíbili $O(1)$, tak to také dodržíme. Ona totiž slova, která při počítání jednoho b_i hledáme, nejsou úplně libovolná. První z nich je jen značka z_i, druhé je $z_i z_{i+1}$, třetí $z_i z_{i+1} z_{i+2}$ atd. Úplně tedy stačí, když si zapamatujeme, kde jsme při hledání předchozího slova ve stromu skončili, a odtamtud popolezeme o krok doleva nebo doprava podle toho, jestli následuje tečka nebo čárka. To nás pro každé slovo opravdu stojí jen konstantní čas. (Vidíte, tady se nám hodilo, že řetězec procházíme zprava

doleva, jinak bychom totiž museli mít ve slovníku slova uložená pozpátku, což íntnagele ínen étsijaz.)

Celkově náš program spotřebuje čas $O(nL + P)$, kde n je délka vstupního morseovkového řetězce, P celková délka slov ve slovníku a L délka nejdelšího slova.

Poznámka: Někteří z vás chytře využili vyhledávací automat k nalezení všech slov pasujících do textu. Tím se řešení za cenu značného prodloužení potenciálně zrychlí na $O(n+P+V)$, kde V je počet výskytů slovníkových slov v textu. To může být někdy lepší, ale v nejhorším případě může být $V \approx nL$, takže si obecně nepomůžeme.

Špek na závěr: Jak elegantně *převádět jednotlivá písmena do morseovky.* Jistě bychom si mohli nadefinovat pole řetězců s převody jednotlivých písmenek, ale přeci si pěkné řešení nezkazíme takovou obludností. Pomůže dvojková soustava. Místo teček a čárek si představíme nuly a jedničky a přidáme na začátek jedničku (jinak bychom nepoznali `.` od `....`). To je nějaké dvojkové číslo, tak si ho zapíšeme desítkově a až budeme potřebovat tečky a čárky, budeme toto číslo postupně dělit dvěma a zbytky nám povědí dvojkové číslice, čili tečky a čárky. Jen jaksi pozpátku – to ovšem nevadí, tak si ho uložíme obráceně. Radši ukážeme příklad: písmenko `a` se zapisuje jako '`.-`'. Obrátíme na '`-.`', přepíšeme do dvojkové soustavy jako 110, což je šestka.

P.S.: Jak jste jistě uhodli, telegrafická zpráva na začátku našeho řešení zněla *„dynamika a trie“.*

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

#define MAX 256
#define INFTY 10000             // Nekonečno

struct vrchol {                 // Vrchol trie
    struct vrchol *syn[2];      // syn[0] pro tečku, syn[1] pro čárku
    struct slovo *slova;        // Seznam slov, která tu končí
    int hloubka;                // Délka morseovkového zápisu (hloubka v trii)
};
struct vrchol koren;

struct slovo {           // Takhle si ukládáme seznamy slov
    struct slovo *dalsi; // Vlastně by si stačilo pamatovat jen jedno slovo,
    char s[1];           // ale třeba se to bude hodit v následující úloze :)
};

// Tabulka kódů:   A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T
char morse[26] = { 6,17,21, 9, 2,20,11,16, 4,30,13,18, 7, 5,15,22,27,10, 8, 3,
//       U  V  W  X  Y  Z
        12,24,14,25,29,19 };
```

```
void preloz(char *co, char *kam)        // Přeloží slovo do morseovky
{
    while (*co) {
        int k = morse[*co++ - 'a'];     // Ďábelský kód písmenka
        while (k > 1) {                 // Postupně rozkládáme na značky
            *kam++ = ".-"[k%2];
            k /= 2;
        }
    }
    *kam = 0;
}

void nacti_slovnik(void)
{
    char slovo[MAX], mslovo[MAX];
    while (fgets(slovo, sizeof(slovo), stdin) && slovo[0] != '\n') {
        int len = strlen(slovo);
        slovo[len-1] = 0;                       // Smažeme konec řádku
        preloz(slovo, mslovo);                  // Přeložíme do morseovky
        struct vrchol *v = &koren;              // Přidáváme do trie
        for (char *c=mslovo; *c; c++) {
            int z = (*c == '-');                // Aktuální značka
            if (!v->syn[z]) {   // Kam dál? Není-li kam, založíme nový vrchol
                v->syn[z] = malloc(sizeof(struct vrchol));
                memset(v->syn[z], 0, sizeof(struct vrchol));
                v->syn[z]->hloubka = v->hloubka + 1;
            }
            v = v->syn[z];                      // Vydáme se tím směrem
        }
        struct slovo *s = malloc(sizeof(struct slovo) + len);
        // Stojíme ve vrcholu, který odpovídá konci slova,
        s->dalsi = v->slova;      // tak tam slovo přidáme
        v->slova = s;
        strcpy(s->s, slovo);
    }
}

int main(void)
{
    char z[MAX];                                // Vstupní řetězec
    int n;                                      // Jeho délka
    int b[MAX+1];                               // Viz popis řešení
    struct vrchol *s[MAX+1];

    nacti_slovnik();                            // Přečteme slovník a vstup
    fgets(z+1, MAX, stdin);                     // Pozor, indexujeme od 1
    n = strlen(z+1)-1;

    b[n+1] = 0, s[n+1] = NULL;                  // Spočítáme všechna b[i] a s[i]
    for (int i=n; i; i--) {
```

```
        b[i] = INFTY, s[i] = NULL;
        struct vrchol *v = &koren;              // Hledáme pasující slova
        int j = i;
        while (v && j <= n) {
            int zn = (z[j++] == '-');           // Následující značka
            v = v->syn[zn];                     // Poskočíme v trii
            if (v && v->slova && b[j] < b[i])   // Sem nějaké slovo pasuje
               b[i] = b[j]+1, s[i] = v;
        }
    }
    if (b[1] >= INFTY)               // Pokud řešení existuje, tak ho vypíšeme
        puts("Řešení neexistuje");
    else {
        printf("Našli jsme řešení na %d slova:\n", b[1]);
        int i = 1;
        while (i <= n) {
            puts(s[i]->slova->s);
            i += s[i]->hloubka;
        }
    }
    return 0;
}
```

20-2-4 Slovíčkaření — Josef Pihera

Před samotným řešením se zamysleme. Budete mít tisíc ponožek, které jsou úplně stejné, kromě velikosti. Řekněme, že mají celkem čtyři různé velikosti, a vy je máte podle jejich velikosti roztřídit. Pravděpodobně si nikdo z vás nezvolí nějakou ponožku jako medián a nezačne třídit menší vlevo, větší vpravo, ani si jich všech tisíc nerozložíte po podlaze a nebudete je spojovat do dvojic, pak do čtveřic atd. Troufnu si odhadnout, že si zvolíte čtyři přihrádky, a do nich budete ponožky rozřazovat. Ano, toto je nejjednodušší a také nejrychlejší řešení. Takovému postupu se říká *přihrádkové třídění*, nebo anglicky *Bucketsort*.

Nyní opusťme ponožky a navraťme se k řetězcům (názvům knih) a významu abecedy. Na moment předpokládejme, že řetězce mají všechny jednotkovou délku. A je jich hodně, s úspěchem můžeme očekávat, že jich je o dost více než prvků abecedy. Zřejmě tedy bude nejvýhodnější zvolit si jednotlivé znaky abecedy jako přihrádky a řetězce do nich „nastrkat“. Označme si N počet řetězců a A počet prvků abecedy. Tento postup od nás vyžadoval přístup k přihrádkám v $O(A)$ a rozřazení řetězců v $O(N)$.

Zobecněme problém pro řetězce stejných délek, ale delších než jedna. Mnohé by asi napadlo roztřídit řetězce podle prvního písmene, tam kde by to nestačilo, tak podle druhého atd. My si ukážeme něco mazanějšího. Použijeme Bucketsort na poslední písmena řetězců a podle nich je roztřídíme. Potom na předposlední písmena, přičemž u řetězců, které připadnou do stejné přihrádky, zachováme pořadí z předchozího třídění. Tím získáme roztřídění řetězců podle

posledních dvou písmen. My se ale nezastavíme u předposledního a budeme pokračovat dále směrem kupředu. Po i-tém kroku budeme mít řetězce utříděné podle posledních i znaků. V momentě, kdy se dostaneme k prvnímu písmenu a tedy i se bude rovnat společné délce řetězců, máme vyhráno. Řetězce jsou lexikograficky utříděny. Právě jste se seznámili s Radixsortem.

Přitvrdíme a povolíme řetězcům mít různé délky. Předpokládejme, že bychom řetězce doplnili na stejnou délku nějakým znakem, který by byl lexikograficky menší než všechny prvky abecedy. Fungovalo by to, ale mohli bychom si ošklivě ublížit na časové i paměťové složitosti - například bychom měli milion řetězců délky jedna a jeden délky milion ... Naštěstí můžeme správný výsledek získat jedním drobným úskokem. Nejprve si však označme L jako délku nejdelšího z řetězců a P jako součet délek všech řetězců. Pokud si na začátku všechny řetězce setřídíme podle délky (opět přihrádkově), můžeme pak postupovat tak, že v i-tém kroku třídění budeme pracovat pouze s řetězci délky alespoň $L-i+1$, tedy v prvním kroku pouze s řetězci délky L atd. Provedeme to tak, že v každém kroku roztřídíme do přihrádek nejprve řetězce z minulého třídění a tedy s větší délkou. Potom před ně do přihrádek naskládáme řetězce s délkou právě $L-i+1$, tedy řetězce, které znakem na zpracovávané pozici končí. Pokud jste uvěřili správnosti Radixsortu, pak je vám nyní jasné, že pomocí tohoto triku opravdu třídíme, a navíc jen tam, kde je to třeba.

Podívejme se, jak rychle takové řešení funguje. Museli jsme provést L kroků (třídíme řetězce od poslední k první pozici). V každém z těchto kroků jsme museli projít přihrádky v setříděném pořadí a inicializovat je v $O(A)$. Tedy tuto část algoritmu provádíme v $O(L \cdot A)$. Určitě se právě ptáte, kampak se schovalo oněch N řetězců a jak to, že netrvá každý krok třídění ve skutečnosti $O(A+N)$, protože až s N řetězci v každém kroku manipulujeme. Pokusme se na to podívat jinak. Díky našemu triku každý řetězec třídíme až tehdy, když je to třeba a má dostatečnou délku. Do té doby s ním nepracujeme. Můžeme tedy tvrdit, že s každým řetězcem pracujeme dohromady tolikrát, kolik má znaků. To v sumě pro všechny řetězce ale znamená $O(P)$. Podobně počáteční setřídění řetězců podle délky je v $O(L+N)$. Celý algoritmus tak běží v $O(L \cdot A+P+N)$, tj. $O(L \cdot A+P)$. V této knížečce naleznete i vzorovou implementaci. Místo abecedy se v ní uvažují celá čísla v intervalu od 0 do $A-1$, což ale nijak nevadí, protože každá abeceda, kde lze porovnávat, musí být převeditelná na tento případ.

Jako vždy existuje ještě o něco zapeklitější, ale o to efektivnější řešení. Uvědomme si, kde uvedené řešení nejvíce „plýtvá“. V každém kroku prochází všechny přihrádky, bez ohledu na to, jestli jsme je použili, nebo ne. Pokud totiž budeme předem znát, které přihrádky jsou použity, můžeme se dívat pouze do nich a také pouze tyto přihrádky inicializovat a posléze i čistit. Je však důležité, abychom tyto přihrádky znali v setříděném pořadí, protože je tak potřebujeme procházet. Nelze ale pro každý krok tuto informaci počítat zvlášť, protože by

to vždy trvalo nejméně $O(A)$ čemuž se snažíme vyhnout. Proto tento výpočet provedeme na začátku, a to naráz, pro všechny pozice. Uvažujme všechny dvojice (písmeno, pozice), které se v řetězcích vyskytují. Tyto dvojice na počátku setřídíme podle písmen. Tím pro každé písmeno budeme vědět, na kterých pozicích se vyskytuje. Lze si tak vytvořit seznam použitých znaků, pro každou pozici od 1 do L, prostě tak, že projdeme písmena ve vzestupném pořadí a za každou pozici, na které se písmeno vyskytne, přidáme toto písmeno na konec seznamu, který této pozici přísluší. A pak už stačí se při třídění řídit touto informací. Dvojic je P, jako znaků. Tedy toto počáteční předpočítání stíháme v $O(P + A)$. Každý krok algoritmu tak už netrvá $O(A)$. Nyní se dohromady ve všech krocích použije $O(P)$ přihrádek, což je citelně lepší. Tedy výsledná časová i paměťová složitost je $O(N + P + A)$.

Tuto úlohu je možno řešit také pomocí struktury zvané trie, která v základním stavu dává $O(P \cdot A)$, ale lze ji také vylepšit výše zmíněným trikem na $O(N + P + A)$.

```
#include <stdio.h>

int main(void)
{
  int  strnum;      // Počet řetězců
  int  alpha;       // Počet znaků v abecedě
  int* str[strnum]; // Pole řetězců (řetězce uvažujeme abstraktně, takže čísel)
  int  len[strnum]; // Pole délek řetězců
  // Řekněme, že nám obsahy těchto čtyř proměnných někdo naplní
  int lensum = 0,maxlen = 0; // součet délek řetězců, délka nejdelšího
  for(int i = 0; i < strnum; i++) {
    lensum += strlen[i];
    if(maxlen < len[i]) maxlen = len[i];
  }
  // Teď si roztřídíme řetězce podle jejich délky
  int lencnt[maxlen + 2], lens[maxlen + 2]; // mapování přihrádek, přihrádky
  for(int i = 0; i <= maxlen; i++) // inicializace
    lencnt[i] = 0;
  lencnt[maxlen + 1] = strnum; //  <- trik pro zjednodušení dalšího počítání

  for(int i = 0; i < strnum; i++)
    lencnt[len[i]]++;
  for(int i = 1; i <= maxlen; i++)
    lencnt[i] += lencnt[i-1];
  for(int i=0; i < strnum; i++) {
    lencnt[len[i]]--;
    lens[lencnt[len[i]]]=i;
  }

  int _bucks[alpha + 1];               // Mapovací pole přihrádek - aktuální
  int _prevbucks[alpha + 1];           // Mapovací pole přihrádek - předchozí
  int _bstr[strnum], _prevbstr[strnum]; // Přihrádky - opět aktuální a předchozí
```

```
  int *bucks = _bucks;             // tohle nám pomůže,
  int *prevbucks = _prevbucks;     // abychom mohli aktuální a předchozí
  int *bstr = _bstr, *prevbstr = _prevbstr; // snadno zaměňovat

  for(int i = 0; i <= alpha; i++)
    bucks[i] = 0;

  for(int i = maxlen; i>0; i--) {
    int *tmp;
    tmp = bucks; bucks = prevbucks; prevbucks = tmp; // zaměníme mapovací pole
    tmp = bstr;  bstr = prevbstr;   prevbstr = tmp;  // a přihrádky

    for(int j = 0; j <= alpha; j++)
      bucks[j] = 0;

    for(int j = 0; j < prevbucks[alpha]; j++) // napočítáme, kolik bude v při-
      bucks[str[prevbstr[j]][i - 1]]++; // hrádkách řetězců z minulého třídění

    for(int j = lencnt[i]; j < lencnt[i + 1]; j++) // a ještě připočteme
      bucks[str[lens[j]][i - 1]]++;                // řetězce délky i

    for(int j = 1; j < alpha; j++)
      bucks[j] += bucks[j - 1];
    bucks[alpha] = bucks[alpha - 1];  // opět oblíbený trik

    for(int j = 0; j < alpha; j++)
      for(int k=prevbucks[j + 1] - 1; k >= prevbucks[j]; k--)
        // Abychom zachovali uspořádání z předchozího třídění u těch řetězců,
        // které skončí ve stejné přihrádce, musíme je procházet odzadu, stejně
        // jako se odzadu přidávají; protože se přihrádky plní odzadu
        int c = str[prevbstr[k]][i - 1];  // roztřídíme do nich nejprve řetězce
        bucks[c]--;                       // z předchozího třídění, protože jsou
        bstr[bucks[c]] = prevbstr[k];     // delší než i
      }
    for(int j = lencnt[i]; j < lencnt[i + 1]; j++) {
      int c = str[lens[j]][i - 1]; // A teď můžeme do předních částí přihrádek
      bucks[c]--;                  //  Dát řetězce délky i
      bstr[bucks[c]] = lens[j];
    }
  }
  // Vypíšeme výsledek, aby to vypadalo, že jsme opravdu něco udělali...
  for(int i = lencnt[0]; i < lencnt[1]; i++) // všechny prázdné řetězce přijdou
    printf("\n");                            // na začátek (ani jsme je netřídili)

  for(int k = 0; k < bucks[alpha]; k++) { // a teď vypíšeme ty neprázdné
    for(int i = 0; i < len[bstr[k]]; i++)
      printf("%d ",str[bstr[k]][i]);
    printf("\n");
  }
  return 0;
}
```

20-2-5 Hroší ohrádka — Martin „Bobřík“ Kruliš

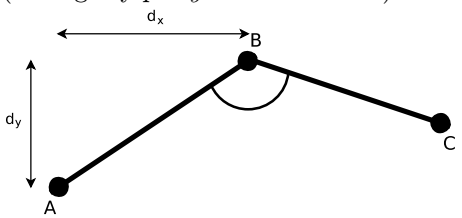
Milí chovatelé hrošíků, přestože nikdo z vás nedosáhl maximálního počtu bodů, poradili jste si s řešením Ohrádky velmi dobře. Našim účastníkům, kteří úlohu vyřešili, věnujeme k Vánocům nějaké ty body. Pro všechny ostatní tu máme alespoň návod, jak si Hroší ohrádku postavit.

Než začneme se samotným hledáním konvexního obalu, setřídíme si souřadnice kůlů. Primárně budeme třídit podle souřadnice x vzestupně (tedy abychom měli kůly v rovině setříděné zleva doprava). Kůly se stejnou x-ovou souřadnicí setřídíme podle y opět vzestupně (tedy zdola nahoru). Náš algoritmus by s drobnými úpravami fungoval, i kdybychom kůly setřídili jinak. Takto ale dostaneme výsledná data rovnou v pořadí, které vyžaduje zadání.

Nyní budeme procházet kůly a vytvoříme *horní část* konvexního obalu. Analogicky pak projdeme kůly ještě jednou a vytvoříme *spodní část*. Obě části nakonec spojíme a dostaneme výsledný konvexní obal. Zbývá ukázat, jak vytvořit horní část obalu (spodní si ani ukazovat nebudeme – sami si rozmyslete, v čem se bude lišit od horní části).

Na začátku vezmeme první kůl – ten který je nejvíce vlevo (a případně také nejvíce dole). Pak postupně přidáváme další kůly a přitom hlídáme, aby se nám neporušila konvexita. Řekněme, že máme kůly $H_1, H_2, ...H_k$, které tvoří začátek horní části konvexního obalu, a právě chceme přidat kůl I. Podíváme se, zda by přidání tohoto kůlu neporušilo konvexnost již hotové části (matematiku okolo si popíšeme za chvíli). V případě, že by nastal problém – tzn. úsečky $H_{k-1}H_k$ a H_kI svírají „špatný“ úhel – vyhodíme H_k z horní části a zkusíme I přidat znovu. Pokud žádný problém nenastane, nebo v horní části je zatím jen jeden kůl, vesele přidáme I.

Nyní nám zbývá vypočítat vzoreček, který bude umět rozhodnout, jaký úhel spolu svírají dvě úsečky. Všimněte si, že nás nezajímá hodnota tohoto úhlu, ale pouze zda je větší nebo menší 180. Úsečky si pro lepší přehlednost označíme AB a BC (jsou tedy spojeny v bodě B) a A_x budeme značit x-ovou souřadnici bodu A (analogicky pro y-ové souřadnice).



Při pohledu na obrázek není těžké si rozmyslet, že tyto úsečky jsou konvexní, pokud je směrnice první úsečky menší, než směrnice druhé:

$$\frac{\delta_x(AB)}{\delta_y(AB)} \leq \frac{\delta_x(BC)}{\delta_y(BC)}$$

Za delty si dosadíme souřadnice bodů:

$$\frac{B_x - A_x}{B_y - A_y} \leq \frac{C_x - B_x}{C_y - B_y}$$

Abychom se zbavili dělení a nemuseli ošetřovat zvlášť případ, kdy jmenovatel některého zlomku je nulový, rozšíříme si rovnici na tvar:

$$(B_x - A_x)(C_y - B_y) \leq (B_y - A_y)(C_x - B_x)$$

Z matematického hlediska není výše uvedená úprava úplně v pořádku. Ovšem není těžké si rozmyslet jak se chovají nevhodné případy (tj. $B_y - A_y = 0$ nebo $C_y - B_y = 0$), a že výsledná rovnice je vlastně to, co jsme celou dobu chtěli.

Nalezenou nerovnici stačí použít jako logickou podmínku, která otestuje, zda jsou dvě úsečky konvexní. Pro spodní část konvexního obalu použijeme stejnou nerovnici, ale s opačnou nerovností.

A nyní se podíváme, jak je na tom časová a paměťová složitost. Někteří čtenáři si možná všimli, že výše popsaný algoritmus vypadá jako backtracking a backtracking je často spojován s exponenciální časovou složitostí. Panika ovšem není na místě, protože náš „skoro-backtracking“ je hodný a pokorně seběhne v lineárním čase. Idea důkazu je jednoduchá. Sice se může stát, že při přidávání některého z kůlů budeme muset hodně kůlů z již hotové části vyhodit, ale na druhou stranu víme, že každý kůl do vytvářené části (horní i dolní) vložíme právě jednou a nejvýše jednou každý kůl vyhodíme. Při počítání složitosti ještě nesmíme zapomenout, že jsme na začátku kůly třídili, a třídění nám zabere $O(N \cdot \log N)$. Takže celková časová složitost bude: $O(N \cdot \log N + N) = O(N \cdot \log N)$.

Paměťová složitost je lineární, protože si potřebujeme pamatovat pouze načtené kůly a vytvářený obal (a obal nemůže obsahovat víc, než původní množství kůlů).

```
#include <stdio.h>
#include <stdlib.h>
// Test, zda úsečky AB a BC svírají správný úhel pro horní část obalu.
#define OK_FOR_TOP(A,B,C) ((B.x - A.x)*(C.y - B.y) <= (B.y - A.y)*(C.x - B.x))
// Test, zda úsečky AB a BC svírají správný úhel pro spodní část obalu.
#define OK_FOR_BOTTOM(A,B,C) ((B.x - A.x)*(C.y - B.y)>=(B.y - A.y)*(C.x - B.x))

struct spoint {         // Struktura zapouzdřující souřadnice jednoho kůlu.
        int x, y; };
```

```
struct spoint *points;  // Pole všech kůlů.
int pointsCount = 0;    // Počet všech kůlů.
/* Načte data ze vstupního souboru do globálních proměnných. */
void load(void) {
        FILE *fp = fopen("kuly.in", "r");
        if (!fp) return;

        fscanf(fp, "%d\n", &pointsCount);
        points = malloc(pointsCount * sizeof(struct spoint));

        for(int i = 0; i < pointsCount; i++)
                fscanf(fp, "%d %d\n", &(points[i].x), &(points[i].y));
}
/* Funkce, která porovná souřadnice dvou bodů a vrátí, zda jsou menší,
 * větší, nebo rovny. Použije se jako predikát pro QuickSort. Vracíme
 * záporné číslo, pokud je item1 < item2, kladné, pokud je item1 > item2
 * a nulu, pokud jsou si rovny. */
int compare(const void *item1, const void *item2) {
        struct spoint *p1 = (struct spoint*)item1;
        struct spoint *p2 = (struct spoint*)item2;

        if (p1->x == p2->x) // X-ové souřadnice jsou stejné, porovnáme y-ové.
                return p1->y - p2->y;
        else
                return p1->x - p2->x;
}
/* Nalezne horní a dolní část konvexního obalu a vypše je do souboru. */
void process(void) {
        if (pointsCount == 0) return;

        // Horní část konvexního obalu.
        struct spoint *resTop = malloc(pointsCount * sizeof(struct spoint));
        // Spodní čast konvexního obalu.
        struct spoint *resBtm = malloc(pointsCount * sizeof(struct spoint));
        int resTopIdx = 0, resBtmIdx = 0;       // Index posledního kůlu v poli.
        int i = 1;                              // Index právě zpracovávaného kůlu.
        // Vložím počáteční bod do horní i dolní části konvexního obalu.
        resTop[0] = points[0];
        resBtm[0] = points[0];
        // Zpracujeme všechny kůly, které leží na stejné x-ové souřadnici,
        // jako první bod.
        while((i < pointsCount) && (points[i].x == points[0].x))
                resTop[++resTopIdx] = points[i++];
                // Přidáme je jen do horní hranice.
        // Projdu v cyklu (ve správném pořadí) všechny kůly.
        while(i < pointsCount) {
                // Horní část konvexního obalu: vyhodíme všechny kůly,
                // které porušují konvexitu s novým kůlem i.
                while((resTopIdx > 0) && !OK_FOR_TOP( resTop[resTopIdx-1],
                                resTop[resTopIdx], points[i] ) )
                        resTopIdx--;
```

```
                // Přidáme kůl "i" do horní části.
                resTop[++resTopIdx] = points[i];

                // Spodní část: vyhodíme všechny kůly,
                // které porušují konvexitu s novým kůlem i.
                while((resBtmIdx > 0) && !OK_FOR_BOTTOM( resBtm[resBtmIdx-1],
                                resBtm[resBtmIdx], points[i] ) )
                        resBtmIdx--;
                // Přidáme kůl "i" do spodní.
                resBtm[++resBtmIdx] = points[i];
                i++;    // Vezmem další kůl.
        }
        // Vypíšeme výsledky do souboru.
        FILE *fp = fopen("plot.out", "w");
        fprintf(fp, "%d\n", resTopIdx + resBtmIdx);
        for(i = 0; i < resTopIdx; i++)  // Vrchní časť vypíšeme normálně.
                fprintf(fp, "%d %d\n", resTop[i].x, resTop[i].y);
        // Spodní část vypíšeme pozpátku a bez posledního prvku.
        for(i = resBtmIdx; i > 0; i--)
                fprintf(fp, "%d %d\n", resBtm[i].x, resBtm[i].y);
}
int main(void) {
        load();         // Načteme souřadnice.
        qsort(points, pointsCount, sizeof(struct spoint), compare);
        process();      // Nalezneme konvexní obal a vypíšeme jej do souboru.
        return 0;
}
```

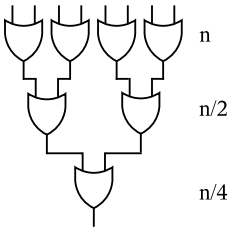
20-2-6 Hrady, hrádky, hradla — Cyril Hrubiš

V druhé sérii našeho seriálu o elektronice, hradlech a radostech podobných jste měli za úkol vymyslet „hromadný“ OR. Je mnoho způsobů, jak se dá taková funkce OR realizovat. Jak už je obvyklé, měli jste být při návrhu šetrní a ještě navíc, dokázat že vaše řešení je nejšetrnější. Nejdříve dokážeme, že na spočítání jednoho výstupu z n vstupů potřebujeme nejméně $n-1$ hradel. Poté ukážeme, že k takovému výpočtu je třeba minimálně logaritmický počet hladin a to $\lceil \log_2(n) \rceil$. Následně ukážeme konstrukci takového obvodu pro obecné n.

Začneme nejdříve důkazem minimálního počtu hradel. K dispozici máme jenom a pouze hradla dvouvstupová. Výstupy hradel se dle definice z prvního dílu seriálu spojovat nesmí, proto přidáním hradla můžeme ze dvou „drátů“ vyrobit jeden. Naopak rozdvojování „vodičů“ nám nijak nepomůže. Tedy obecně pro n „vodičů“ na vstupu potřebujeme $n-1$ hradel abychom je zredukovali postupným přidáváním hradel na jeden výstup.

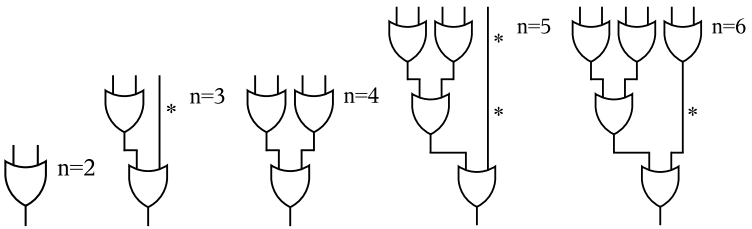
Nyní se vrhneme na minimální odhad počtu hladin. Jak už víme z předchozího odstavce, jedno hradlo nám ze dvou „drátů“ umí vyrobit jeden a nic lepšího se nám s jedním hradlem vyrobit nepodaří. Tedy za jednotku času, což

je čas na průchod informace jedním hradlem, umíme z n „vodičů“ vyrobit minimálně $n/2$. Hladin tedy bude tolik, kolik členů posloupnosti $n, \frac{n}{2}, \frac{n}{2^2}, \frac{n}{2^3}, \ldots, 2$. Z toho nám vychází počet hladin $\lceil \log_2(n) \rceil$.



Nyní víme, jaké má mít takový obvod parametry. Zbývá vymyslet, jak vlastně vypadá. Pro n, jež je mocninou dvojky je konstrukce jednoduchá a přímo plyne z důkazu minimální hloubky zapojení. V takovém případě bude mít každá hladina postupně $n, \frac{n}{2}, \frac{n}{2^2}, \frac{n}{2^3}, \ldots, 2$ hradel, jejichž vstupy jsou zapojené do výstupů předchozí vrstvy. Na obrázku je takový obvod pro $n = 8$.

Pro obecné n budeme obvod konstruovat jakoby indukcí. Začneme pro $n = 2$ jedním hradlem OR. Hradla budeme přidávat dle následujících pravidel. Nejdříve si ale zadefinujeme pojem zaplněná hladina. To je taková, jejíž hranicí neprochází vodič. Následně pokud existuje nějaká nezaplněná hladina, přidáme hradlo OR, tak, že prochází drátem, který procházel hranicí hladiny, tím nám přibyde jedno hradlo a jeden vstup, který protáhneme do vstupní hladiny (tím nám může přibýt prázdných hladin). Pokud prázdné hladiny nejsou, přidáme hradlo tak, že bude dělat OR mezi dosavadním výstupem a přidáváným vstupem (tím nám pro $n \neq 2^n - 1$ jistě vzniknou volné hladiny). Na obrázku je prvních pět kroků konstrukce, hvězdičkami jsou označeny volné hladiny.



20-3-1 Platba koně — Michal „vorner“ Vaner

Napřed provedeme malý trik. Každá hodnota je s přesností na 2 desetinná místa. Když všechny hodnoty vynásobíme 100 (tedy, převedeme z Korun na Haléře), dostaneme celá čísla, se kterými se mnohem lépe pracuje. Mimo to, ne každý handlíř umí pracovat s desetinnými čísly.

Nyní si na chvíli představme, že handlíř je naprosto chudý a nemá ani vindru (tedy, jeho hromádka na vracení je prázdná). Mimo to, je rozumné mít pouze kladné mince a kladnou cenu koně (i když, u některých koní ...).

Jak by se dal řešit takový problém? Půjdeme na to od lesa. Rozdělíme to na fáze a chceme po k-té fázi vědět, které všechny obnosy lze zaplatit pomocí k prvních mincí.

Stav na začátku, tedy po nulté fázi, je jasný. Dokážeme zaplatit jedinou částku a to 0.

V každé fázi vezmeme minci o hodnotě h a vedle každé částky zaplatitelné pomocí $k-1$ prvních mincí přidáme do naší množiny ještě částku zvětšenou o h.

Z tohoto již lze snadno poznat, že daná částka jde zaplatit. Jednoduše se bude vyskytovat v množině. Jak ale poznat, kterými mincemi ji máme zaplatit? Při přidávání každé částky do množiny si k ní připíšeme také, ze které částky vznikla. Odečtením získáme poslední použitou minci a když se podíváme na onu předchozí částku, tak můžeme obdobným způsobem zrekonstruovat předchozí minci, až se dostaneme na začátek.

Musíme vyřešit, jak budeme reprezentovat tuto množinu. Všimneme si, že veškeré hodnoty jsou celá čísla a proto i jejich součty budou celá čísla. Navíc, nikdy nebude menší než 0 a větší, než součet všech Vildových mincí s_v. Můžeme použít pole, jehož indexy budou čísla $0 \dots s_v$, v každé fázi toto pole projít a poznamenat do něj hodnoty nové.

Dále je třeba zajistit, aby námi vybraná možnost byla ta, která má nejméně použitých mincí. Inu, zavedeme tuto podmínku do každé fáze, tedy na konci k-té fáze budeme mít všechny zaplatitelné obnosy a každý na nejmenší počet mincí. Když budeme chtít poznamenat, že lze zaplatit nějaká hodnota a tato hodnota již zaplatit jde, tak ji přepíšeme na novou jen v případě, že je tato nová na méně mincí. Zřejmě to funguje, neboť ta s nejmenším počtem možností buď používá k-tou minci a nebo nepoužívá, což je přesně to, co ověřuje tato podmínka.

Poslední problém, který je třeba vyřešit, jsou handlířovy mince. Jednoduchým trikem je sesypeme do stejného pytle jako mince Vildovy, jen je všechny vynásobíme číslem -1. Tento trik vypadá jednoduše, ale je třeba vyřešit několik detailů. Hlavním z nich je rozsah pole. Již není pravda, že by součet některých mincí nemohl být záporný. Avšak, když vyzkoušíme napřed všechny kladné mince a potom nám už zbudou jen záporné, tak zaznamenávat, že umíme zápornou částku k ničemu není, neboť ji již nikdy nad nulu nedostaneme.

Stejně tak si rozmyslíme horní hranici. Určitě se nikdy nedostaneme nad součet všech Vildových mincí. Ale také nemá nikdy cenu ukládat částky, které přesahují cenu koně a součet handlířových mincí dohromady, takové částky již nedokážeme dostatečně snížit, i kdyby handlíř vrátil vše, co měl. Stačí tedy vzít minimum z těchto dvou hodnot.

Označme toto minimum jako μ. Potom paměťová složitost je očividně $O(\mu + M + N)$, kde N a M jsou počty mincí na jednotlivých hromádkách. Časová složitost je $O(\mu \cdot (N + M))$, neboť pro každou minci proběhne jedna fáze a v každé fázi projdeme celé pole.

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

typedef struct {
        int predchozi;
        int minci;
} castka_t;

#define INFINITY 0x4fffffff // Takto se pozná něco, co nejde zaplatit

void pridej( castka_t castky[], int mince, int index, int limit ) {
        int nova = index + mince;
        if( nova < 0 || nova > limit ) // mimo rozsah
                return;
        if( castky[ index ].minci + 1 < castky[ nova ].minci ) {
                castky[ nova ].predchozi = index;
                castky[ nova ].minci = castky[ index ].minci + 1;
        }
}

int main( int argc, const char *argv[] ) {
        int n, m, p, vil_celkem, han_celkem;
        float prep; // Dočasné, na nehaléřové hodnoty
        scanf( "%d%d%f", &n, &m, &prep );
        p = ( int ) ( prep * 100 ); // Převod na haléře
        int mince[ m + n ];
        vil_celkem = han_celkem = 0;
        for( int i = 0; i < m + n; ++ i ) {
                float nova;
                scanf( "%f", &nova );
                nova *= 100; // Převod na haléře
                mince[ i ] = ( int ) nova;
                if( i >= n ) {
                        han_celkem += mince[ i ];
                        mince[ i ] *= -1; // Handléřovy mince "vracejí"
                } else {
                        vil_celkem += mince[ i ];
                }
        }
        int limit = han_celkem + p; // Vybrat vhodný rozsah pole
        if( vil_celkem < limit )
                limit = vil_celkem;
        if( p > limit ) {
                printf( "Na to Vilda nemá\n" );
                return 0;
        }
        if( p < 0 ) {
                printf( "Nepodporujeme záporné koně\n" );
                return 0;
        }
        castka_t castky[ limit + 1 ];
```

```
        for( int i = 1; i <= limit; ++ i ) {
                castky[ i ].minci = INFINITY;
        }
        castky[ 0 ].predchozi = 0;
        castky[ 0 ].minci = 0;
        for( int i = 0; i < n; ++ i )
                // Opačně, abychom nenacházeli z této fáze
                for( int j = limit; j >= 0; -- j )
                        pridej( castky, mince[ i ], j, limit );
        for( int i = n; i < n + m; ++ i )
                // Záporné mince -> tímto směrem
                for( int j = 0; j <= limit; ++ j )
                        pridej( castky, mince[ i ], j, limit );
        if( castky[ p ].minci == INFINITY ) {
                printf( "Koně zaplatit nelze\n" );
                return 0;
        }
        while( p ) {
                printf( "%f\n", ( p - castky[ p ].predchozi ) / 100.0 );
                p = castky[ p ].predchozi;
        }
        return 0;
}
```

20-3-2 Dva lupiči — Petr Kratochvíl

Máme čtyři výroky lupičů, a každý z nich nějakým způsobem omezuje možné dvojice čísel. Projdeme si tyto omezující podmínky jednu po druhé. Označme si velitelova čísla jako x, y; víme, že $2 \leq x, y \leq 99$.

První věta nám říká, že součin xy jde rozložit na součin dvou čísel více než jedním způsobem. To znamená, že xy je součin alespoň tří prvočísel, která nejsou všechna stejná. Označme A_1 množinu všech těchto povolených součinů.

Druhá věta nám říká, že ať součet $x + y$ rozložíme na součet dvou čísel jakkoliv (čímž získáme řekněme čísla $\alpha, \beta \in \langle 2, 99 \rangle$, kde $\alpha + \beta = x + y$), vždycky bude součin $\alpha\beta$ ležet v množině A_1. Množinu všech součtů, které toto splňují, označme B_1.

Z třetí věty víme, že součin xy jde rozložit na součin dvou čísel α, β tak, aby $\alpha + \beta \in B_1$, právě jedním způsobem. Množinu všech součinů, které tuhle podmínku splňují, označíme A_2.

A poslední věta nám říká, že součet $x + y$ jde rozložit na součet dvou čísel α, β tak, aby $\alpha\beta \in A_2$, právě jedním způsobem. Množinu všech součtů, které to splňují, označíme B_2.

Chceme teď najít všechna čísla x, y taková, že $xy \in A_1 \cup A_2$ a $x + y \in B_1 \cup B_2$. Tím jsme úspěšně formulovali problém matematicky, ale jak ho vyřešit? Inu, pomocí čtyřech uvedených podmínek vyškrtáme zakázané součiny a součty a uvidíme, co zbude.

Povolené součty jsou v intervalu $\langle 4, 198\rangle$, a chceme z nich vyškrtat ty, které se dají zapsat jako součet dvou prvočísel, nebo jako součet prvočísla a jeho druhé mocniny.

Aby to vyškrtávání zabralo méně času, můžeme využít *Goldbachovu hypotézu (http://en.wikipedia.org/wiki/Goldbach_conjecture)*, která říká, že každé sudé číslo (větší než dva) je možné zapsat jako součet dvou prvočísel. Sice jde pouze o hypotézu (a slavný otevřený problém), ale na počítači byla její platnost ověřena (alespoň) pro čísla do 10^{18}. Takže nám stačí škrtnout všechna sudá čísla, a z lichých ta, co jsou o 2 větší než nějaké prvočíslo. A součet prvočísla a jeho druhé mocniny je vždycky sudý.

Pozor, je tu jedna záludnost. Potřebujeme, aby ta dvě prvočísla byly přípustné hodnoty čísel x, y, tedy aby byla v intervalu $\langle 2, 99\rangle$. Může se nám stát, že číslo sice rozložíme na součet dvou prvočísel, ale pokud jedno z těch prvočísel bude moc velké, stále se jedná o přípustný součet. Na druhou stranu součiny jako $99 \cdot 99$ přípustné nejsou, i když jde o součin alespoň tří prvočísel, která nejsou všechna stejná.

Teď projdeme všechny dvojice čísel z intervalu $\langle 2, 99\rangle$ (možné součiny). Rovnou škrtneme součin dvou prvočísel a třetí mocninu prvočísla. A nakonec projdeme všechny dělitele d možného součinu s a podíváme se, jestli existuje právě jeden dělitel d tak, že součet $d + s/d$ je v množině B_1 povolených součtů.

A zbývá aplikovat poslední podmínku. Projdeme si množiny povolených součtů a pro každý z nich si najdeme všechny jeho rozklady na součet dvou čísel α, β z intervalu $\langle 2, 99\rangle$. Pokud mezi všemi rozklady existuje právě jeden, pro který je součin $\alpha\beta$ povolený (nevyškrtnutý), našli jsme řešení.

Pro ruční procházení je těch možností docela hodně, a tak se vyplatilo napsat si program. Pro výpočetní sílu počítače je však jejich počet nepatrný, a proto dáme přednost jednoduššímu kódu před optimalizacemi pomocí prvočísel.

Nuže konečně uvedu dlouho očekávaný výsledek, úlohu řeší právě dvojice čísel $\{4, 13\}$.

```
#include <stdio.h>

int vysledek_x, vysledek_y;

int a_nevi(int soucin) {
        int rozklad = 0;
        for (int i = 2; (i <= 99) && (i*i <= soucin); i++)
                if ((soucin % i == 0) && (soucin/i <= 99))
                        rozklad++;
        if (rozklad >= 2)
                return 1;
        else return 0;
}
```

```
int b_vi_ze_a_nevi(int soucet) {
        for (int i = 2; (i <= 99) && (2*i <= soucet); i++)
                if (!a_nevi(i*(soucet-i)))
                        return 0;
        return 1;
}

int a_uz_vi(int soucin) {
        int rozklad = 0;
        if (!a_nevi(soucin))
                return 0;
        for (int i = 2; (i <= 99) && (i*i <= soucin); i++) {
                if ((soucin % i == 0) && (b_vi_ze_a_nevi(soucin/i + i)))
                        rozklad++;
                if (rozklad > 1)
                        break;
        }
        if (rozklad == 1)
                return 1;
        else return 0;
}

int b_taky_vi(int soucet) {
        int rozklad = 0;
        if (!b_vi_ze_a_nevi(soucet))
                return 0;
        for (int i = 2; (i <= 99) && (2*i <= soucet); i++)
                if (a_uz_vi(i*(soucet-i))) {
                        rozklad++;
                        vysledek_x = i;
                        vysledek_y = soucet - i;
                }
        if (rozklad == 1)
                return 1;
        else return 0;
}

int main(void) {
        for (int i=2; i<=198; i++)
                if (b_taky_vi(i))
                        printf("(%d, %d)\n", vysledek_x, vysledek_y);
        return 0;
}
```

20-3-3 Cesta z kopce — Petr Onderka

Úloha jde nelépe vyřešit, jak si drtivá většina z vás všimla, v čase $O(N)$. Budeme hledat nejdelší podposloupnost končící nějakým členem posloupnosti A, která splňuje zadání (v dalším textu podposloupnost znamená podposloupnost splňující zadání, tedy obsahující nejvýše k stoupání).

Začneme s podposloupností obsahující jen zvolený prvek a její začátek postupně posunujeme směrem k počátku posloupnosti A. Dokud podposloupnost obsahuje méně než k stoupání, je všechno v pořádku. Jakmile ji ale rozšíříme tak, že už má právě k stoupání, musíme pokračovat opatrně, abychom nepřidali další stoupání. Skončíme tedy těsně za nějakým stoupáním (na druhém prvku z rostoucí dvojice), které by už překročilo limit, případně na začátku celé posloupnosti A. Pokud takto najdeme nejdelší podposloupnost pro každý prvek z A, bude mezi nimi určitě hledaná celkově nejdelší. Obdobnou úvahou při posunování konce podposloupnosti místo začátku zjistíme, že konec hledané podposloupnosti bude těsně před nějakým stoupáním, případně na konci celé posloupnosti.

Když obě úvahy spojíme dohromady, zjistíme, že není třeba hledat začátek a konec podposloupnosti jinde než u stoupání a na úplném začátku nebo konci. Najdeme si tedy všechna stoupání v posloupnosti A a pro každé z nich hledáme nejdelší podposloupnost, která končí těsně před ním. Pokud právě zkoumané stoupání, bude i-té, tak pro začátek podposloupnosti musíme přeskočit k stoupání a bude tedy ležet hned za $(i-k-1)$-tým. Hodnoty $i \leq k$ vůbec nemusíme uvažovat, protože u nich končící podposloupnosti budou začínat prvním prvkem A, stejně jako pro $i = k+1$, pro nějž bude délka větší než pro všechna menší i.

Z výše uvedeného vyplývá, že pro zjištění výsledku si vůbec nemusíme pamatovat hodnoty A, kromě dvou posledních k zjištění stoupání. Navíc stačí znát jen $k+1$ stoupání před aktuálním, jelikož stoupání $k+1$ míst před právě zkoumaným potřebujeme v tomto kroku a následující budeme potřebovat v dalších krocích. Hledání nejdelší posloupnosti pak může probíhat přímo při hledání stoupání. Nejdříve si zapamatujeme prvních $k+1$ stoupání a pak vždy když najdeme další, zkontrolujeme délku podposloupnsti, která u něj končí a začíná u nejstaršího stoupání, které si pamatujeme, což je právě to $k+1$ míst před aktuálním. Toto nejstarší stoupání pak zapomeneme a zapamatujeme si právě nalezené. Taková datová struktura se nazývá fronta. Nakonec si ještě musíme dát pozor, pokud je počet stoupání nejvýše k, abychom za řešení přijali celou posloupnost.

Časová složitost je $O(N)$, protože procházíme jedenkrát zadanou posloupnost a pro každý její prvek provádíme konstantně mnoho operací. Paměťová složitost je $O(k)$, kvůli frontě délky $k+1$.

```
program cestaZkopce;
const MaxK = 20;
var N, k, a, b, i, j, pred, akt: integer;
    stoupani: array [0..MaxK] of integer;
begin
 a := 1;
 b := 1;
```

```
 stoupani[0] := 1;
 j := 1;
 read(N, k);
 read(pred);
 for i:=2 to N do begin
  read(akt);
  if pred < akt then begin                          {máme stoupání}
   if j <= k then                                   {ale zatím málo}
    stoupani[j] := i
   else begin                                       {teď už dost}
    if i-1 - stoupani[j mod (k+1)] > b - a then begin
{zbytek po dělení používám proto, aby pole stoupani bylo zatočené do kruhu}
     a := stoupani[j mod (k+1)];                    {zatím nejdelší podposloupnost}
     b := i - 1;
    end;
    stoupani[j mod (k+1)] := i;                     {uložíme do fronty}
   end;
   inc(j);
  end;
  pred := akt;
 end;
 {ještě zkontrolujeme poslední podposloupnost}
 if N - stoupani[j mod (k+1)] > b-a then begin
  if j <= k then a := 1                             {vezmeme celou posloupnost}
  else a := stoupani[j mod (k+1)];  {nebo jen od prvního stoupání z fronty}
  b := N;
 end;
write('a = ', a, ' b = ', b);
end.
```

20-3-4 Orientace na mapě — Tereza Klimošová

Nejprve si nejspíš uvědomíme, že v acyklickém orientovaném grafu musí být alespoň jeden vrchol, do kterého nevede žádná hrana – zdroj. Z každého vrcholu (který není zdroj) můžeme cestou proti směru hran dojít do nějakého zdroje. Proto při hledání vrcholů, mezi nimiž vede nejvíce cest, můžeme předpokládat, že počáteční vrchol je zdroj – kdyby cesty vycházely z jiného vrcholu, můžeme všechny prodloužit až do nějakého zdroje. Tím se jejich počet určitě nezmenší. (Z podobného důvodu bychom také mohli hledat koncové vrcholy pouze ve stocích – vrcholech z nichž nevede žádná hrana.)

Vzápětí si uvědomíme, že zdrojů v grafu může být mnoho, takže nám tohle pozorování práci neušetří a algoritmus nezlepší, ale využít ho můžeme ... Z každého zdroje tedy spočítáme cesty do jednotlivých vrcholů.

Máme-li pro nějaký vrchol v spočíst počet cest z určitého zdroje, lze to udělat tak, že sečteme počty cest do všech vrcholů, ze kterých vede hrana do v. K tomu ovšem musíme tyto počty cest znát. Proto je nutné počítat cesty do vrcholů ve správném pořadí – v topologickém pořadí (o němž se píše v kuchařce ke třetí sérii). Topologické pořadí určíme například tak, že projdeme graf ze

zdroje do hloubky a pamatujeme si pořadí, ve kterém jsme naposled opouštěli jednotlivé vrcholy - tímto způsobem je dostaneme setříděné pozpátku – zdroj bude úplně poslední, takže musíme počítat počty cest do vrcholů od konce. Když máme spočítané cesty do všech vrcholů, zapamatujeme si maximální počet cest (a kam vedly) a prozkoumáme cesty z dalšího zdroje. Pak už stačí jenom vybrat zdroj, z něhož vede nejvíce cest.

Jak to všechno bude složité? Na jednotlivé průchody do hloubky potřebujeme $O(N + M)$ času. Počet potřebných průchodů závisí na počtu zdrojů v grafu, může být až $O(N)$. Celkem se dostáváme na časovou složitost $O(N \cdot (N + M))$. Paměťová složitost vzorového řešení je $O(N^2)$, protože si seznamy sousedů pamatuje ve zbytečně dlouhých polích, při šikovnějším způsobu by stačilo $O(N + M)$.

```
program mapa;

const C=42;
type vrchol = record
                hrany:array [1..C] of integer;
                deg:integer;
                zdroj:boolean;
                prosel:boolean;
                cesty:integer;
                end;

var
  graf:array [1..C] of vrchol;
  topol:array[1..C] of integer;
{cisla vrcholu v topologickem poradi}
  maxvrchol,maxhodnota:integer;{kde najdeme maximum a jaka je jeho hodnota}
  zdr:integer; {pocet zdroju}
  top:integer;{pocet vrcholu v topologickem usporadani z aktualniho zdroje}
  i,j,k,N,u,v:integer;

  zdroje:array [1..C] of record
                           start,cil,cesty:integer;
                           end;

procedure pruchod(i:integer);
  var m:integer;
  begin
    m:=1;
    if not graf[i].prosel then begin
     while graf[i].deg>=m do
        begin

          pruchod(graf[i].hrany[m]);
          inc(m);
        end;
     graf[i].prosel:=TRUE;
```

```
          inc(top);
      topol[top]:=i;
          end;
  end;

begin
{nacteni}
   readln(N);
 {pocet vrcholu}
   for i:=1 to N do begin
     graf[i].deg:=0;
     graf[i].zdroj:=TRUE;
     end;
   readln(u,v);
   while u<>0 do begin
     inc(graf[u].deg);
     graf[v].zdroj:=FALSE;
     graf[u].hrany[graf[u].deg]:=v;
     readln(u,v);
     end;
{nalezeni zdroju}
   zdr:=0;
   for i:=1 to N do begin
     if graf[i].zdroj then begin
                             inc(zdr);
                             zdroje[zdr].start:=i;
                           end;
   end;

{pruchod do hloubky - topologicke trideni ze zdrojovych vrcholu}
    for i:=1 to zdr do begin
      for j:=1 to N do graf[j].prosel:=FALSE;
      top:=0;
      pruchod(zdroje[i].start);
{spocitani cest}
      for j:=1 to N do graf[j].cesty:=0;
      graf[topol[top]].cesty:=1;
      for  j:= top downto 1 do begin
        v:=topol[j];
        for k:=1 to graf[v].deg do begin
          u:=graf[v].hrany[k];
          graf[u].cesty:=graf[u].cesty+graf[v].cesty;
          end;
        end;
{vybrani nejvetsiho poctu cest}
     maxhodnota:=0;
      for j:=1 to N do if graf[j].cesty>maxhodnota
        then begin
          maxhodnota:=graf[j].cesty;
          maxvrchol:=j;
        end;
```

```
      zdroje[i].cil:=maxvrchol;
      zdroje[i].cesty:=maxhodnota;
      end;
      maxhodnota:=0;
      for i:=1 to zdr do if zdroje[i].cesty>maxhodnota
        then begin
          maxhodnota:=zdroje[i].cesty;
          maxvrchol:=i;
        end;
      writeln('Mezi vrcholy ',zdroje[maxvrchol].start,' a ',
        zdroje[maxvrchol].cil,'vede ',zdroje[maxvrchol].cesty,'cest.');

end.
```

20-3-5 Asfaltování — Martin „Bobřík“ Kruliš

Zdravím všechny řešitele upatlané od asfaltu. Mám pro vás dobrou zprávu: Nebojte, za pár měsíců se to oloupe. A nyní vám přinášíme exkluzivně algoritmus od samotného vrchního cestáře, který nám jej (samozřejmě pod pohrůžkou mučení) vyzradil.

Nejprve si naši úlohu převedeme do řeči grafů. Města představují vrcholy, cesty mezi nimi budou hrany, a protože lze cestovat po celé Hipopotámii, bude graf souvislý. Chceme najít párování hran takové, aby každá hrana byla spárovaná a dvě spárované hrany měly společný vrchol. Nedá mnoho přemýšlení, že zmíněné párování nemůže existovat, pokud je počet hran lichý. Od teď se tedy budeme zabývat pouze grafy, které mají sudý počet hran.

Klíčem k řešení našeho problému bude algoritmus prohledávání do hloubky, též známý jako DFS (Depth First Search). Podívejme se, jak bude vypadat situace vstoupíme-li při procházení do vrcholu u. Nejprve zpracujeme všechny dosud nenavštívené sousedy vrcholu u, jak nám káže algoritmus DFS. Následně se podíváme, s kolika nespárovanými hranami vrchol inciduje. Je-li jich sudý počet, můžeme spárovat hrany libovolně mezi sebou. Pokud jich je lichý počet, necháme hranu vedoucí k otci (k vrcholu, ze kterého jsme do u přišli při DFS) nespárovanou (taťka se nám o ni postará) a zbývající hrany, kterých už je sudě, opět libovolně spárujeme.

Zbývá ukázat, že u vrcholu s, ve kterém jsme prohledávání započali, budeme mít na konci algoritmu sudý počet nespárovaných hran. Kdyby tomu tak nebylo, měli bychom problém, protože s už nemá žádného „taťku“, který by mu pomohl vyřešit problémy s lichou hranou. Naštěstí ale víme, že na začátku máme sudý počet (nespárovaných) hran. Při párování nám ubývají nespárované hrany vždy po dvou, takže se zachovává jejich sudý počet. V okamžiku, kdy se vrátíme v DFS zpět do vrcholu s, můžou být nespárované pouze hrany incidující s s. A protože jich je celou dobu sudý počet, můžeme je vesele spárovat.

Budeme-li reprezentovat graf seznamem sousedů, je časová i paměťová složitost algoritmu $O(N + M)$, kde N je počet vrcholů a M je počet hran.

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

#define MAXN    100000
#define MAXM    500000
#define SWAP_INT(x, y) { int tmp = x; x = y; y = tmp; }

/* Struktura uchovávající údaje o jedné hraně */
struct s_edge {
        int a, b;       // Mezi kterými vrcholy hrana vede.
        int pair;       // Se kterou hranou je spárovaná (-1, pokud ještě není).
};

/* Počátek hran od daného vrcholu v seznamu hran a stupeň vrcholu */
struct s_vertex {
        int start;              // Index první hrany v seznamu hran (resp. poli
                                // "ep"), která inciduje s tímto vrcholem.
        int deg;                // Stupeň vrcholu.
        int visited;    // Zda již byl vrchol navštíven.
};

/* Globální proměnné */
struct s_edge edges[MAXM];       // Seznam hran (tak jak je načten ze souboru).
struct s_vertex vertices[MAXN];  // Seznam vrcholů.
int ep[2*MAXM];                  // Přeuspořádání hran (kvůli seznamu sousedů).
int N, M;                        // Počet vrcholů a hran.

/* Načte vstupní data a vytvoří reprezentaci grafu. */
void read_input(void)
{
        /* Otevřeme soubor */
        FILE *fp = fopen("asfalt.in", "r");
        if (!fp) {
                perror("Cannot open input file");
                exit(1);
        }
        /* Přečteme počet vrcholů a hran. */
        fscanf(fp, "%d %d", &N, &M);
        /* Ošetříme speciální případ, kdy je M liché a úloha tak nemá řešení. */
        if ((M % 2) == 1) {
                FILE *fout = fopen("asfalt.out", "w");
                if (!fout) {
                        perror("Cannot open output file");
                        exit(1);
                }
                fputs("no\n", fout);
                exit(0);
```

```
        }
        /* Načteme seznam hran */
        for (int i = 0; i < M; i++) {
                fscanf(fp, "%d %d", &edges[i].a, &edges[i].b);
                edges[i].a--;  edges[i].b--;    // Indexujeme města od 0 do N-1.
                edges[i].pair = -1;             // Hrana ještě není spárovaná.
                vertices[edges[i].a].deg++;
                vertices[edges[i].b].deg++;
        }
        fclose(fp);

        /* Vybudujeme si index ep nad polem hran,
           díky kterému budeme mít jednoduchý seznam sousedů. */
        for (int i = 1; i < N; i++) {
                vertices[i].start = vertices[i-1].start+vertices[i-1].deg;
                vertices[i-1].deg = 0;
        }
        vertices[N-1].deg = 0;
        for (int i = 0; i < M; i++) {
             ep[ vertices[edges[i].a].start + vertices[edges[i].a].deg++ ] = i;
             ep[ vertices[edges[i].b].start + vertices[edges[i].b].deg++ ] = i;
        }
}


/* Vrátí druhý konec hrany edge incidující s vrcholem vertex. */
inline int edge_end(int vertex, int edge)
{
        if (edges[edge].a != vertex)
                return edges[edge].a;
        return edges[edge].b;
}


/* Uloží výsledky do výstupního souboru. */
void print_out(void)
{
        /* Otevřeme výstupní soubor. */
        FILE *fout = fopen("asfalt.out", "w");
        if (!fout) {
                perror("Cannot open output file");
                exit(1);
        }

        /* Vytiskneme spárované hrany. */
        for (int i = 0; i < M; i++)
                if (edges[i].pair > i) {
                        int res[4] = { edges[i].a, edges[i].b,
                              edges[edges[i].pair].a, edges[edges[i].pair].b };

                        if ((res[0] == res[2]) || (res[0] == res[3]))
```

```
                                SWAP_INT(res[0], res[1])

                        if ((res[0] == res[3]) || (res[1] == res[3]))
                                SWAP_INT(res[2], res[3])

                        fprintf(fout, "%d %d %d\n", res[0]+1,res[1]+1,res[3]+1);
                }
        fclose(fout);
}

/* Prohledávání do hloubky na párování hran. */
void dfs(int actVertex, int parent)
{
        vertices[actVertex].visited = 1; // Označíme vrchol za navštívený.
/* Zavolame se na vsechny sousedni vrcholy, ve kterych jsme dosud nebyli */
        for (int i = vertices[actVertex].start;
                i < vertices[actVertex].start + vertices[actVertex].deg; i++)
                if (!vertices[edge_end(actVertex, ep[i])].visited)
                        dfs(edge_end(actVertex, ep[i]), actVertex);

        /* Nyní spárujeme zbylé hrany. */
        int pair = -1;
        for (int i = vertices[actVertex].start;
                i < vertices[actVertex].start + vertices[actVertex].deg; i++)
                if ((edges[ep[i]].pair == -1)
                        && edge_end(actVertex, ep[i]) != parent) {
                        if (pair == -1)  // Zatím nemame hranu ke spárování...
                                pair = ep[i];
                        else {   // Už na nás jedna nespárovaná hrana čeká
                                edges[ep[i]].pair = pair;
                                edges[pair].pair = ep[i];
                                pair = -1;
                        }
                }

        /* Pokud jedna hrana zbyla plonková - spárujeme ji s hranou k rodiči */
        if (pair != -1) {
                /* Bohužel hranu k otci musíme nejdřív najít... */
                int i = vertices[actVertex].start;
                while (edge_end(actVertex, ep[i]) != parent) i++;

                /* Index hrany k otci je teď uložen v i. */
                edges[ep[i]].pair = pair;
                edges[pair].pair = ep[i];
        }
}

int main(void)
{
        read_input();
        dfs(0, 0);
```

```
        print_out();
        return 0;
}
```

20-3-6 Hrady, hrádky, hradla — Cyril Hrubiš

V minulé části seriálu jste měli za úkol vymyslet obvod, který zjistí, zda-li je číslo na vstupu dělitelné třemi. Než začneme s vymýšlením obvodu, podíváme se na to, jaké zbytky po dělení třemi dávají číslice ve dvojkovém zápisu.

pozice číslice	0	1	2	3	...
hodnota desítkově	2^0	2^1	2^2	2^3	...
modulo třemi	1	2	1	2	...
zapsáno dvojkově	1	10	1	10	...

Vidíme, že se zbytek opakuje periodicky a pro liché pozice dostáváme zbytek $1_{10} = 01_2$ a pro sudé zbytek $2_{10} = 10_2$. Formálně lze toto pozorování dokázat indukcí, pro liché pozice dostáváme $n_0 = 2^0 = 1$, $n_k = 2^{(2k+1)}, n_{k+1} = 2^{(2k+3)}$, pak $n_{k+1} = 4 \cdot 2^{(2k+1)} = 3 \cdot 2^{(2k+1)} + 2^{(2k+1)}$ Vidíme tedy, že pro další číslici platí, že je součtem něčeho krát tři, což má jistě zbytek po dělení třemi nula, plus předchozí číslice, aplikováno „rekurzivně“ dostáváme, že všechny číslice na lichých pozicích mají stejný zbytek modulo třemi. Pro sudé pozice je důkaz podobný. A teď už dost formalismu a pojďme se podívat dál.

Vidíme, že když si budeme brát vstup po dvojicích, ty sečteme, pak bude toto číslo dělitelné třemi právě tehdy když bylo dělitelné třemi číslo původní. Což odpovídá tomu, že sečteme zbytky na lichých pozicích plus zbytky na sudých pozicích krát dva $a = a_0 + 2 \cdot a_1 + 4 \cdot a_2 + \ldots + 2^{n-1} \cdot a_{n-1} + 2^n \cdot a_n$, pak součet zbytků po dělení třemi je $S = a_0 + 2 \cdot a_1 + a_2 + 2 \cdot a_3 + \ldots + a_{n-1} + a_n = a_0, a_1 + a_2, a_3 + \ldots + a_{n-1}, a_n$, kde a, b znamená binární číslo poskládané z binárních číslic a a b, neboť ve dvojkovém zápisu je násobení dvěma posunutí doleva (obdobně jako násobení desíti v soustavě desítkové).

Teď nám už zbývá jenom vymyslet obvod, který sečte dvě dvoubitová čísla modulo třemi. Číslo 00 má stejný zbytek po dělení třemi jako 11. Sčítání je komutativní a proto nám nezáleží na pořadí sčítání, tato operace je tedy jednoznačně určena následující tabulkou. Značka „$\equiv$“ zde znamená, že čísla mají stejný zbytek po dělení třemi.

Vstup A	Vstup B	Výstup
01	01	10
10	10	01
01	10	$00 \equiv 11$
01	$00 \equiv 11$	01
10	$00 \equiv 11$	10
$00 \equiv 11$	$00 \equiv 11$	$00 \equiv 11$

Když se na tuto operaci pozorně podíváme, zjistíme, že se nápadně podobá obyčejnému sčítání, až na to, že se přenos znovu přičte k výsledku.

Takže máme obvod, který má na vstupu čtyři bity, dvě dvoubitová čísla a na výstupu dva bity, jedno dvoubitové číslo. Nyní stačí tyto obvody poskládat tak, že na každé výpočetní hladině zmenšíme počet dvoubitových zbytků na polovinu. Vstup jako obvykle doplníme dostatečným počtem nul. Číslo pak bude dělitelné třemi, když nám na konci zbyde 11 nebo 00. Jelikož obvod na sčítání má konstantní hloubku má celé zapojení asymptoticky logaritmickou složitost stejně jako obvod na počítání parity z předchozího seriálu.

Rozmyslete si, že pro dělitelnost třemi v zápise BCD bude fungovat stejný postup.

20-4-1 Druidí nápisy — Josef Pihera & Martin Mareš

Tato úloha vám pravděpodobně přivodila značné bolesti hlavy, a proto došla všeho všudy dvě řešení. My si tedy budeme chvíli lámat hlavu společně, přičemž se pokusíme vyhnout zmíněným intelektuálním bolestem.

První úskok, kterého se dopustíme, bude odkaz na řešení úlohy 20-2-3. Jistě jste si povšimli, že zadání jsou si velmi podobná a věřte, že tak je tomu i u řešení. Proto se před dalším čtením ujistěte, že znáte zadání i řešení 20-2-3 a porozuměli jste dané problematice.

V tento moment můžeme předpokládat existenci několika částí našeho algoritmu a stavět na nich. Pro jistotu si však ještě shrňme nejdůležitější body, z nichž vyjdeme: k užitku přijde, že umíme vstupní slovník reprezentovat *trií*, kde navíc umíme pro každou posloupnost značek (značkami budeme rozumět např. tečky a čárky) okamžitě vypsat, jaká *všechna* slova může posloupnost kódovat; dalším stavebním kamenem bude postup, jakým jsme hledali nejkratší slovní reprezentaci řetězce – ten budeme upravovat pro naše potřeby a proto se na něho podíváme důkladněji.

Jak tedy vypadalo pole, které jsme pomocí *dynamického programování* naplnili, a co nám vlastně sdělovalo za informaci? Pole nám na indexu i pro každou podposloupnost vstupu od i do n (tedy jinak řečeno pro každý sufix vstupu začínající indexem i) poví, jaké slovo si máme vybrat jako první, aby výsledný rozklad vstupu na slova byl co nejkratší. To nás v důsledku odkazuje na další pozici v tomto poli atd. ... což už vedlo k řešení.

Nyní si představme, že naše pole B bude „chytřejší“ a poví nám více informací. Řekněme, že nám ke každému sufixu vstupu poví pro každé j, jestli je takovýto sufix rozložitelný na právě j slov. Takové pole je dvojrozměrné - $B[i,j] = 1$ pokud je sufix začínající na indexu i rozložitelný na j slov, v opačném případě je $B[i,j] = 0$. Konstrukce není nikterak složitá. Pro sufix nulové délky - tedy podposloupnost, která začíná za koncem řetězce a zároveň tam i končí - víme, že je rozložitelný pouze na 0 slov ($B[n+1,0] = 1$ ale všechna

ostatní $B[n+1,1] = B[n+1,2] = \ldots = B[n+1,n] = 0$. Podobně jako v 20-2-3 budeme postupovat od nejkratšího sufixu až po nejdelší (celá posloupnost). Tedy pole B vyplňujeme od indexu n do indexu 1. Pro každý sufix snadno vyplníme hodnoty pro různá j, když známe všechna taková j kratších sufixů. Podrobněji prozkoumejme část, kdy jsme v druhé sérii vyplňovali položky pole na indexu i. Díky trii jsme nalézali jednotlivá slova, která mohou na daném indexu začínat a podle nich přepisovali údaj o nejkratším možném rozkladu. Zůstaňme u toho, že nacházíme jednotlivá slova od daného počátku i. Pak tedy víme, kde má začínat další slovo patřičného rozkladu (totiž hned za koncem prvního slova) – označme si takové místo indexem q ($q = i + d$, kde d je délka prvního slova a zřejmě tedy $q > i$). Protože postupujeme od nejkratších sufixů, tj. pole vyplňujeme od konce, tak hodnoty na pozici q již známe. Nyní nám stačí podívat se, na kolik slov umíme rozložit sufix vstupu začínající indexem q. Jestliže lze sufix začínající v q rozložit na j slov, potom sufix začínající i lze rozložit na $j+1$ slov. Tj. pokud $B[q,j] = 1$ pak nastavíme $B[i,j+1] = 1$. Zřejmě jen zkoušíme „dolepovat“ různá slova před začátek již existujících rozkladů – ačkoliv rozklady jako takové si nepamatujeme, pouze vedeme v patrnosti jejich existenci.

Tedy, právě jsme sestavili pole, podle kterého umíme říci, zda daný sufix lze rozložit na j slov. K čemu je nám to dobré? Umíme totiž nalézt všechny rozklady o daném počtu slov a vypsat je! Podívejme se na index 1, tedy na rozklad pro celý vstup. Vezměme si všechna j od nejmenšího k největšímu a uvažujme jen ta, pro která lze vstup rozložit na j slov (tedy $B[1,j] = 1$). Pro každé takové j můžeme zkusit s pomocí trie dohledat různá slova, kterými může rozklad začínat. Každé takové slovo někde končí a rozklad pokračuje na pozici q bezprostředně za ním následující. My se na tato q podíváme a pokud pro sufix začínající od q existuje rozklad na $j-1$ slov, pak si slovo, jenž nás posunulo na index q, můžeme do rozkladu vybrat. Zkráceně: Vyzkoušíme všechna slova, která mohou být na začátku, a podíváme se, jestli si je můžeme opravdu vybrat, tedy jestli za ně můžeme „dolepit“ rozklad pokračující $j-1$ slovy (jednička se zřejmě odečítá za již zvolené slovo). Jak dál? Jsme na indexu q a chceme rozložit sufix od q na $j-1$ slov. To už je ale ten samý problém, jako když jsme zkoušeli rozložit celý vstup na j slov. Pouze se liší místo, kde má rozklad začít, a počet slov, které má rozklad mít. Opět zkusíme všechna možná slova pro pokračování rozkladu a budeme se dívat, kam dál.

Jistě vás tedy napadá myšlenka použít rekurzi a de facto nasadit backtrack. My to opravdu uděláme, ale neděste se, ukážeme totiž, že to bude „slušný“ backtrack, který díky předpočteným informacím z části s dynamickým programováním (konstrukce dvojrozměrného pole) poběží rozumně rychle.

Backtrackující funkce potřebuje ke své činnosti znát počet již vypsaných rozkladů a stav právě zkoumaného rozkladu. Stav je charakterizován slovy, kte-

ré jsme si již vybrali pro začátek rozkladu, indexem i, na kterém končí poslední z těchto slov, a konečně počtem slov j, které ještě do úplného rozložení zbývá. Funkce pak vyzkouší všechna slova, která by mohla na zadaném indexu začínat, a pokud pro nějaké z nich zjistí, že lze rozložit zbytek vstupu na $j-1$ slov (zkontroluje $B[q, j-1] = 1$, kde q značí index těsně za kontrolovaným slovem), zavolá sama sebe. Tomuto synovskému volání přidá do rozkladu nalezené slovo a nový upravený stav – tedy index posunutý za konec takového slova a j snížené o 1. Pokud se po návratu ze synovského volání bude počet vypsaných rozkladů roven K, funkce svoji činnost ukončí. Tuto funkci spustíme postupně od nejkratšího na všechny rozklady začínající na indexu 1 – vyzkoušíme tak rozklady celého vstupu.

Podívejme se, jak dlouho trvá jedno volání našeho backtracku. Jsme na pevně zvoleném indexu a máme zadaný počet slov, na který máme vstup ještě dorozložit. Chceme přitom vypsat všechny možné rozklady dané délky. Projdeme až L indexů, kde může nějaké vybrané slovo končit (L označuje délku nejdelšího slova ve slovníku). Avšak pro každé slovo už vykonáme dotaz v konstantním čase do našeho zkonstruovaného pole. Pak už opět voláme sami sebe.

U backtracků obecně nám časovou složitost zhoršuje veliké množství větví výpočtu. Průběh výpočtu nějaké backtrackující funkce si totiž můžeme představovat jako strom, kde každý vrchol odpovídá jednotlivým voláním funkce, a hrany znázorňují vztah volající-volaný. Pokud se strom v každém vrcholu větví třeba jen dvakrát a hloubka stromu je řekněme 100, pak celková velikost stromu bude 2^{100}. V našem případě vyzkoušíme až L indexů. Ještě poznamenejme, že určitě $L \leq n$, protože kdyby tomu tak nebylo, museli bychom mít nějaké slovo delší než celý vstup. Takové slovo ale není k ničemu a můžeme ho zahodit. Mohlo by se zdát, že náš algoritmus poběží v čase $O(n^n)$, což se tváří dosti beznadějně.

Nyní si připomeňme, že nám stačí K nejkratších rozkladů, tedy můžeme backtrack po vypsání těchto rozkladů ukončit. Dalším důležitým postřehem je, že nikdy nevstoupíme do „slepých“ větví výpočtu – tedy nezkoušíme něco, co nevede k výsledku. To nám zajistilo právě dynamickým programováním zkonstruované pole B, kterého se tak v důsledku ptáme, zda máme vůbec nějaký rozklad zkoušet. Spojením těchto pozorování je fakt, že strom volání naší backtrackující funkce má právě K listů. Celková hloubka stromu je nejvýše n (připomeňme, že za n rozumíme délku vstupní posloupnosti)– úroveň v hloubce h odpovídá částečnému rozkladu s h slovy, ale protože každé slovo má alespoň jedno písmeno, nemůžeme nikdy ve stromu být hlouběji než v hloubce n. Tedy celkový počet volání backtrackující funkce je nejvýše $K \cdot n$. Pro každý list pak musíme ještě celý rozklad vypsat, což ale stíháme rovněž v $K \cdot n$. To už dává pro funkci rozumně vypadající časový odhad $O(K \cdot n^2)$, kde pro každé volání funkce zohledňujeme režii $O(n)$ na vyzkoušení slov.

Celková paměťová složitost programu je nejvíce zatížena použitým dvojrozměrným polem B, protože vše ostatní je lineárně velké vzhledem k délce vstupu, výsledkem je tedy $O(n^2)$. Časová složitost je ovlivněna konstrukcí trie v $O(P)$, kde P označme celkovou velikost slovníku. Dále počítáme ono dvojrozměrné pole B – ke každému z n indexů vyzkoušíme až n dalších. Pro každý z těchto n indexů vykonáme až n přepisování údajů (vyplňování hodnot rozkladů). Tedy dynamický výpočet je v $O(n^3)$. Backtrack potom v $O(K \cdot n^2)$. Celkem tedy $O(K \cdot n^2 + n^3)$.

Zrychlujeme ... Pozorní čtenáři si jistě všimli, že s časem poměrně plýtváme a jistě lze úlohu vyřešit lépe. Konkrétně budeme zrychlovat backtrack.

Podívejme se, zda nevykonáváme příliš mnoho kroků v každém z volání naší funkce. Zkoumáme totiž často až zbytečně mnoho indexů, jestli z nich nemůže náš rozklad pokračovat. Tedy ke každému i zkoušíme až n indexů. Kdyby se nám podařilo vyhnout se testování zbytečných indexů, znatelně si pomůžeme. Označme $q_1, q_2 \ldots, q_m$ všechny indexy, pro něž se náš výpočet větví a kterými pokračuje rozklad. Za zbytečné indexy pak budeme považovat ty, které leží za q_m. Na první pohled se zdá, že si pranic nepomůžeme, ale opak je pravdou. Pohleďme na problém šalamounsky a „naúčtujme“ procházení indexů až do q_m někomu jinému, v našem případě oním šťastlivcem bude výpis rozkladu.

Uveďme na příkladu: Ocitáme se právě uprostřed výpočtu. Řekněme, že náš rozklad má již h slov a nacházíme se na indexu 100. Ve slovníku je celkem W slov, ale pouze pro w z nich existují rozklady od aktuálního indexu. Délka nejdelšího z těchto w slov je řekněme 42. Délka nejdelšího slova ve slovníku je třeba 123. Pak za zbytečné indexy pokládáme všechny od 143 dál (až do $223 = 100 + 123$). Tedy $B[143, 0 \ldots m] \ldots B[223, 0 \ldots m]$ už nezkoušíme. Oněch prvních 42 znaků musíme vypsat tak jako tak, což znamená, že tyto kroky není nutné započítávat.

Pro každé z použitých $q_1, \ldots q_m$ budeme stejně muset vypsat celé slovo a proto můžeme s čistým svědomím všechny kroky potřebné k jejich určení započítávat do výpisu těchto slov (víceméně násobíme konstantou 2). Problematické jsou pouze zbytečné indexy, které nemáme komu naúčtovat, ale právě proto je vynecháme. Ve skutečnosti jsme tak schovali veškerou časovou režii naší funkce do výpisu, o kterém ale víme, že spotřebuje $O(K \cdot n)$.

Co k tomuto úskoku budeme potřebovat? Postačí nám, když budeme znát poslední index q_m a to pro každou dvojici (*index*, *počet slov rozkladu*) – to znamená navíc ke každému $B[i, j]$. Pro rozklady o různém počtu slov se totiž $q_1, \ldots q_m$ obecně liší. Během výpočtu našeho dvojrozměrného pole B si budeme počítat i nějaké další pomocné pole $Q[i, j] = q_m$. V backtracku postačí se dívat do tohoto pole a nezkoušet indexy za $Q[i, j]$.

Snížili jsme odhad časové složitosti funkce na $O(K \cdot n)$ a celkovým výsledkem je $O(K \cdot n + n^3)$.

Ještě poznámka na závěr: Náš algoritmus jsme se záměrně optimalizovali na paměťovou složitost nezávislou na K a to za cenu dynamického výpočtu v $O(n^3)$. Důvodem je možná velikost K. Představme si, že vstup bude pouze z teček a ve slovníku budou pouze dvě slova. Jejich zápisem bude jedna a dvě tečky. Počet možných rozkladů vstupu délky n pak bude F_n, kde F_n je n-té Fibonacciho číslo. Ty snadno odhadneme zdola na $2^{n/2}$, protože z $F_{n+2} = F_{n+1} + F_n$ plyne, že F_{n+2} je alespoň dvakrát větší než F_n. A exponenciální paměťovou složitost si opravdu dovolit nemůžeme.

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

#define MAX 256
#define INFTY 10000             //  Nekonečno

struct vrchol {                 //  Vrchol trie
    struct vrchol *syn[2];      //  syn[0] pro tečku, syn[1] pro čárku
    struct slovo *slova;        //  Seznam slov, která tu končí
    int hloubka;                //  Délka morseovkového zápisu (hloubka v trii)
};
struct vrchol koren;

struct slovo {                  //  Takhle si ukládáme seznamy slov
    struct slovo *dalsi;        //  Stačilo by si pamatovat jen jedno slovo,
    char s[1];                  //  ale třeba se to bude hodit v další úloze :)
};

//  Tabulka kódů:   A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T
char morse[26] = { 6,17,21, 9, 2,20,11,16, 4,30,13,18, 7, 5,15,22,27,10, 8, 3,
//  U  V  W  X  Y  Z
  12,24,14,25,29,19 };

void preloz(char *co, char *kam)              //  Přeloží slovo do morseovky
{
    while (*co) {
        int k = morse[*co++ - 'a'];           //  Ďábelský kód písmenka
        while (k > 1) {                       //  Postupně rozkládáme na značky
            *kam++ = ".-"[k%2];
            k /= 2;
        }
    }
    *kam = 0;
}

void nacti_slovnik(void)
{
    char slovo[MAX], mslovo[MAX];
    while (fgets(slovo, sizeof(slovo), stdin) && slovo[0] != '\n') {
        int len = strlen(slovo);
```

```
        slovo[len-1] = 0;                       //  Smažeme konec řádku
        preloz(slovo, mslovo);                  //  Přeložíme do morseovky
        struct vrchol *v = &koren;              //  Přidáváme do trie
        for (char *c=mslovo; *c; c++) {
            int z = (*c == '-');                //  Aktuální značka
            if (!v->syn[z]) {   //  Kam dál? Není-li kam, založíme nový vrchol
                v->syn[z] = (vrchol*) malloc(sizeof(struct vrchol));
                memset(v->syn[z], 0, sizeof(struct vrchol));
                v->syn[z]->hloubka = v->hloubka + 1;
            }
            v = v->syn[z];                      //  Vydáme se tím směrem
        }
        struct slovo *s = (struct slovo*) malloc(sizeof(struct slovo) + len);
                            //  Stojíme ve vrcholu, který odpovídá konci slova,
        s->dalsi = v->slova;    //  tak tam slovo přidáme
        v->slova = s;
        strcpy(s->s, slovo);
    }
}

int B[MAX+1][MAX+1]; /* B[i][j]==1 pokud lze pokrýt [i...N-1] přesně j slovy */

// Zde pomocí dynamického programování naplníme B
void SpoctiB(char *vstup, int N)
{
        B[N][0] = 1;
        for (int i=N-1; i>=0; i--) // Postupujeme od konce vstupu
        {
                struct vrchol *v = &koren; // procházíme trii postupně od kořene
                int j = i;
                while (v && j < N)
                {
                        v = v->syn[vstup[j++] == '-'];
                        if (v && v->slova)  // ano, nějaké slovo zde končí
                                for (int k=0; k<N; k++)
                                        if (B[j][k])
                                        {// Od indexu j umíme rozložit k slovy
                                                B[i][k+1] = 1;
                                        }// zde (od i) umíme rozložit k+1 slovy
                }
        }
}

int K; // počet rozkladů, které máme vypsat
int K_hotovo; // kolik rozkladů jsme již vypsali

/* vypiš rozklad od i...N-1 s j slovy, kde v *rozklad jsou již uložena slova
   z rozkladu od 1..i-1 a *p udává, kde v řetězci *rozklad máme pokračovat */
void PisReseni(char *rozklad, char *p, char *vstup, int i, int N, int j)
{
        if (i >= N) // Rozklad této větve výpočtu je již dokončen
```

```
        {
                *p = 0;
                puts(rozklad);
                K_hotovo++;
                return;
        }

        struct vrchol *v = &koren;
        *p++ = ' ';
        while (v && i < N)
        {
                v = v->syn[vstup[i++] == '-'];
                if (v && v->slova && B[i][j-1]) // končí-li zde  nějaké slovo
                        for (struct slovo *slovo = v->slova; slovo;
                                slovo=slovo->dalsi)
                        {                   // pak projdeme všechna taková slova
                                int delka = strlen(slovo->s);
                                // dopíšeme si je k částečnému rozkladu
                                memcpy(p, slovo->s, delka);
                                // a rekurzivně pokračujeme
                                PisReseni(rozklad, p+delka, vstup, i, N, j-1);
                                if(K_hotovo == K) // máme-li jich dost, skončíme
                                        return;
                        }
        }
}

int main(void)
{
    char z[MAX];                //  Vstupní řetězec
    int n;                      //  Jeho délka
    struct vrchol *s[MAX+1];

    nacti_slovnik();            //  Přečteme slovník a vstup
    fgets(z, MAX, stdin);       //  Pozor, indexujeme od 1
    n = strlen(z)-1;

        scanf("%d", &K);    // Načteme, kolik rozkladů se po nás chce
        K_hotovo = 0;

        SpoctiB(z, n);
        char rozklad[MAX];
        for(int j=0; j<=n && K_hotovo<K; j++)  // ptáme se na všechny rozklady
        {                           // celého vstupu, od nejkratších počínaje
                if(B[0][j])
                {
                        PisReseni(rozklad, rozklad, z, 0, n, j);
                } //  a každý nalezený vypíšeme (resp. všechny rozklady délky j)
        }
```

```
    if (K_hotovo < K)           //  Pokud žádané řešení není, dáme o tom vědět
        {
                if(K_hotovo)
                        printf("Existuje jen %d rozkladů.\n", K_hotovo);
                else
                        puts("Řešení neexistuje");
        }
                return 0;
}
```

20-4-2 Stonehedge — Michal „vorner“ Vaner

Napřed uděláme několik pozorování.

Z každého rohu vedou právě dvě úsečky. Méně nedává smysl a více dává křížení.

Jedna z těchto úseček bude vodorovná a jedna svislá (jinak by to nebyl roh, ale průchod bodem). Tedy, každý vrchol má svého vodorovného a svislého souseda.

Nyní si vezměme například vodorovné sousedy (pro svislé to bude obdobné). Vodorovný soused má stejnou y-ovou souřadnici.

Rozdělme si tedy všechny body do skupin podle y. Podívejme se na jednu takovou skupinku a setřiďme si ji, řekněme, odleva. Ten úplně vlevo musí mít svého souseda (a to v této skupince), ale jediný, který připadá v úvahu je ten nejbližší napravo od něj. Takže je spojíme. Tím jej samozřejmě použijeme (může mít jen jednoho souseda) a stejná situace tedy nastává se třetím a čtvrtým a tak dále.

Všimněme si, že v každé skupince musí být sudý počet vrcholů. Kdyby nebylo, tak to zřejmě není kámen popsaný v zadání (poslední nemá s čím sousedit).

Takto můžeme zpracovat všechny skupinky. A obdobným způsobem můžeme spárovat i svislé dvojice.

Jak to ale udělat rychle? Pokud si setřídíme body lexikograficky podle (y, x), pak budou vždy všechny body se stejnou y-ovou souřadnicí za sebou, seřazené dle x-ové. A protože mají skupinky sudé počty, můžeme takto setříděnou posloupnost prostě začít spojovat po dvojicích od začátku (a nestarat se, kde začíná jedna skupinka a druhá končí).

Nyní již máme ke každému bodu jeho svislého a vodorovného souseda (svislé sousedy uděláme stejně, jen setřídíme dle (x, y)). Musí tvořit uzavřený cyklus (neboť v kameni nejsou díry a všechny body musí být na kameni a obvod je jistě souvislý). Takže jej stačí jen vypsat a jediný problém je, kterým směrem začít.

Pokud si vybereme některý vrchol na „horní vrstvě“ a ten má svého pravého souseda, pak je to zajisté po směru hodinových ručiček. A nejlevější vrchol svého

pravého souseda bude mít určitě. Mimochodem, tento vrchol skončil jako první při setřídění dle (y, x).

Jak rychle to dokáže běžet? Načtení zvládneme lineárně, výpis také (to jen procházíme kolem dokola, než narazíme opět na ten první). Rozdělení na dvojice jde také lineárně – každý bod zapojíme jednou vodorovně a jednou svisle. Jediný problém je tedy s tříděním, které (v obecném případě) nezvládneme rychleji, než v $O(N \cdot \log N)$.

Paměťová složitost je lineární, stačí si pamatovat jednotlivé body a jejich sousedy.

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

#define VODOROVNE 0
#define SVISLE 1
#define X 0
#define Y 1

typedef struct bod_t {
        int souradnice[ 2 ];
        struct bod_t *sousedi[ 2 ];
} bod_t;

int co[ 2 ] = { X, Y }; // Kterým směrem se porovnává

int porovnej( const void *_1, const void *_2 ) {// Lexikografické porovnání
        bod_t * const *a = _1, * const *b = _2;
        if( (*a)->souradnice[ co[ 0 ] ] == (*b)->souradnice[ co[ 0 ] ] )
                return (*a)->souradnice[ co[1] ] - (*b)->souradnice[ co[ 1 ] ];
        else
                return (*a)->souradnice[ co[0] ] - (*b)->souradnice[ co[ 0 ] ];
}

bod_t *sparuj( int n, bod_t *body ) {// Utvoří dvojice podle nastavení v co
        bod_t *pozice[ n ];
        for( int i = 0; i < n; ++ i )
                pozice[ i ] = &body[ i ];
        qsort( pozice, n, sizeof *pozice, porovnej );
        for( int i = 1; i < n; i += 2 ) {
                pozice[ i ]->sousedi[ co[ 1 ] ] = pozice[ i - 1 ];
                pozice[ i - 1 ]->sousedi[ co[ 1 ] ] = pozice[ i ];
        }
        return pozice[ 0 ];
}

int main( void ) {
        int n;
        scanf( "%d", &n );
        bod_t body[ n ];
```

```
        for( int i = 0; i < n; ++ i ) {
        scanf( "%d%d", &body[ i ].souradnice[ X ], &body[ i ].souradnice[ Y ]);
        }
        sparuj( n, body );// Svisle
        co[ 0 ] = Y;
        co[ 1 ] = X;
        bod_t *akt = sparuj( n, body ), *start = akt;// Vodorovně a ulož první
        int smer = VODOROVNE;
        do {// Obejdi
                printf( "%d %d\n", akt->souradnice[ X ], akt->souradnice[ Y ] );
                akt = akt->sousedi[ smer ];
                smer = ( smer + 1 ) % 2;
        } while( akt != start );
        return 0;
}
```

20-4-3 Mince — Pavel Čížek

Do zadání se nám vloudila jedna drobná nejednoznačnost. Není zřejmé, zda v případě, kdy jsou na počátku 4 mince lícem vzhůru trpaslík hned ukončí hru, či ne. Pokud tomu tak nebude, přidáme na začátek „tah“, kdy neotočíme žádnou minci, a vše převedeme na první případ.

Nejprve si uvědomme, že všechny možnosti, jak jsou mince otočeny (nazveme toto stavem mincí), je možno rozdělit na šest skupin (matematicky se mluví o kongruenci), které se při otočení nemění (tj. pokud vezmeme stav z nějaké skupiny a otočíme soudek, bude stav patřit do té samé skupiny). Popišme je vzájemnou polohou a počtem mincí, které jsou lícem vzhůru. To mohou být všechny (a tedy končíme), tři mince, dvě mince úhlopříčně, dvě mince vedle sebe, jedna mince, nebo žádná. Pokud tedy zjistíme, kterou skupinu převádí daný tah (tj. otočení mincí) na kterou, můžeme najít jejich posloupnosti, které každou z pozic převedou na první z nich a tím jsme hotovi.

Jak na to? Pro zjednodušení uvažujme, že každý lichý tah otočíme na soudku všechny mince a navrhujeme jen sudé. Ten nám převádí skupinu se čtyřmi a žádnou mincí lícem nahoru mezi sebou. Stejně tak mezi sebou přechází trojice, resp. samotná mince. Zbylých dvou skupin, odpovídajících dvěma mincím lícem vzhůru, se tento tah nedotkne. Slučme tedy příslušně skupiny mezi sebou a označme si je jako (4) v případě, že jsou všechny mince stejnou stranou vzhůru, (2U) když jsou dvě mince úhlopříčně lícem vzhůru a dvě ne, (2V) případ, kdy jsou dvě mince vedle sebe lícem vzhůru a dvě ne a (1) stavy, je jedna mince nějak otočena a zbylé tři opačně.

Další užitečné pozorování je, že při otočení sudého počtu mincí se nezmění parita počtu mincí lícem vzhůru (tj. pokud byl lícem vzhůru lichý počet mincí, zůstane lichý, a pokud sudý zůstane sudý). Tedy otočením sudého počtu mincí převedeme stav ze skupiny (1) na stav ze skupiny (1) a stavy skupin

((4),(2U),(2V)) na stavy z té samé trojice skupin. Pokud otočíme jednu (nebo libovolný lichý počet mincí), převede to (1) na stav z některé ze skupin ((4),(2U),(2V)) (do které skupiny přesně se dostaneme záleží na otočení soudku a kterou přesně minci bereme) a stavy ze skupin ((4),(2U),(2V)) na stavy skupiny (1). Pokusme se tedy najít posloupnost otáčení sudého počtu mincí, kterou jsme schopni dostat stavy ze „sudých" skupin ((4),(2U),(2V)) dostat na (4). Pokud nám nevyřeší problém, tak musel stav patřit (a stále patří) do skupiny (1). Otočením jedné mince ho převedeme na nějaký stav ze „sudých" skupin a, jelikož už víme, že stav ze skupiny (1) nemůžeme mít, stejnou posloupností tahů trpaslíka porazíme.

Zbývá tedy podívat se na to, jak vyřešit skupiny (4), (2U) a (2V). Uvažujme pro tento odstavec, že víme, že lícem nahoru byl sudý počet mincí. (4) ani řešit nemusíme - tu nám zastaví trpaslík. (2U) je jednoduché - otočením dvojice mincí úhlopříčně na soudku dostaneme (4) (a tedy trpaslík nás zastaví) a z (2V) se nám stane (2V). Po „prvním" tahu, pokud nás trpaslík nezastavil, víme, že stav soudku je ze skupiny (2V). Jako „druhý" tah otočíme dvojici mincí vedle sebe. Podle toho, jak byl zrovna natočen soudek, dostaneme stav ze skupiny (4) nebo (2U). Pokud nás stále trpaslík nezastavil, patří do skupiny (2U) a tedy po otočení dvojice mincí úhlopříčně budou všechny rubem (nebo lícem) vzhůru.

Tím jsme tedy nastínili, jak získat vyhrávající strategii. Konkrétně bude vypadat:

```
otoč všechny 4 mince
otoč 2 mince úhlopříčně
otoč všechny 4 mince
otoč 2 mince vedle sebe
otoč všechny 4 mince
otoč 2 mince úhlopříčně
otoč všechny 4 mince
otoč jednu minci
otoč všechny 4 mince
otoč 2 mince úhlopříčně
otoč všechny 4 mince
otoč 2 mince vedle sebe
otoč všechny 4 mince
otoč 2 mince úhlopříčně
otoč všechny 4 mince
```

Malá poznámka na závěr. Rychlejší strategie, než zde uvedené ani neexistuje. Pro začátek uvažujme, že máme jen zavázané oči (a nevíme, jaké je počáteční otočení mincí), ale trpaslík netočí se soudkem. Uvažujme všechny přípustné stavu soudku (tj. takové, kdy nám trpaslík ještě nezastavil hru). Každým tahem můžeme maximálně jednu z těchto pozic převést na tu, kdy jsou všechny lícem vzhůru a tím jí vyloučit (buď nás trpaslík zastaví, nebo to tenhle stav mincí nebyl). Zbylé pozice se nám vzájemně jednoznačně převedou na (možná) jiné stavy. Vzájemně jednoznačně znamená, že po provedení tahu budeme schopni

určit jaká byla původní pozice (např. provedením toho samého tahu podruhé se vrátíme zpět) a tedy i že počet různých přípustných pozic se nám nezmenší, resp. zmenší o jednu, kterou jsme vyloučili. Jelikož na začátku máme jeden z 15-ti stavů a nevíme který, potřebujeme alespoň 15 tahů abychom libovolný z nich otočili lícem vzhůru. Pokud trpaslík „začne“ točit soudkem, určitě strategie řešící tuhle úlohu nebude kratší než v případě, kdy neotáčí. A protože nalezená strategie 15 tahů má, musí být nejkratší možná.

20-4-4 Skupinky pro chytré — Pavel Machek

Operace nad skupinkami přesně odpovídají operacím nad haldou z kuchařky 4. série, a naše rešení bude opravdu vycházet z haldy. Protože nechceme hledat jedince s minimálním IQ (těch je dost :-), ale s maximálním, bude zapotřebí otočit porovnávání.

Větší rozdíl spočívá v tom, že operace potřebujeme provádět „nedestruktivně“ – nesmíme měnit již existující skupinky. Na principu fungování haldy se nic nemění, ale data nebudeme moci ukládat do pole jako v kuchařce. Strom uložíme jako sadu prvků (struct Node) pospojovaných pointery na levého a pravého syna. Operace nad haldou tak bude jednoduché dělat nedestruktivně: místo modifikace prvku naalokujeme nový prvek, zkopírujeme hodnoty ze starého prvku, a upravíme co je potřeba upravit. Při modifikaci syna budeme muset vždy vyrobit i nového otce a dál až ke kořeni.

Pro haldové operace potřebujeme efektivně umět pracovat s nejpravějším prvkem ve spodní hladině, a potřebujeme umět určit otce daného prvku. V poli je situace jednoduchá, požadovaný prvek je v poli tolikátý, kolik je prvků v haldě, a otec je na pozici $i/2$ (kde i je pozice syna).

Jednoduché řešení by bylo přidat do každého prvku ukazatel na jeho otce, a držet si ukazatel na nejpravější prvek ve spodní hladině; ale toto řešení použít nemůžeme, protože ukazatel na otce by nám neumožnil pracovat „nedestruktivně“.

Naštěstí je možné i-tý prvek najít pomocí bitového zápisu i, stačí postupovat od kořene a dle hodnoty bitu jít do levého nebo pravého podstromu. Pokud si do pomocného pole schováme prvky, které jsme prošli, odpadne též problém s hledáním předchůdců.

Časová složitost operací `insert` a `delete_best` je $O(\log(n))$, časová složitost `find_best` je $O(1)$. Operace `find_best` nepotřebuje žádnou dodatečnou paměť, `insert` i `delete_best` naalokují $O(\log(n))$.

(S díky Petru Ondrúškovi.)

```
#include <cstdio>
#include <cstring>

#define MAX_NUM_QUEUES 1000
```

```
// Struktura osoby. Alokujeme ji jen jednu, ostatni jsou odkazy.

struct Person {
        int IQ;
        char *Name;
        Person (int newIQ, char *newName) {
                IQ = newIQ;
                Name = strdup(newName);
        }
};

#define SWAP(A, B) do {  \
        Person *Tmp = A; \
        A = B;           \
        B = Tmp;         \
} while (0)

// struktura jednoho vrcholu v halde
struct Node {
        Node *Left, *Right;
        Person *P;
};


struct Queue {
        Node Root;
        int Size;
} Queue[MAX_NUM_QUEUES];

int Queues = 2;

// najde nejvetsi prvek - ten je vzdy ve vrcholu haldy
char *
find_best(int ID)
{
        return Queue[ID].Root.P->Name;
}

// odebere koren z haldy
int
delete_best (int ID)
{
        Queue[Queues].Size = Queue[ID].Size - 1;
        Node *Root;
        Node *A = &Queue[ID].Root;
        Node *B = &Queue[Queues].Root;
        int Pos = Queue[ID].Size;
        int log = 0;
        while (Pos >> log)
                log++;
```

```
/* Pujdeme po strome dolu, smerem k poslednimu prvku (Pos), a
   budeme prvky posunovat nahoru */

*B = *A;
for (int i = log - 2; i > 0; i--) {
        if ((Pos >> i) & 1) {
                puts ("right");
                B->Right = new Node;
                A = A->Right;
                B = B->Right;
        } else {
                puts ("left");
                B->Left = new Node;
                A = A->Left;
                B = B->Left;
        }
        *B = *A;
}
Root = &Queue[Queues].Root;

/* presunout nejpravejsi prvek v dolni hladine do vrcholu */

if (Pos & 1) {
        Root->P = B->Right->P;
        B->Right = NULL;
} else {
        Root->P = B->Left->P;
        B->Left = NULL;
}

/* bublat dolu a zabezpecit invariant IQ rodice > IQ synu */

A = Root;
while (1) {
        Node *Left = A->Left;
        Node *Right = A->Right;
        if (Right != NULL && Right->P->IQ > Left->P->IQ) {
                if (Right->P->IQ > A->P->IQ) {
                        A->Right = new Node;
                        *(A->Right) = *Right;
                        SWAP(A->P, A->Right->P);
                        A = A->Right;
                } else
                        break;
        }
        else if (Left != NULL) {
                if (Left->P->IQ > A->P->IQ) {
                        A->Left = new Node;
                        *(A->Left) = *Left;
                        SWAP(A->P, A->Left->P);
```

```
                                A = A->Left;
                        }
                        else
                                break;
                }
                else
                        break;
        }
        return Queues++;
}

// vlozi prvek do haldy
int
insert (int ID, int IQ, char *Name)
{
        Queue[Queues].Size = Queue[ID].Size + 1;
        Node *A = &Queue[ID].Root;
        Node *B = &Queue[Queues].Root;
        int Pos = Queue[Queues].Size;
        int log = 0;
        while (Pos >> log)
                log++;

        /* odkazy na vrcholy na ceste k nejpravejsimu vrcholu ve
           spodni vrstve */

        Node *Path[log];
        Path[log - 1] = B;

        /* Sejdeme po strome smerem dolu k nejpravejsimu vrcholu */

        for (int i = log - 2; i >= 0; i--) {
                *B = *A;
                if ((Pos >> i) & 1) {
                        B->Right = new Node;
                        if (i)
                                A = A->Right;
                        B = B->Right;
                } else {
                        B->Left = new Node;
                        if (i)
                                A = A->Left;
                        B = B->Left;
                }
                Path[i] = B;
        }

        // pridat novy vrchol

        B->Left = B->Right = NULL;
        B->P = new Person (IQ, Name);
```

```
        // vybublat nahoru a zabezpecit rovnovahu

        for (int i = 0; i < log - 1; i++)
                if (Path[i]->P->IQ > Path[i + 1]->P->IQ)
                        SWAP(Path[i]->P, Path[i + 1]->P);
        return Queues++;
}

int
main(void)
{
        Queue[1].Size = 1;
        Queue[1].Root.P = new Person (0, "UNDERFLOW");
        insert(1, 130, "Ales");
        insert(2, 110, "Petr");
        insert(1, 140, "Jana");
        printf("%s\n", find_best(2));
        printf("%s\n", find_best(3));
        printf("%s\n", find_best(4));
        delete_best(3);
        printf("%s\n", find_best(5));
        return 0;
}
```

20-4-5 Roboti na útěku — Martin „Bobřík“ Kruliš

Řešitele této úložky lze rozdělit do dvou skupin. Skupina první (která měla mohutnost pouze 3) přišla, viděla a s přehledem vyřešila. Skupina druhá (značně početnější) přišla, viděla, nevěřila svým očím, a tak raději vůbec neřešila. Pojďme se nyní podívat, jak si s roboty poradit ...

K řešení našeho problému použijeme procházení stavového prostoru do šířky. U všech procházení je nejtěžší poznat, jak vypadá zmíněný stavový prostor. Obecně je stavový prostor orientovaný graf, kde vrcholy představují jednotlivé stavy a hrany přechody mezi nimi (jednotlivé kroky). V našem případě je stav definován polohou obou robotů a všech stráží. Z každého stavu pak vedou nejvýše čtyři hrany – jedna pro každý potencionální příkaz, který mohou roboti obdržet (stráže se pohybují automaticky, takže není potřeba jejich pohyb dále řešit). Některé hrany (případně i stavy) mohou být z prostoru vyloučeny, protože v nich dojde k zajmutí některého z robotů.

Bohužel nemůžeme stavy reprezentovat přímočaře, jak bylo popsáno. Pejsek je zakopaný v počtu stráží. Kdyby jich v obou bludištích byl maximální počet (tedy 10), potřebovali bychom vhodně ukládat informace o $(x_1 y_1 x_2 y_2)^{11}$ stavech. Naštěstí víme, že stráže neustále chodí po stejných trasách a délka jejich tras může být pouze 2, 3 nebo 4. Podle toho se pozice stráže opakuje vždy po 2, 4 nebo 6 krocích. Nejmenší společný násobek těchto čísel je 12, takže

každých 12 kroků se pozice všech stráží opakuje. Pokud víme, ve které z 12-ti možných pozic se právě stráže nachází, jsme schopni spočítat její polohu jen ze vstupních dat.

Jak již bylo naznačeno, prostor budeme prohledávat do šířky. Pro každý stav si potřebujeme pamatovat informaci, zda jsme v něm už byli, a také z jakého stavu jsme se do něho dostali, abychom pak mohli zrekonstruovat výslednou cestu. Maximální počet stavů bude $12 \times (20 \times 20)^2$, protože existuje 12 možných pozic stráží a maximální rozměry bludiště jsou 20×20, celkem tedy 2560000 stavů. Stav dokážeme bez problému zakódovat do 32-bitového integeru, takže celkem spotřebujeme necelých 10 MB paměti na stavy a nejvýš tři čtvrtiny této hodnoty na frontu. S těmito čísly už se do CodExu vejdeme.

Samotný algoritmus pak přesně odpovídá tomu, co již známe z kuchařky. Na počátku vložíme do fronty výchozí stav a označíme jej za použitý. V každém kroku vybereme jeden stav z fronty, spočítáme všechny stavy, do kterých se z něj dá dostat, ty si označíme a vložíme do fronty. Algoritmus končí v okamžiku, kdy narazíme na stav, ve kterém jsou oba roboti venku z bludiště.

Výslednou cestu pak zrekonstruujeme tak, že postupujeme od koncového stavu zpět k výchozímu podle informací, které jsme si ke stavům uložili. Jediný zádrhel spočívá v tom, že cestu musíme vypsat v opačném pořadí, takže si ji musíme nejprve uložit a pak teprve vypsat.

Voilà. Nyní už jen dochutíme podle libosti a servírujeme kompilátoru.

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

#define MAX_ROWS                20
#define MAX_COLS                20
#define MAX_PATROL_STATES       12
#define MAX_MAZE_SIZE           (MAX_ROWS * MAX_COLS)
#define MAX_CONFIGS     (MAX_PATROL_STATES*(MAX_MAZE_SIZE+1)*(MAX_MAZE_SIZE+1))

const char *moves = "NSEW";

struct scoords {                // Koordináty robota v bludišti.
        char row, col;
};

// Konfigurace robotů a stráží.
struct sconfig {
        unsigned char ticker;   // Tikač určuje, v jaké fázi se nacházejí
                                // stráže (počítá modulo 12).
        char move;              // Tah (N, S, E nebo W), kterým jsme se
                                // do této konfigurace dostali.
        // X je počáteční konfigurace a \0 je zatím neprozkoumaná konfigurace.
        struct scoords robot[2];
};
```

```
// Struktura zapouzdřující frontu konfigurací.
struct sfifo {
        struct sconfig data[MAX_CONFIGS];
        int first, last;
};

// Struktura zapouzdřující jednu stráž.
struct sguard {
        struct scoords start;   // Políčko, na kterém stráž začíná svou hlídku.
        unsigned char len;      // Délka hlídkovací trasy.
        char direction; // Směr, kterým se stráž na začátku dívá (N, S, E, W).
};

/* Stavový prostor všech možných konfigurací. Každá konfigurace ukazuje na
 * předchozí tak, jak byly procházeny v BFS. Dosud nenavštívené konfigurace
 * mají nastavený move na \0. */
struct sconfig configs[MAX_PATROL_STATES][MAX_MAZE_SIZE+1][MAX_MAZE_SIZE+1];

// Jednotlivá bludiště (hodnota 0 = volné pole, 1 = zeď).
char mazes[2][MAX_ROWS][MAX_COLS];
int rows[2], cols[2];

// Seznam stráží.
struct sguard guards[2][10];
int guardsCount[2];

// Fronta konfigurací.
struct sfifo fifo;

// Pole pro výsledky.
char results[MAX_CONFIGS];
int resultsCount = 0;

/*
 * Souřadnice a směry.
 */

// Převede znak reprezentující světovou stranu na relativní koordináty.
struct scoords getDirectionCoords(char direction) {
        struct scoords res;
        res.row = res.col = 0;
        switch(direction) {
                case 'N':       res.row = -1;   break;
                case 'S':       res.row = 1;    break;
                case 'E':       res.col = 1;    break;
                case 'W':       res.col = -1;   break;
        }
        return res;
}
```

```
// Otestuje, zda je robot venku z bludiště.
int isRobotOut(struct scoords robot) {
        return (robot.row < 0) ? 1 : 0;
}

// Vrací true, pokud jsou si koordináty rovny. Jinak vrací false.
int isEqual(struct scoords coords1, struct scoords coords2) {
        return (coords1.row == coords2.row) && (coords1.col == coords2.col);
}

// Posune robota daným směrem (pokud je to možné).
void moveRobot(struct scoords *robot, struct scoords direction, int mazeIdx) {
        if (isRobotOut(*robot)) return;
        struct scoords newPos = *robot;
        newPos.row += direction.row;
        newPos.col += direction.col;
        if ((newPos.row < 0) || (newPos.col < 0) ||
                (newPos.row >= rows[mazeIdx]) || (newPos.col >= cols[mazeIdx]))
                robot->row = -1;
        else if (mazes[mazeIdx][(unsigned)newPos.row][(unsigned)newPos.col]==0)
                *robot = newPos;
}
 /* Práce se stavovým prostorem konfigurací. */
// Zakóduje pozici robota do integeru. Robot může být mimo bludiště.
int encodeCoords(struct scoords coords) {
        int res = coords.row * MAX_COLS + coords.col;
        if (coords.row < 0) res=MAX_MAZE_SIZE; // Pokud je robot mimo bludiště.
        return res;
}
// Označí danou konfiguraci za zpracovanou.
void markConfigVisited(struct sconfig C, struct sconfig Prev, char move) {
        struct sconfig *config = &configs[C.ticker][ encodeCoords(
                C.robot[0]) ][ encodeCoords(C.robot[1]) ];
        *config = Prev;
        config->move = move;
}

// Podívá se na stav dané konfigurace.
unsigned char isConfigVisited(struct sconfig C) {
        return (configs[C.ticker][ encodeCoords(C.robot[0]) ][ encodeCoords(
                C.robot[1]) ].move != '\0');
}

// Nalezne následující konfiguraci když se roboti hýbou daným směrem.
struct sconfig getNextConfig(struct sconfig config, char move) {
        struct scoords direction = getDirectionCoords(move);
        struct sconfig res = config;
        res.ticker = (res.ticker + 1) % 12;
        for(int i = 0; i < 2; i++)
                moveRobot(&res.robot[i], direction, i);
        return res;
```

```
}
 /* Práce s FIFO. */
// Inicializuje frontu.
void initFifo(struct sfifo *F) {
        F->last = -1;
        F->first = 0;
}

// Pokud je fronta prázdná, vrací true, jinak false.
int isFifoEmpty(struct sfifo *F) {
        return (F->last < F->first);
}

// Vloží prvek C do fifo.
void pushFifo(struct sfifo *F, struct sconfig C) {
        F->data[ ++F->last ] = C;
}

// Vyjme prvek z fronty a uloží jej do C. Pokud se to povedlo, vrací true.
int popFifo(struct sfifo *F, struct sconfig *C) {
        if (isFifoEmpty(F)) return 0;
        *C = F->data[ F->first++ ];
        return 1;
}
 /* Stráže. */
// Vypočítá pozici stráže podle časovače.
struct scoords getGuardPos(struct sguard guard, unsigned char ticker) {
        ticker %= (guard.len - 1) * 2;
        struct scoords res = guard.start;
        struct scoords direction = getDirectionCoords(guard.direction);
        int len = (guard.len-1) - abs(ticker - guard.len + 1);
        res.row += direction.row * len;
        res.col += direction.col * len;
        return res;
}

// Došlo při přesunu do nové konfigurace k polapení některého z robotů?
int isCaptured(struct sconfig config, struct sconfig newConfig) {
        struct scoords pos1, pos2;
        for(int i = 0; i < 2; i++)
                if (!isRobotOut(config.robot[i]))
                        for(int g = 0; g < guardsCount[i]; g++) {
                        pos1 = getGuardPos(guards[i][g], config.ticker);
                        pos2 = getGuardPos(guards[i][g], newConfig.ticker);
                        if (isEqual(pos2, newConfig.robot[i]) ||
                                        (isEqual(pos1, newConfig.robot[i])
                                        && isEqual(pos2, config.robot[i])))
                                return 1;
                        }
        return 0;
}
```

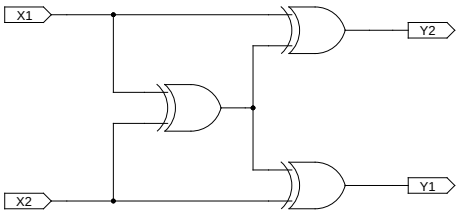
```
 /* Hlavní procedury.*/
// Načte z daného souboru jedno bludiště a jeho sttáže.
void loadMaze(FILE *fp, struct scoords *robot, int mazeIdx) {

        // Načteme rozměry.
        fscanf(fp, "%d %d\n", &rows[mazeIdx], &cols[mazeIdx]);

        // Načteme mapu bludiště.
        char ch;
        for(int i = 0; i < rows[mazeIdx]; i++) {
                for(int j = 0; j < cols[mazeIdx]; j++) {
                        fscanf(fp, "%c", &ch);
                        switch(ch) {
                                case '#':       // Nastavíme pole na stěnu.
                                        mazes[mazeIdx][i][j] = 1;
                                        break;

                                case 'X':       // Označíme počáteční pozici.
                                        robot->row = i;
                                        robot->col = j;
                                // A tady propadneme do následujícího case...
                                case '.':       // Nastavíme pole jako volné.
                                        mazes[mazeIdx][i][j] = 0;
                                        break;
                        }
                }
                fscanf(fp, "\n");
        }

        // Načteme stráže.
        fscanf(fp, "%d\n", &guardsCount[mazeIdx]);
        for(int i = 0; i < guardsCount[mazeIdx]; i++) {
                int row, col, len;
                fscanf(fp, "%d %d %d %c\n", &row, &col, &len,
                        &guards[mazeIdx][i].direction);
                guards[mazeIdx][i].start.row = row - 1;
                guards[mazeIdx][i].start.col = col - 1;
                guards[mazeIdx][i].len = len;
        }
}

// Načteme data ze vstupního souboru.
void load(void) {
        FILE *fp = fopen("robots.in", "r");
        if (!fp) exit(1);

        struct sconfig config;
        config.ticker = 0;
        for(int i = 0; i < 2; i++)
                loadMaze(fp, &config.robot[i], i);
```

```
        fclose(fp);
        pushFifo(&fifo, config);
        configs[0][ encodeCoords(config.robot[0]) ][ encodeCoords(
                config.robot[1]) ].move = 'X';
}

// Pustí nad daným bludištěm prohledávání do šířky.
void bfs(void) {
        struct sconfig config, newConfig;
        while(popFifo(&fifo, &config)) {
                for(int i = 0; i < 4; i++) {
                        newConfig = getNextConfig(config, moves[i]);
                        if (!isConfigVisited(newConfig)
                                && !isCaptured(config, newConfig)) {
                                markConfigVisited(newConfig, config, moves[i]);
                                if (isRobotOut(newConfig.robot[0])
                                        && isRobotOut(newConfig.robot[1]))
                                        return;
                                pushFifo(&fifo, newConfig);
                        }
                }
        }
}

// Zapíše výsledky do souboru.
void writeResults(void) {
        FILE *fp = fopen("robots.out", "w");
        if (!fp) return;

        // Projdeme možné koncové stavy.
        int i = 0;
        while((i < MAX_PATROL_STATES)
                && (configs[i][MAX_MAZE_SIZE][MAX_MAZE_SIZE].move == 0))
                i++;

        // Neexistuje korektní řešení.
        if (i == MAX_PATROL_STATES) {
                fprintf(fp, "-1\n");
                return;
        }

        // Projdeme cestu zpet k pocatecnimu stavu.
        struct sconfig *config = &configs[i][MAX_MAZE_SIZE][MAX_MAZE_SIZE];
        while(config->move != 'X') {
                results[resultsCount++] = config->move;
                config = &configs[config->ticker][ encodeCoords(
                        config->robot[0]) ][ encodeCoords(config->robot[1]) ];
        }

        // Nalezenou cestu zapíšeme v opačném pořadí do souboru.
        fprintf(fp, "%d\n", resultsCount);
```

```
        while(resultsCount-- > 0)
                fprintf(fp, "%c\n", results[resultsCount]);
}

int main(void) {
        initFifo(&fifo);
        load();
        bfs();
        writeResults();
    return 0;
}
```

20-4-6 Hrady, hrádky, hradla — Martin Mareš

První podúloha, tedy nalezení „křižítka“, bude snadná. Budou nám totiž stačit tři hradla XOR:



Proč tento obvod dělá to, co potřebujeme? Pokud jsou oba vstupy stejné, odpoví prostřední hradlo nulou, takže krajní hradla jen zkopírují vstup na výstup, což je správně. Pokud jsou naopak vstupy různé, prostřední hradlo odpoví jedničkou, takže krajní hradla vstup negují, a to je opět správně.

Dokázat bychom to mohli i algebraicky ($\oplus$ je XOR):

$$Y_2 = X_1 \oplus (X_1 \oplus X_2) = (X_1 \oplus X_1) \oplus X_2 = 0 \oplus X_2 = X_2.$$

A jak se dá na něco takového přijít? Zkusme uvažovat takto: Máme najít obvod se vstupy X_1 a X_2 a výstupy Y_1 a Y_2, který bude vždy počítat $Y_1 = X_1$ a $Y_2 = X_2$. Navíc tento obvod má být rovinný a pokud ho vepíšeme do kružnice, mají vstupy a výstupy po obvodu této kružnice dávat pořadí X_1, X_2, Y_1, Y_2. Jistě můžeme předpokládat, že hradlo, ze kterého vystupuje Y_2, leží přímo na kružnici, takže do něj můžeme podél kružnice přivést vstup X_1. Analogicky hradlo pro Y_1 může znát Y_2. Jak tedy spočítáme z X_1 hodnotu $Y_2 = X_2$? K tomu je potřeba informace o tom, zda se X_1 a X_2 liší či nikoliv. Tu lze snadno počítat uvnitř kružnice. A naše konstrukce je na světě.

Mimochodem, vystačili bychom si i s hradly NAND: Vzpomeňte si na konstrukci hradla XOR ze čtyř NANDů – byla rovinná. Můžeme ji tedy dosadit do našeho schématu a získat křížící obvod z NANDů.

Druhá podúloha je zdánlivě dočista triviální. Pokud nějaký obvod není rovinný, je to proto, že se v něm vodiče kříží. Tak křížení nahradíme naším křižítkem a získáme obvod, v němž je o jedno křížení méně. To nám stačí provést konečně-krát a obvod bude rovinný. Jenže ...

1. V jednom místě grafu by se mohlo křížit i více hran. Pokud se to stane, určitě existuje nějaké malé okolí tohoto bodu, kde nejsou žádná hradla ani další křížení. Tak všechny hrany, které se křížily, „rozšoupneme“ do tohoto okolí a z násobného křížení tím vyrobíme několik obyčejných.

2. Kdybychom křižítko umístili nešikovně, mohli bychom vyrobit nová křížení a celý proces zroviňování by se nikdy nezastavil. Pomůžeme si podobně jako od násobných křížení: najdeme dostatečně malou kružnici okolo místa křížení, kde se vyskytují jenom ty vodiče, které se křížení účastní (taková určitě existuje), a obvod vlepíme do ní. Víme přeci, že se do kružnice dá vepsat a jistě ho můžeme „ohnout“ tak, aby měl vývody na správných místech.

3. Mohli bychom v obvodu vytvořit cyklus – třeba tehdy, když se kříží vodič vedoucí na vstup nějakého hradla s vodičem vedoucím z výstupu téhož hradla. Cykly jsme sice v definici hradel v první sérii výslovně nezakázali (a jeden z ukázkových obrázků dokonce cyklus obsahuje), ale není nijak jasné, jak se takové obvody chovají. Teprve v páté sérii se něco takového pokoušíme zavést. Proto cykly v obvodech, kde dříve nebyly, při zroviňování nechceme vytvářet.

To se zařídí snadno: nakreslíme si schéma tak, aby bylo *topologicky setříděné* podle x-ové souřadnice. Na přímce $x = 0$ budou ležet ta hradla, která závisí pouze na vstupech obvodu; poskládáme je tam libovolně. Na přímku $x = 1$ umístíme ta, která závisí na vstupech obvodu a na výstupech již umístěných hradel, a tak dále. Pokud v obvodu není cyklus, postupně takto umístíme všechna hradla. Všimněte si, že nyní jsou v každém křížení oba vodiče naprosto nezávislé – hodnota na jednom nijak nezávisí na tom, jaká hodnota putuje druhým. Přidáváním křižítek tedy už nemohou cykly vznikat.

20-5-1 Dračí cestování — Mária Vámošová

Protože drak je velmi inteligentní zvíře, před cestou si dobře rozmyslel, že každá cesta z překladiště do překladiště ho stojí drahocennou síru, takže je určitě dobré počet cest minimalizovat a nosit vždy co nejvíc jde. Náklad 14000 kg se dá rozdělit na 3 náklady po 4200 kg a jeden 1400 kg těžký. Tím pádem musí absolvovat celkem 7 cest (4 cesty tam a 3 zpátky) na přenesení celého nákladu na první překladiště. Kdyby mu ale v nějakém momentu zbylo jenom 12600 $(3 \cdot 4200)$ kg síry, dokázal by ji odnést na 5 cest. Pro 8400 kg by se musel vracet jenom jednou, což dělá jenom 3 cesty a 4200 kg unese najednou. Drak si okamžitě uvědomil, že překladiště se mu nejvíc vyplatí dělat tak, aby mu v něm zbylo o jednu várku síry míň než právě má, aby to dokázal odnést na méněkrát. Proč je každé jiné rozložení překladišť horší nebo nanejvýš stejně dobré? Zaveďme si

značení: Nechť $S(A)$ je množství síry v překladišti A, N nosnost draka, SP jeho spotřeba na kilometr, $V(A,B)$ vzdálenost mezi překladišti A a B a $M(A,B)$ množství síry, které je drak schopný přenést z překladiště A do překladiště B. Je celkem snadné vidět, že počet cest, které musí drak uletět na přenesení nákladu z překladiště A do překladiště B je roven horní celé části z $S(A)/N$ vynásobené dvěma (drak se musí vracet) mínus jedna (poslední várku se už nevrací). Tím pádem $M(A,B) = S(A) - \mathit{počet\_cest} \cdot V(A,B) \cdot SP$ (tolik toho drak sní po cestě). Z toho ale vyplývá, že kdybychom nějaké překladiště posunuli dál, pak by drak letěl zbytečně kousek trasy o dvě cesty víc (tam a zpátky), i kdyby už nemusel, čím spotřebuje víc síry. Naopak, kdybychom posunuli nějaké překladiště o kousek blíž, zbylo by na tom překladišti víc síry a drak by musel k dalšímu stanovišti letět zase o dvě cesty víc (protože by ji neunesl na míň cest, kousek by mu tam zbyl). Taky je jednoduché vidět, že dělat si ještě nějaké překladiště navíc nemá smysl, protože jestli projde část trasy na n-krát a pak druhou část trasy taky na n-krát, spotřebuje přitom stejné množství síry jako kdyby letěl celou trasu na n-krát ($\mathit{počet\_cest} \cdot V1 \cdot SP + \mathit{počet\_cest} \cdot V2 \cdot SP = \mathit{počet\_cest} \cdot (V1+V2) \cdot SP$) a tím taky stejné množství síry přenese. Když jsme si tedy dokázali optimální rozložení překladišť, snadno podle toho spočítáme maximální množství síry, které je drak schopný přenést na 4200 kilometrovou vzdálenost. Drak si dělá stanoviště tak, aby na dalším stanovišti zbylo o jednu várku míň než má, tj. v takové vzdálenosti, aby při přenosu spotřeboval právě jednu várku. První překladiště si tedy drak zřídí $1400/7 = 200$ km od startu. Tím se zbaví jedné várky. Další várky se zbaví za $4200/5 = 840$ km, další za $4200/3 = 1400$ km. Nyní je drak 1760 km od cíle a na překladišti má už jenom jednu 4200 kilogramovou várku. Tu si naloží na záda a domů se vrátí s 2440 kg síry.

20-5-2 Piškorcův deratizátor Tereza Klimošová & Tomáš Gavenčiak

A nejhorší ze všeho jsou trpaslíci, ty potvory vám vlezou všude a strašně rychle se množí. Já je chytám a většinou je zahazuju, ale mám švagra a ten si je nechává na chov ...

Protože Temný mág není zdaleka jediný, kdo má se skřítky problémy, pojďme si říct, jak na ně.

Je jednoduché si uvědomit, že čarodějnice může každý den kouzlit vždy jako první. Pokud by v optimálním plánu kouzlila čarodějnice až po kouzelníkovi, mohla klidně kouzlit i před ním. Dále zajisté platí, že pokud může poté kouzelník provést deratizaci, musí ji provést ihned. Pokud by totiž čekal alespoň den, ztrojnásobil by se počet skřítků a on by místo jedné deratizace musel provádět tři.

Jeden den deratizace probíhá tak, že čarodějnice se rozhodne, jak změní populaci skřítků. Poté kouzelník sesílá deratizační kouzlo, dokud není skřítků méně než N. Tedy pouze čarodějnice má možnost volby. A pokud deratizace ne-

skončila (nějací skřítci zůstali), jejich počet skřítků se ztrojnásobí a deratizace pokračuje.

Problém převedeme na hledání nejkratší cesty v grafu. Uvažujme graf s vrcholy očíslovanými 0 až $N-1$. Hodnota vrcholu odpovídá počtu skřítků na konci dne (po kouzlení, ale před trojnásobením). Hrana z vrcholu i do vrcholu j znamená, že se po jednom dni deratizace změní počet skřítků z i na j. Z každého vrcholu vedou nejvýš tři hrany, protože čarodějnice si může vybrat ze tří možností. Hrany v tomto grafu budou ohodnocené a ohodnocení hrany bude počet mágem seslaných deratizačních kouzel, tedy číslo 0, 1 nebo 2.

Nejkratší cesta v tomto grafu z vrcholu $K-1$, K či $K+1$ do vrcholu 0 je určitě řešením naší úlohu. Jak nejrychleji takovou hranu nalézt? Pokud by hrany nebyly ohodnocené, stačilo by použít prohledávání do šířky (pokud nevíte, co to je, nastal dobrý čas pro přečtení kuchařky třetí série 20. ročníku KSP). Protože jsou ale hrany ohodnocené jenom hodnotami 0, 1 a 2, můžeme upravit prohledávání do šířky následovně: nebudeme mít jednu, ale tři fronty F_0, F_1 a F_2. Ve frontě F_i budeme mít vrcholy, do kterých se dostaneme po provedení i deratizačních kouzel. Vrcholy budeme vybírat jenom z F_0 a jejich nenavštívené sousedy, do kterých vede hrana s ohodnocením j, budeme ukládat do fronty F_j. Když nám fronta F_0 dojde, uděláme z F_1 novou frontu F_0, z F_2 novou frontu F_1 a nová F_2 bude prázdná. Tak zaručíme, že vrcholy budeme zpracovávat v pořadí rostoucího počtu deratizací, které potřebujeme, abychom se do těchto vrcholů dostali.

Časová složitost tohoto postupu je $O(N)$, protože graf se skládá z N vrcholů a $3N$ hran, při upraveném procházení navštívíme každý vrchol nejvýš jednou a každého souseda umíme zpracovat v konstantním čase. Paměťová složitost je zřejmě také lineární.

... to přece musí jít lépe! A taky že jo. Je možné dokázat, že stačí najít postup, který zabere co nejméně dní, protože takový postup použije nejmenší možný počet deratizačních kouzel. (Není to zas tak těžké, zkuste si to!) Poté je už řešení v čase $O(\log N)$ nasnadě: na konci nultého dne jsou možné počty skřítků $K-1$, K a $K+1$. Na konci prvního $3K-4$, ... , $3K+4$, konci i-tého $3^iK-(3^{i+1}-1)/2$ až $3^iK+(3^{i+1}-1)/2$, přičemž jde dosáhnout každého počtu mezi. Stačí tedy najít první interval takového tvaru, který obsahuje násobek N-ka.

20-5-3 Obrana před draky — Milan Straka

Udatných drakobijců bylo mezi našimi řešiteli pomálu, takže většinu stačil drak skolit dříve, než stačili mrknout. Tak alespoň posmrtně si můžeme předvést, jak hledat místo pro vypuštění mocného protidračího kouzla.

Nejdřív si uvědomíme, že pokud máme kružnici, která obsahuje optimální počet temných kamenů, můžeme ji vždy posunout tak, aby na její hranici byly

alespoň dva zadané body (alias temné kameny).

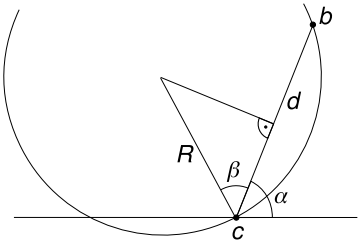
Jednoduchý algoritmus je nyní nasnadě: pro každé dva body najdeme kružnice, na kterých tyto dva body leží. Protože poloměr kružnice je pevně dané R, existují takové kružnice nejvýš dvě. Celkem tedy máme $O(N^2)$ kružnic a víme, že jedna z nich je optimální. Stačí tedy pro každou z těchto kružnic zjistit, kolik je v ní zadaných bodů, což můžeme udělat jednoduše v lineárním čase. Tak získáme řešení s časovou složitostí $O(N^3)$ a lineární paměťovou složitostí.

Je ale zřejmé, že takové řešení třináct bodů nezíská. Nyní si popíšeme řešení se složitostí $O(N^2 \log N)$. Nebudeme zkoumat kružnice pro každé dva body, ale budeme zkoumat vždy všechny kružnice, na jejichž hranici je daný bod.

Vybereme tedy jeden ze zadaných bodů, kterému budeme říkat centrální, a budeme uvažovat kružnici, jejíž střed je o R nad daným bodem (kružnice tedy „stojí“ na tomto bodu). Pokud touto kružnicí budeme kolem daného bodu otáček tak, aby byl pořád na jejím okraji, budou ostatní body vstupovat a vystupovat z této kružnice. Stačí sledovat vstupující a vystupující body, upravovat si počet bodů v kružnici a po jedné otáčce kružnice kolem bodu si vybrat kružnici s maximálním počtem bodů uvnitř. Když tento postup provedeme pro všechny zadané body, získáme zajisté kružnici s největším počtem bodů uvnitř.

Mějme nějaký centrální bod. Polohu kružnice, která se ho dotýká, budeme určovat úhlem, který svírá spojnice středu kružnice a centrálního bodu s osou x. Řekněme, že pro daný centrální bod a další bod dokážeme zjistit úhly, kdy tento další bod vstoupí do kružnice, kterou rotujeme kolem centrálního bodu, a kdy z ní vystoupí. Pro všechny necentrální body získáme $O(N)$ úhlů. Pokud tyto úhly setřídíme, můžeme pak v lineárním čase provést otáčení kružnice kolem centrálního bodu a najít tak kružnici s největším počtem bodů uvnitř. Pro jeden centrální bod bude tento postup trvat $O(N \log N)$ času kvůli třídění úhlů. Celkem tak získáme řešení v čase $O(N^2 \log N)$ s lineární paměťovou složitostí.

Zbývá vyřešit několik detailů. Zaprvé, pro jeden úhel je třeba zpracovat nejprve všechny vstupy a pak až výstupy bodů. Za druhé, v počáteční pozici kružnice musíme spočítat, které body jsou už uvnitř. Můžeme to provést triviálně v lineárním čase, protože to provádíme pro každý centrální bod jenom jednou. A za třetí, musíme umět spočítat úhel vstupu a výstupu bodu z kružnice. Mějme centrální bod c a jiný bod b, který je od něj vzdálený $d \leq 2R$.



Úhel α je zřejmě $\operatorname{atan}[(b.y - c.y)/(b.x - c.x)]$, což můžeme v jazyce C zapsat jako $\operatorname{atan2}(b.y - c.y, b.x - c.x)$, a úhel β je $\operatorname{acos}\left(\frac{d}{2}/R\right)$. Úhel vstupu pak získáme jako $\alpha + \beta$ a úhel výstupu jako $\alpha - \beta$.

Program implementuje popsaný algoritmus s tím rozdílem, že kružnicí otáčí proti směru hodinových ručiček a body, které jsou na začátku v kružnici, poznává tak, že jejich úhel vstupu je větší než úhel výstupu.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#define MAX_B      10000
#define DIST2(a,b) (((a).x-(b).x)*((a).x-(b).x)+((a).y-(b).y)*((a).y-(b).y))
#define EPS        1e-10

double N;
struct bod     { double x;   double y; } body[MAX_B]; int B;
struct udalost { double uhel; int bod; } u[ 2*MAX_B]; int U;

int udalost_cmp_uhel(const void* a, const void* b) {
  if (((struct udalost*)a)->uhel == ((struct udalost*)b)->uhel) return 0;
  return ((struct udalost*)a)->uhel < ((struct udalost*)b)->uhel ? -1 : 1;
}

int main(void) {
  scanf("%d%lf", &B, &N); assert(B>=1);
  for (int i=0;i<B;i++) scanf("%lf%lf", &body[i].x, &body[i].y);

  int best=1; struct bod best_pos=body[0]; double best_uhel=0;
  int body_uvnitr[2*B];
  for (int i=0;U=0,i<B;i++) {                  // každý bod může být na hranici
    int act=1;
    for (int j=0;j==i && j++,j<B;j++) {     // spočítej události ostatních bodů
        double d=DIST2(body[i],body[j]); if (d>4*N*N) continue;
        double uhel_zaklad=atan2(body[j].y-body[i].y,body[j].x-body[i].x);
        if (uhel_zaklad<0) uhel_zaklad+=2*M_PI;
        double uhel_zmena=acos(sqrt(d)/(2.0*N));
        // dvě události, první vstup
        u[U].bod=j; u[U++].uhel=uhel_zaklad-uhel_zmena-EPS;
        // a druhá výstup z kružnice
        u[U].bod=j; u[U++].uhel=uhel_zaklad+uhel_zmena+EPS;
        if (u[U-2].uhel< 0     ) u[U-2].uhel+=2*M_PI;     // normalizace
        if (u[U-1].uhel>=2*M_PI) u[U-1].uhel-=2*M_PI;
        // je bod uvnitř ve výchozí pozici?
        act+=body_uvnitr[j]=u[U-1].uhel<u[U-2].uhel;
    }
    // setřiď události podle úhlu
    qsort(u, U, sizeof(struct udalost), udalost_cmp_uhel);

    for (int j=0;j<U;j++) {                              // projdi události
      act+=body_uvnitr[u[j].bod]?-1:1;
      body_uvnitr[u[j].bod]^=1;
      // aktualizuj maximum
      if (act>best) best=act, best_pos=body[i], best_uhel=u[j].uhel;
```

```
    }
  }

  printf("Temný mák získá %d temných kamenů, pokud zakouzlí v (%g,%g)\n",
         best, best_pos.x+cos(best_uhel)*N, best_pos.y+sin(best_uhel)*N);
  return 0;
}
```

20-5-4 Dračí chodbičky — Michal „vorner“ Vaner

Napřed, jak bude algoritmus fungovat. Nejdříve bude ignorovat veškeré jeskyně s pokladem a na tom zbytku spočítá minimální kostru, například algoritmem popsaným v kuchařce 18-3. Poté vezme každou jeskyni s pokladem a připojí ji k nejbližší jeskyni bez pokladu. Jediné na co si je třeba dát pozor je speciální případ, pouze dvě jeskyně, obě s pokladem.

Proč to funguje? Kdyby byly dvě jeskyně s pokladem spojeny napřímo a žádná další cesta z nich nevedla, pak utvoří zcela samostatnou komponentu. Tedy, každá taková musí být připojená k některé bez pokladu. Je jedno, ke které, neboť zbylé jeskyně musí být navzájem propojené (a lze nahlédnout, že přes jeskyně s pokladem to nelze). Tedy vybereme si tu, ke které vede nejkratší chodba.

Zbylý kus musí být navzájem propojený a mít minimální možný součet hran. Toto přesně počítá algoritmus minimální kostry a jeho zdůvodnění správnosti lze nalézt ve zmíněné kuchařce.

Zbývá ještě časová a paměťová složitost. Paměťová je jednoduchá, pamatujeme si každý vrchol (jeskyni) a hranu (chodbu), tedy $O(N + M)$. V časové bude jednak figurovat tvorba minimální kostry, který je $O(N + M \cdot \log M)$. Při připojování pokladů projdeme každý vrchol a každou hranu nejvýše jednou, tedy zde je složitost $O(N + M)$. Celková bude tedy $O(N + M \cdot \log M)$.

A jedna implementační poznámka na závěr. Obě fáze jsou na sobě zcela nezávislé. Proto je možné tyto dvě fáze prolnout a udělat je obě na jeden průchod seřazenými hranami, jen si u hran dáme pozor, aby maximálně jeden z konců byl s pokladem a nebyl již připojen jinam.

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

struct vrchol_t {
  // Jak bohatá je to jeskyně?
  int poklad;
  // Už jsem to s něčím propojil?
  int spojeno;
  // V tomto postavíme dfu
  int dfu_otec;
  int dfu_vaha;
};
```

```
struct hrana_t {
  // Indexy obou konců
  int vrcholy[2];
  int delka;
};

static int mensi_hrana(const void *_1, const void *_2) {
  return ((struct hrana_t *) _1)->delka - ((struct hrana_t *) _2)->delka;
}

static int dfu_find(struct vrchol_t vrcholy[], int v) {
  if(vrcholy[v].dfu_otec == v)
    return v;
  else {
    int result = dfu_find(vrcholy, vrcholy[v].dfu_otec);
    vrcholy[v].dfu_otec = result;// Zkomprimuj cestu
    return result;
  }
}

static void dfu_union(struct vrchol_t vrcholy[], int v1, int v2) {
  if(vrcholy[v1].dfu_vaha > vrcholy[v2].dfu_vaha)
    dfu_union(vrcholy, v2, v1);// Zapojovat spravnym smerem
  else {
    vrcholy[v1].dfu_otec = v2;
    vrcholy[v2].dfu_vaha += vrcholy[v1].dfu_vaha;
  }
}

int main(void) {
  // Nějaké to načtení
  int vrcholu, hran;
  scanf("%d%d", &vrcholu, &hran);
  struct vrchol_t vrcholy[vrcholu];
  for(int i = 0; i < vrcholu; i++) {
    scanf("%d", &vrcholy[i].poklad);
    vrcholy[i].dfu_otec = i;
    vrcholy[i].dfu_vaha = 1;
    vrcholy[i].spojeno = 0;
  }
  struct hrana_t hrany[hran];
  for(int i = 0; i < hran; i++) {
    scanf("%d%d%d", &hrany[i].vrcholy[0],
        &hrany[i].vrcholy[1], &hrany[i].delka);
  }
  // Hrany jsou potřeba setříděné
  qsort(hrany, hran, sizeof *hrany, mensi_hrana);
  // Bereme jednotlivé hrany a vybírame, jestli je chceme
  for(int i = 0; i < hran; i++) {
    // Nespoj dvě s pokladem, pokud to není speciální případ
```

```
    if(((vrcholy[hrany[i].vrcholy[0]].poklad
        && vrcholy[hrany[i].vrcholy[1]].poklad)
        && (vrcholu != 2))
        || (vrcholy[hrany[i].vrcholy[0]].poklad
        && vrcholy[hrany[i].vrcholy[0]].spojeno)
        || (vrcholy[hrany[i].vrcholy[1]].poklad
        && vrcholy[hrany[i].vrcholy[1]].spojeno))
      continue;
    int komponenty[] = {
        dfu_find(vrcholy, hrany[i].vrcholy[0]),
        dfu_find(vrcholy, hrany[i].vrcholy[1])};
    if(komponenty[0] != komponenty[1]) {
      printf("%d %d %d\n", hrany[i].vrcholy[0],
          hrany[i].vrcholy[1], hrany[i].delka);
      dfu_union(vrcholy, komponenty[0], komponenty[1]);
      vrcholy[hrany[i].vrcholy[0]].spojeno = 1;
      vrcholy[hrany[i].vrcholy[1]].spojeno = 1;
    }
  }
  return 0;
}
```

20-5-5 Roztržitý matematik — Martin „Bobřík“ Kruliš

Milí řešitelé a řešitelky, podle výsledků praktické úložky, které právě nemohu najít, jste si všichni vedli skvěle. Nebo si alespoň myslím, že jste si vedli skvěle ... tedy určitě alespoň většina z vás ...

Každopádně jsem si tu pro vás připravil nástin řešení, abyste si udělali alespoň hrubou představu o tom, jak to u nás chodí a na co si dávat pozor. Na úvod bych rád zdůraznil, že s papíry se to nemá tak jednoduše, jak by se mohlo na první pohled zdát. Jakmile na papír cokoli napíšete, začne žít vlastním životem a sám od sebe se přesunuje. Má tendenci se schovávat pod jiné papíry, když ho právě potřebujete, a naopak ležet na vrchu a překážet, pokud zrovna hledáte něco jiného.

Ale to jsem trochu odbočil ... ach ano – to řešení. Někde jsem ho tu měl připravené. Kam se asi mohlo schovat? V zásadě teď může být kdekoli. Věřili byste, že jsem jednou našel svůj článek dokonce až pod automatem na kávu? Opravdu netuším, jak se tam dostal, protože automat je na chodbě poměrně daleko od mého kabinetu ...

Ale abych se vrátil – problém, se kterým se každý den potýkám, se nazývá move-to-front transformace. Můj kolega z informatiky tvrdí, že se používá také při kompresi, ale to mi příliš nepomůže. Jádro problému spočívá v rychlém nalezení a odebrání i-tého papíru v pořadí a jeho vložení na začátek tak, aby se správně posunuly ostatní papíry.

Půjdeme-li na to přímo, nenarazíme na žádné potíže. Všechny papíry si uložíme do pole tak, že i-tý papír se nachází na indexu i. Nalezení papíru

máme zadarmo v konstantním čase. Papír odebereme a všechny papíry, které jsou před ním, posuneme o jednu pozici. Tím se nám vzniklá díra zaplní a naopak vytvoříme díru na první pozici. Nyní na začátek vložíme odebraný papír a máme hotovo.

Tohle řešení má lineární časovou složitost na každou operaci (tzn. celkem $O(N \cdot k)$, kde N je počet papírů a k počet operací), takže se hodí k přerovnávání několika papírků na stole mého pořádkumilovného kolegy, ale prohledání celého mého kabinetu by zabralo věčnost ...

Dlouho jsem si s tím lámal hlavu, až mi kolega informatik poradil lepší řešení. Jak jsem se dozvěděl, klíčem jsou stromy – tím nemyslím to, co mi roste pod okny, ale binární stromy. Je vhodné použít nějakou variantu vyvažovacích stromů (AVL, Červeno-černé, ...), protože jinak vaše řešení rychle zdegeneruje na lineární spojový seznam. Sám se ve stromech příliš nevyznám, takže pokud vás zajímají detaily, nahlédněte do kuchařky.

V každém vrcholu u bude uložen počet prvků (označme jej $c(u)$) v podstromě, který má u jako kořen, a také číslo papíru, který je v tomto vrcholu uložen.

Takový strom postavíme jednoduše. Na začátku víme, že papíry jsou seřazeny od 1 do N. Kořen našeho stromu bude reprezentovat prostřední papír z daného intervalu. Levý a pravý podstrom pak vygenerujeme rekurzivně. Počet prvků v každém podstromě spočítáme také snadno: stačí v každém vrcholu sečíst:

$$c(\textit{levého podstromu}) + c(\textit{pravého podstromu}) + 1.$$

Nyní se podívejme, jak rychle nalézt, co hledáme. Řekněme, že jsme ve vrcholu u a pátráme po i-tém papíru (oproti zadání je budeme číslovat od nuly, to vyjde elegantněji). Podíváme se na počet prvků v levém podstromě $\ell = c(\textit{levý syn}\ u)$. Pokud je $i < \ell$, víme, že se hledaný prvek nachází v levém podstromu, je-li $i = \ell$, hledaným prvkem je u sám, a konečně v posledním případě ($i > \ell$) se hledaný papír nachází v pravém stromu. Samozřejmě si musíme dát pozor, když přecházíme do pravého podstromu. Tam už nehledáme i-tý papír, ale papír s indexem $i - \ell - 1$.

Odebrání samotného papíru pak probíhá podle pravidel mazaní z binárního vyhledávacího stromu (viz kuchařka a též níže). Stejně tak musíme po mazaní provést vyvážení stromu, které závisí na tom, jaký typ stromu jsme použili (opět viz kuchařka). Po mazání je nezbytné ještě opravit všechny hodnoty $c(u)$ ve vrcholech, které ležely po cestě k hledanému papíru.

Odebraný papír vložíme do stromu na nejlevější pozici (tedy na první místo). Opět dodržíme pravidla pro vkládání do stromu, opravíme všechny hodnoty $c(u)$ po cestě a provedeme vyvážení.

Nakonec potřebujeme ještě vypsat konečnou permutaci dokumentů. Stačí pouze projít a vypsat náš strom v pořadí in-order.

Časová složitost uvedeného algoritmu je $O(\log N)$ na jednu operaci, protože hledání, mazání i vkládání trvá u vyváženého binárního stromu logaritmicky dlouho. Paměťová složitost se nám přitom nezhoršila. Sice spotřebujeme několikrát víc paměti, ale asymptoticky zůstáváme stále na příjemné složitosti $O(N)$.

Jeden student mi ještě tvrdil, že zná řešení v čase $O(k\sqrt{N})$, ale vůbec si nejsem jistý, jak by takové řešení mělo fungovat, takže si můžete zkusit takové řešení napsat za domácí cvičení.

Náš čas na konzultaci bohužel vypršel a já se s vámi musím rozloučit. Někde jsem tu měl papír se seznamem dalších schůzek – ale kam jsem si ho sakra založil ...?

Martin „Bobřík“ Kruliš

V programu si vyzkoušíme jednu méně tradiční, ale moc pěknou odrůdu stromů, totiž BB-α stromy již letmo zmíněné v kuchařce. Místo podle hloubky budou vyvažovány podle váhy, čili počtu vrcholů v podstromech. V dokonale vyváženém stromu platí, že se váha levého a pravého podstromu každého vrcholu liší nejvýše o jedničku. My nebudeme tak přísní: povolíme, aby jeden podstrom měl až dvakrát větší váhu než druhý.

Všimněte si, že toto pravidlo nám stále zaručuje logaritmickou hloubku: oba synové vrcholu s váhou w mají váhy nejvýše $(2/3)w$, jejich synové nejvýše $(2/3)^2 w$ atd., takže váhy stále klesají exponenciálně. Hloubka stromu tedy bude přibližně $\log_{3/2} n$.

Když do stromu přidáme nový vrchol, a nebo naopak nějaký smažeme, přepočítáme všechny váhy na cestě ze změněného vrcholu do kořene (všimněte si, že to tak jako tak v naší úloze potřebujeme). Přitom budeme kontrolovat, jestli váhy stále splňují vyvažovací podmínku. Jakmile objevíme vrchol, ve kterém podmínka neplatí, nebudeme se snažit vyvážení obnovit rotacemi (což by také šlo), ale podnikneme daleko razantnější krok: celý podstrom rozebereme a předěláme na dokonale vyvážený.

Přeskládání celého podstromu je samozřejmě časově náročné: trvá $O(\textit{váha podstromu})$ – inu, když se kácí les, létají třísky –, ale ukážeme, že nebudeme „kácet“ příliš často, takže se nám složitost nepokazí.

Představme si na chvilku, že do stromu jenom vkládáme. Vyberme si nějaký podstrom a sledujme, co se s ním bude dít. Nejprve je během některého kácení postaven jako dokonale vyvážený a tehdy se váha jeho levého a pravého syna rovnají (± 1) nějakému číslu w. Pak do podstromu přibývají nové prvky a vyvážení se postupně zhoršuje. Nakonec se situace stane příliš nahnutou, a tak podstrom pokácíme. Všimněte si, že mezi vytvořením podstromu a jeho pokácením se musel jeden syn stát dvakrát těžším než druhý, takže se muselo objevit alespoň w nových prvků. Přebudování samo trvá $O(w)$, tudíž stačí, aby na něj každý z w nově vložených prvků přispěl časem $O(1)$. Každý prvek si tak-

to předplatí všechna pokácení na cestě z něj do kořene (přesně těm přispívá), a tak celkově přispěje časem $O(\log n)$.

Mazání nám tuto analýzu trochu zkomplikuje, ovšem snadno nahlédneme, že když je povoleno jak vkládat, tak mazat, nejrychlejší způsob, jak pokácet strom váhy $2w$, je smazat z jednoho jeho podstromu $w/2$ vrcholů a druhý podstrom nechat na pokoji. (Na počátku měly podstromy váhy w, na konci má jeden $w/2$ a druhý stále w.) Opět na to potřebujeme řádově w operací, takže původní úvaha s příspěvkem $O(\log n)$ na operaci stále funguje.

Kolik času tedy spotřebujeme na jednu operaci? Inu, když máme smůlu a zrovna jsme způsobili jedno nebo více kácení, operace může trvat až n kroků. Ovšem díky našemu principu „předplácení si“ stále platí, že libovolných k po sobě jdoucích operací trvá $O(k \log n)$, což nám pro řešení úlohy úplně stačí. Tomu se říká, že každá operace má *amortizovanou časovou složitost* $O(\log n)$.

Stálo to trochu počítání, ale zato jsme si ušetřili spoustu programování. Inu, i informatici jsou líní a někdy i roztržití.

Martin Mareš

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

struct node {                   // Vrchol stromu
  struct node *left, *right;    // Synové
  int id;                       // Číslo papíru
  int weight;                   // Velikost (váha) podstromu
};

static int weight(struct node *n)
{
  return (n ? n->weight : 0);   // Váha stromu, prázdný má 0
}

// Vytvoří ze stromu seznam, 'right' ukazuje na následníka.
struct node *tree_to_list(struct node *n, struct node *tail)
{
  if (!n)
    return tail;
  n->right = tree_to_list(n->right, tail);
  return tree_to_list(n->left, n);
}

// Odpojí 'count' prvků ze seznamu '*lp' a vytvoří z nich
// vyvážený strom. Vrátí ukazatel na kořen.
struct node *list_to_tree(struct node **lp, int count)
{
  if (!count)
    return NULL;
  int half = (count-1)/2;
  struct node *left = list_to_tree(lp, half);
```

```
  struct node *root = *lp;
  *lp = root->right;
  root->left = left;
  root->right = list_to_tree(lp, count-1-half);
  root->weight = count;
  return root;
}

// Přepočítá váhy podle synů a pokud je strom příliš vychýlený, přebuduje ho.
struct node *reweight(struct node *n)
{
  int lw = weight(n->left);
  int rw = weight(n->right);
  n->weight = 1 + lw + rw;
  if (lw+rw > 1 && (lw > 2*rw || rw > 2*lw))
    {
      struct node *tmp = tree_to_list(n, NULL);
      n = list_to_tree(&tmp, n->weight);
    }
  return n;
}

// Vloží vrchol do stromu před všechny ostatní.
struct node *add_start(struct node *root, struct node *new)
{
  if (!root)
    {
      root = new;
      new->left = new->right = NULL;
    }
  else
    root->left = add_start(root->left, new);
  return reweight(root);
}

// Smaže k-tý nejmenší prvek ve stromu a vrátí jak tento
// prvek (*kthp), tak nový kořen stromu.
struct node *del_kth(struct node *n, int k,
                     struct node **kthp)
{
  if (k < weight(n->left))
    { // k-tý nejmenší je v levém podstromu
      n->left = del_kth(n->left, k, kthp);
      return reweight(n);
    }
  k -= weight(n->left);
  if (k)
    { // k-tý nejmenší je v pravém podstromu
      n->right = del_kth(n->right, k-1, kthp);
      return reweight(n);
    }
```

```
  // Mám ho, pryč s ním. Má jen jednoho syna?
  *kthp = n;
  if (!n->left)
    return n->right;
  if (!n->right)
    return n->left;
  // Má dva => prohodím s maximem levého podstromu.
  n->left = del_kth(n->left, n->left->weight-1, kthp);
  int id = n->id;
  n->id = (*kthp)->id;
  (*kthp)->id = id;
  return reweight(n);
}

// Vypíše všechny hodnoty ve stromu.
void dump_tree(FILE *fo, struct node *root)
{
  if (!root)
    return;
  dump_tree(fo, root->left);
  fprintf(fo, "%d ", root->id);
  dump_tree(fo, root->right);
}

int main(void)
{
  FILE *fi = fopen("papiry.in", "r");
  FILE *fo = fopen("papiry.out", "w");
  int N, K, op;
  fscanf(fi, "%d%d", &N, &K);

  // Vložíme všechna lejstra do stromu.
  struct node *root = NULL;
  for (int i=N; i>0; i--)
    {
      struct node *n = malloc(sizeof(*n));
      n->id = i;
      root = add_start(root, n);
    }

  // Vykonáváme příkazy ze vstupu.
  for (int i=1; i<=K; i++)
    {
      fscanf(fi, "%d", &op);
      struct node *n;
      root = del_kth(root, op-1, &n);
      root = add_start(root, n);
    }

  // Vypíšeme, jak to dopadlo.
  dump_tree(fo, root);
```

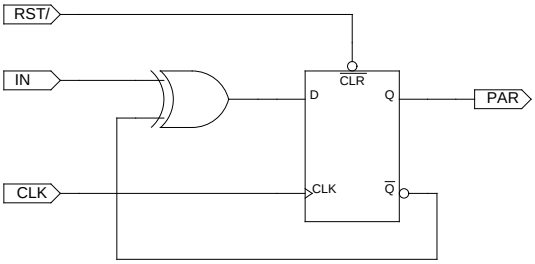
```
  fputc('\n', fo);
  fclose(fo);
  fclose(fi);
  return 0;
}
```

20-5-6 Hrady, hrádky, hradla — Martin Mareš

Jak se průběžná parita má chovat? Inu, pokud na vstupu přijde nula, parita se nijak nemění. Pokud naopak přijde jednička, parita se zneguje. Jinými slovy nová parita je XOR staré parity s bitem na vstupu. Pak už stačí přidat klopný obvod D, aby si paritu pamatoval a novou zapsal při náběžné hraně hodinového signálu.

Ještě bychom měli domyslet, jakou hodnotu má mít parita na začátku výpočtu. Za tímto účelem doplníme klopný obvod D o vstup `CLR` (clear), který způsobí jeho vynulování, a necháme na uživateli, ať na začátku „zatahá za reset“ a obvod zinicializuje. (Takový vstup mají i opravdové klopné obvody.)

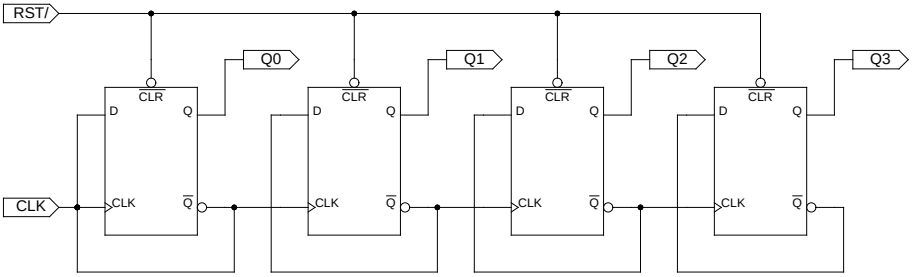
Celý obvod bude tedy vypadat takto:



Ani čítač nebude složitý. Nejnižší bit dvojkového čítače při každém „tiku“ hodin svou hodnotu zneguje, jinými slovy na jeho spočítání stačí dělička frekvence zmíněná v zadání. Další bit v pořadí se změní právě tehdy, když nejnižší bit přejde z jedničky do nuly, čili když nastane náběžná hrana na negovaném výstupu zmíněné děličky. Takže stačí na tento negovaný výstup napojit další děličku, a tak dále.

Obvod pro N-bitový čítač tedy bude sestávat z N klopných obvodů D zapojených jako děličky. Schéma pro $N = 4$ najdete níže.

Zbývá dodat snad jen to, že i u obvodů tohoto druhu má smysl zkoumat časovou složitost – v tomto případě dobu, za kterou se obvod po tiku hodinového signálu ustálí a vydá stabilní výstup. U našeho čítače je to $O(N)$ kroků (změna musí „probublat“ až N D-čky). Šel by ale postavit i tak, aby reagoval už za $O(\log N)$ kroků, zkuste si to třeba o prázdninách vymyslet. S páječkou v ruce se s vámi loučí autoři seriálu.


RST/
Q0
Q1
Q2
Q3
CLR
D
Q
CLK
Q̄

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
1.	Peter Ondrúška	SPŠDubnica	4	23	224,4
2.	Filip Hlásek	GMikulášPL	1	21	173,5
3.	Vlastimil Dort	GŠpitálsPH	2	19	146,5
4.	Jan Michelfeit	G HBrod	4	17	144,3
5.	Filip Štědronský	GMikulášPL	1	16	138,1
6.	Alena Skálová	GNaVPláni	4	15	134,8
7.	Petr Malý	GSladNámPH	4	16	133,6
8.	Libor Plucnar	GPBezruče	3	15	119,9
9.	Štěpán Weber	GBuďánkaPH	3	13	113,4
10.	Pavel Veselý	G Strakon	3	16	94,9
11.	Trung Ha duc	GMasarykPL	2	12	93,0
12.	Tomáš Toufar	G Bílovec	4	10	83,5
13.	Vojtěch Tůma	G Jihlava	4	11	81,3
14.	Petr Babička	G SvětláNS	3	10	64,2
15.	Stanislav Fořt	G Tábor	0	14	60,2
16.	Jan Škoda	GMikulášPL	1	8	59,1
17.	Jitka Novotná	G Bílovec	3	8	58,6
18.	Lukáš Kripner	G Litvínov	2	7	57,7
19.	Jakub Hrnčíř	GFXŠaldyLI	1	8	55,9
20.	Radim Cajzl	G NMnMor	1	8	54,9
21.	Pavel Kratochvíl	ZŠSvětlá	0	7	51,0
22.	David Marek	SPŠ Zlín	4	7	49,2
23.	Tomáš Jakl	G MTřebová	4	8	45,9
24.	Adam Streck	G Hořice	4	5	35,6
25.	Jan Matějka	GJírovcoČB	3	4	35,4
26.	Milan Rybář	GJungmanLT	3	6	35,0
27.	Jakub Kaplan	GJKTylaHK	4	4	34,4
28.	Jiří Setnička	G25březnPH	1	6	34,3
29.	Roman Smrž	GOhradníPH	4	3	34,0
30.	Jakub Suchý	GMikulášPL	1	4	33,9
31.	Pavel Taufer	GArcibisPH	2	5	33,1
32.	Jiří Zárevúcky	SŠInformFM	3	4	32,5
33.	Tomáš Sýkora	G VKlobou	4	5	30,8
34.	David Brázdil	G Zlín	3	6	29,8
35.	Martin Vlach	G Jihlava	4	4	29,7
36.	Karel Tesař	SPŠEPlzeň	2	4	28,4
37. – 38.	Jiří Keresteš	SPŠEPlzeň	2	4	28,2

37. – 38.	Petr Sokola	SPŠ Zlín	4	3	28,2
39.	Vojtěch Kolář	G Neratov	3	4	27,3
40.	Dominik Smrž	GOhradníPH	0	4	27,1
41.	Jakub Červenka	GŠpitálsPH	2	3	26,0
42.	Martin Patera	GArabská	2	5	25,5
43.	Jan Žák	G HBrod	3	3	24,1
44.	Alžběta Pechová	SPŠSVsetín	3	4	22,6
45.	Miroslav Klimoš	G Bílovec	3	2	19,4
46.	Jan Vaňhara	G Holešov	3	3	17,8
47.	Marek Nečada	G Jihlava	4	1	10,6
48.	Petr Holášek	G Příbor	4	2	10,5
49.	Nikolas Zigmund	ZŠHavířov	1	5	8,9
50.	Lukáš Timko	G Tábor	0	4	8,6
51.	Peter Uhnák	GBBolzana	2	1	7,4
52.	Peter Smatana	EkoGLabsBO	3	1	6,4

Obsah

Petr Kratochvíl a kolektiv

Korespondenční seminář z programování XX. ročník

Autoři a opravující úloh:
Jan Bulánek, Pavel Čížek, Zbyněk Falt, Tomáš Gavenčiak, Cyril Hrubiš, Tereza Klimošová, Petr Kratochvíl, Jana Kravalová, Martin Kruliš, Pavel Machek, Martin Mareš, Petr Onderka, Josef Pihera, Milan Straka, Mária Vámošová, Michal Vaner

Vydal MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 249. publikaci.

TEX-ová makra pro sazbu ročenky vytvořil Martin Mareš.
S jejich pomocí ročenku vysázel Petr Kratochvíl.
Ilustrace (včetně té na obálce) vytvořil Martin Kruliš.
Sazba byla provedena písmem Computer Modern v programu TEX.

Vytisklo Reprostředisko UK MFF.

Vydání první, 176 stran
Náklad 300 výtisků
Praha 2008

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

ISBN 978-80-7378-055-5

ISBN 978-80-7378-055-5