Milí řešitelé a řešitelky!

Přichází druhé pololetí středoškolského školního roku, my vysokoškoláci dotahujeme druhou třetinu tradičního zápasu studenti vs. vyučující a zima je v plném proudu. Navíc aritmetický průměr čísel v těchto dvou odstavcích je přibližně 3,053, což je docela blízko π.

Třetí sérii jsme úspěšně uzavřeli a čtvrtou držíte v ruce. Jako tradičně obsahuje 7 úloh, ze kterých se do celkového bodového hodnocení započítávají 4 nejlépe vyřešené.

Nezapomeňte, že k vyřešení některých úloh stačí prostudovat vhodné kuchařky.

Termín odevzdání čtvrté série je stanoven na pondělí 28. března v 8:00 SELČ, což znamená, že papírové řešení byste měli podat na poštu do středy 23. března.

Řešení přijímáme elektronicky na stránce *https://ksp.mff.cuni.cz/submit/*. Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš šifrovací certifikát – zde je jeho SHA1 hash: `7F:53:E7:00:60:F2:24:93:8F:52:51:EC:1E:A8:34:54:86:69:32:7D`.

Také nám řešení můžete poslat klasickou poštou na adresu

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

Na případné dotazy vám rádi odpovíme také na adrese *ksp@mff.cuni.cz* a v diskusním fóru na našem webu.

Čtvrtá série třiadvacátého ročníku KSP

Jen výjimečně se najdou lidé píšící programy s dokumentací obsáhlejší než samotný kód a provádějící veškerou činnost s obdivuhodnou pečlivostí. V této sérii si představíme osobnost, pro niž je psaní dobře okomentovaných programů takřka denním chlebem a která povýšila programování na umění programování.

Možná už tušíte, že se jedná o Donalda Ervina Knutha, a vybavilo se vám jeho dílo Umění programování (v originále The Art of Computer Programming), obsahující mnohé znalosti informatiky, popisy základních algoritmů a jejich matematickou analýzu. Výjimečnost tohoto muže potvrzuje i titul emeritního profesora (plným názvem Professor Emeritus of The Art of Computer Programming) na Stanfordově univerzitě v USA.

Emeritní znamená, že odešel z profesionálního života, nicméně mu zůstal čestný profesorský titul. Knuth se totiž rozhodl opustit univerzitu, aby se mohl plně věnovat práci na Umění programování. Chce-li napsat o nějakém tématu, hluboko se do něho ponoří, přečte o něm spoustu článků, z nichž vybere pro čtenáře nejpřínosnější poznatky, každý algoritmus si naprogramuje…

Proto také jeho práce na Umění programování trvá již od roku 1962, nyní jsou napsány tři svazky (Základní algoritmy, Seminumerické algoritmy, Vyhledávání a třídění), čtvrtý (Kombinatorické algoritmy) se dokončuje, ale v plánu jsou ještě další tři.

Celkem by tedy mělo vyjít sedm svazků.

23-4-1 Studenti a profesoři — 12 bodů

Jedna z věcí, jež ho na univerzitě zaměstnávala (a tedy mu ubírala čas od psaní), bylo vedení vědeckých prací studentů (např. psaní článků do odborných časopisů).

Studenti na Stanfordově univerzitě se chtějí prosadit a napsat co nejvíce článků, přičemž každý z nich má vytipováno několik profesorů, pod jejichž vedením by chtěl článek psát. S jinými profesory spolupracovat nechce a nebude.

Studenti jsou schopni psát maximálně K článků najednou. Leč čas profesorů je omezený, každý z nich je totiž ochoten spolupracovat maximálně s K studenty, přičemž je jim jedno, kteří to budou.

Vaším úkolem je najít algoritmus, který zjistí, jestli je možné, aby každý student psal právě K článků a každý profesor spolupracoval právě s K studenty, a pokud ano, tak vypsat, který student bude spolupracovat s kterým profesorem.

Můžete předpokládat, že profesorů i studentů je stejně, totiž N, a nemusíte uvažovat situaci, že by student chtěl psát u jednoho profesora více článků.

Příklady: pro vstup $N = 4$, $K = 2$, student S1 chce psát článek s profesory P1 a P2, student S2 s P1, P2, P3, P4, student S3 s P2, P3, P4 a student S4 s profesory P3, P4, jsou řešením tyto páry student–profesor: S1–P1, S1–P2, S2–P1, S2–P3, S3–P2, S3–P4, S4–P3, S4–P4.

Pro vstup $N = 5$, $K = 2$, studenti S1 a S2 chtějí psát u profesorů P1, P2, P3, student S3 u P3, P4, P5 a studenti S4, S5 u P4, P5, řešení neexistuje. Existovalo by, kdyby bylo $K = 1$, ale to už je zase jiný vstup.

Aby byl Knuth při své práci co nejméně vyrušován, zrušil si 1. ledna 1990 e-mailovou adresu. Jak píše na svém webu, cítí se nyní šťastnější, protože mu už nechodí nevyžádaná pošta. I přesto výhod elektronické komunikace a internetu stále využívá, ale většinu e-mailů za něj vyřizuje jeho sekretářka.

23-4-2 Paralelní profesoři — 10 bodů

Na chvíli si představme, že neexistuje nic jako e-mail, internet, ba dokonce obyčejná „šnečí“ pošta. Jak by si takoví profesoři sdělovali své poznatky?

Na univerzitě pracuje N profesorů. Na začátku dne má každý svůj unikátní nový objev, který chce sdělit všem ostatním.

Jelikož však ve skupince třech a více profesorů je vzájemné sdělování poznatků nemožné kvůli překřikování, pořádají

profesoři během dne občas sezení. Během nich utvoří dvojice (ne nutně všichni jsou ve dvojici) a ve dvojicích si vymění všechny objevy, které mají (tedy i objevy získané dříve od jiných profesorů). Během sezení se dvojice nemění.

Jaký je pro různá N minimální počet sezení, která musí proběhnout, aby každý profesor znal objevy všech svých kolegů?

Třeba pro $N = 2$ stačí jedno sezení, pro $N = 4$ jsou potřeba dvě (nejdřív A–B a C–D a potom A–C a B–D).

Jak D. E. Knuth nedávno v jednom rozhovoru poznamenal, mnohdy naráží na odborné texty, jejichž jediným cílem je překonat tajuplnými metodami jednodušší algoritmy, ale pouze, když bude velikost vstupu větší než počet protonů ve vesmíru. Ve svých knihách proto předkládá jednodušší, ale stále efektivní metody.

Jak však ukáže následující úloha, ne vždy jsou jednoduché a rychlé metody dobré.

23-4-3 Zabugovaný program — 8 bodů

Dva zlatokopové objevili bohaté ložisko zlata a společně vykopali velké množství nugetů. Chtějí se o ně rozdělit rovným dílem, na což si napsali program. Každý nuget má svoji zadanou cenu (přirozené číslo), seznam nugetů program dostane na vstupu, na výstup vypíše, jak si je mají rozdělit, nebo oznámí, že řešení neexistuje.

Například pro nugety o hodnotách 3, 3, 5, 5 dostanou oba nugety s cenou 3, 5; pro sadu nugetů 3, 3, 5 řešení neexistuje.

Co čert (nebo spíš zlatokop) nechtěl, v programu je chyba a ne malá, nehledejte tedy zapomenutý středník, chybu v použití knihovní funkce jako `qsort` nebo neošetření čtení vstupu. Zlatokopové špatně vymysleli celý algoritmus a program by si zasloužil od základu přepsat.

To však nechte na nich, ať se pocvičí v algoritmizaci, vaším úkolem bude pouze přesvědčit je, že program napsali špatně – najít jim vstup, na němž vypíše špatný výsledek, a určit pro tento vstup správný výsledek. Bonusové body neminou řešitele, kteří najdou nejmenší takový vstup co do počtu předmětů nebo celkové ceny.

Oba programy (C, Python) naleznete na konci letáku.

Zmiňme ještě trochu biografických údajů. Donald Ervin Knuth se narodil v roce 1938 ve Wisconsinu. Už od mládí vykazoval vysokou inteligenci, když v 8 letech dokázal složit z písmen „Ziegler's Giant Bar" 4500 anglických slov do jedné soutěže, přestože porota měla jen 2500 slov. V roce 1956 nastoupil na Case Institute of Technology na fyziku, ale byl záhy přesvědčen, aby se věnoval matematice.

K informatice se dostal v podstatě náhodou při své letní práci, když narazil na počítač IBM 650. Zaujal ho natolik, že u něj strávil dlouhé hodiny jeho zkoumáním. Ve 20 letech napsal program na analýzu výkonnosti univerzitního basketbalového týmu, který mu vynesl trochu slávy. Tomuto počítači dokonce později věnoval jednu svoji práci slovy „na památku mnohých příjemných odpolední".

S pracemi na Umění programování začal již v roce 1962 (tedy 6 let po nástupu na vysokou školu) a první tři svazky vyšly v letech 1968, 1969 a 1973. Poté byl však zklamán změnou techniky sazby jeho knih, protože se změnilo písmo a snížila kvalita. Rozhodl se proto vyvinout vlastní systém pro sazbu textů, a tak vznikly jeho dva další známé počiny: TEX jako nový systém pro sázení textů a METAFONT pro tvorbu fontů.

Dotáhl sazbu a propracovanost svých knih dokonce tak daleko, že se rozhodl vyplatit 0x$1.00 (jeden hexadecimální dolar, tedy $2.56) každému, kdo najde v nějaké jeho knize chybu. Na svých stránkách má seznam odměněných, který dnes čítá bezmála 400 jmen.

23-4-4 Závorky v TEXu — 10 bodů

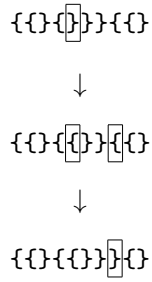
V TEXu se hojně využívají složené závorky (např. v definicích či voláních maker) a lze je i libovolně vnořovat. Občas se ale stane, že se člověk překlepne a místo `{` napíše `}` nebo naopak.

Vymyslete algoritmus, který na vstupu dostane posloupnost (řetězec) N složených otevíracích a zavíracích závorek (bez dalších jiných znaků) a nalezne minimální počet změn znaku `{` na znak `}` nebo naopak, aby byl řetězec správně uzávorkovaný. Žádné jiné změny, kromě přepsání jednoho znaku na druhý, nejsou povoleny.

Správně uzávorkovaný znamená, že ke každé otevírací závorce existuje odpovídající uzavírací, která je od ní napravo, a podobně ke každé uzavírací existuje odpovídající otevírací, jež je od ní nalevo.

Nepůjde-li posloupnost závorek žádným způsobem změnit na správně uzávorkovaný řetězec, měl by to být váš algoritmus schopen rozpoznat.

Příklad: pro vstup `{{}{}}}{{}` je jedním z možných nejkratších postupů ke správnému uzávorkování tento:



Výsledek je tedy 2.

Jak je uvedeno na začátku, Knuth píše dobře dokumentované programy. V 70. letech, kdy vznikal TEX, si vymyslel dokonce vlastní styl programování, v originále nazvaný literate programming (česky by se dalo přeložit jako „kultivované programování"), který se snaží programování více přiblížit myšlení člověka a ne potřebám strojů.

Ve zdrojovém kódu se míchá dokumentace a samotný kód (např. v Pascalu či C), přičemž Knuth si naprogramoval utility na vyextrahování čistého zdrojového kódu pro účely kompilace a programátorské dokumentace v TEXu.

O tom, že Knuth není žádný suchar a rád si hraje s čísly, vypovídá několik věcí. Už jako středoškolský student odeslal do soutěže vědeckých talentů svůj první matematický článek o číselných soustavách se záporným či dokonce komplexním základem. Vymyslel soustavu o základu $2i$, jejíž speciální vlastností je, že každé komplexní číslo může být reprezentováno pouze číslicemi 0, 1, 2 a 3 a bez znaménka.

Zvláštní je také systém číslování verzí TEXu a METAFONTu. Jak se TEX stává dokonalejším, jeho verze se stále více blíží číslu π. Prohlásil, že po jeho smrti se číslo verze programu TEX s definitivní platností ustálí na π a všechny zbývající bugy se stanou vlastnostmi programu. Podobně se verze METAFONTu blíží základu přirozeného logaritmu, číslu e, a po jeho smrti bude provedena podobná změna jako u TEXu.

23-4-5 Palindromnásobky — 11 bodů

Když už jsme u těch čísel, naprogramujeme si jednu zajímavou úlohu z této oblasti. Víte, co je zajímavé na roce 1991? Když ho vydělíte 11, získáte 181, přičemž všechna tato tři čísla jsou palindromy. (Palindrom je takový řetězec, který se stejně čte zleva i zprava.)

Vaším úkolem bude napsat program, který na vstupu dostane čísla K a D a na výstup vypíše počet násobků čísla K, jež mají délku D a jsou palindromy (všechno v desítkovém zápise). $1 < K < 1\,000$ a $D < 20$.

Zdá-li se vám, že takových čísel musí být hrozně málo, tak věřte, že pro každé K nedělitelné 10 existuje nekonečně mnoho jeho palindromických násobků. Číslo dělitelné 10 takový násobek nemá, protože se nuly na začátku nepíší.

Formát vstupu a výstupu: v souboru `vstup.in` jsou na prvním řádku dvě přirozená čísla K a D oddělená mezerou. Do souboru `pocet.out` vypište na první řádku počet násobků K délky D, které jsou palindromy.

Příklady:

vstup.in	pocet.out
3 1	3
25 3	2
12 4	7
60 4	0
81 6	0

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[1] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

Veškerý svůj čas však Don Knuth nevěnuje jen informatice. Se svou ženou Jill mají doma například vlastní varhany s 812 píšťalami.

Další zajímavou zálibou je focení kosočtverečných dopravních značek, které se vyskytují např. v USA, Kanadě a Austrálii. Na svých stránkách má slušnou sbírku 1069 fotek různých dopravních značek,[2] včetně kuriozit jako značky s nápisem „Anti-icing spray system“.

23-4-6 Knuthovy cesty po státech — 9 bodů

Knuth si během jedné cesty po státech, při níž fotil značky, při průjezdu křižovatkou vždy zapsal její číslo. On si je totiž předem očísloval. Zajímalo by ho, jakou největší souvislou část cesty (podle počtu křižovatek) neprošel žádnou křižovatkou více než jednou.

Například pro posloupnost křižovatek

$$3, 4, 1, 2, 4, 8, 7, 2, 3, 8, 2, 9, 1, 4$$

je správným řešením posloupnost

$$3, 8, 2, 9, 1, 4.$$

Jako třešničku na informatickém dortu si uvedeme citát z jednoho jeho dopisu: „Beware of bugs in the above code; I have only proved it correct, not tried it.“ („Pozor na chyby ve výše uvedeném kódu; pouze jsem dokázal jeho správnost, ale nezkoušel jsem ho.“)

I přes stáří 73 let používá Knuth moderní prostředky. Své internetové stránky[3] často aktualizuje a pracuje na dvou počítačích: má Mac na vytváření grafiky a přístup k internetu a notebook s Linuxem na práci. Je tedy možné, že používá i některé z utilit zmíněných v dalším díle seriálu (ačkoliv píše v editoru Emacs a ne ve Vimu).

23-4-7 Bratrstvo Seda a Grepa — 16 bodů

Tento text volně navazuje na předchozí tři série, některé pasáže nemusí být lehce pochopitelné bez jejich znalosti.

Ukážeme si, jak regulární výrazy používáme v praxi. Programy, které nás budou zajímat, jsou UNIXové nástroje `sed`, `grep` a případně světoznámý editor Vim. Jsou snad v každé distribuci Linuxu, na Windows je možné použít balík Cygwin.[4]

Na vyhledávání v textu se používá program `grep`. Není to žádná aplikace s klikacím frontendem, používá se trapně v terminálu. O to je však rychlejší. Základní použití je `egrep <regex>`, kdy program čte standardní vstup (to, co mu píšete na klávesnici) a vypisuje na standardní výstup (terminál). Vypisuje ty řádky, na kterých našel podřetězec vyhovující zadanému regexu.

Možná si říkáte – proč `egrep`? Pohled do manuálu (`man grep`) napoví, že existovaly starší regexy, které toho uměly méně a mají trochu jinou syntaxi. Kvůli zpětné kompatibilitě starých skriptů byla tedy přidána tato varianta programu `grep`. Ze stejného důvodu budeme později u programu `sed` používat přepínač `-r`.

Když si tedy pustíme `egrep 'ba*gr+'`, tak na vstup

```
grbagr
baagrr
rgba
bbarg
rbgrbgg
```

dostaneme výstup

```
grbagr
baagrr
rbgrbgg
```

Můžeme chtít, aby `grep` četl vstup ze souboru. Za zadaný regex můžeme napsat libovolně mnoho jmen souborů, které má číst (oddělené mezerami).

Pokud čte jeden, vypíše prostě vhodné řádky. Když jich je víc, vypíše na začátku každého řádku ještě jméno souboru, ve kterém jej našel. Toto chování se dá regulovat volbami `-h` (bez jmen) a `-H` (vypiš jméno, i když je soubor sám), které můžete napsat před výraz.

Takže například

```
egrep -h Kája jmena prijmeni archiv
```

vyhledá všechny řádky ze souborů

`jmena`, `prijmeni` a `archiv`,

[1] http://ksp.mff.cuni.cz/zaciname/codex.html
[2] http://www-cs-faculty.stanford.edu/~uno/diamondsigns/diam.html
[3] http://www-cs-faculty.stanford.edu/~uno/
[4] http://www.cygwin.com/

které obsahují výraz `Kája`. Volba `-h` potlačila výpis jmen souborů.

Ještě existuje zajímavá volba `-v`, která logicky obrátí výpis (vypisuje jen ty řádky, které hledaný výraz neobsahují). Například když hledáte, jak často k vám na web chodí lidé z jiných prohlížečů než IE a FF, tak ze záznamů prostě vyřadíte řádky odpovídající IE a FF...

Programy za sebe můžete řetězit znakem `|` – vezme standardní výstup prvního programu a nevypisuje ho na terminál, ale předhodí ho druhému programu na standardním vstupu. Například `egrep ba+gr|egrep -v baaagr`. Někdy je jednodušší programy zřetězit než psát jeden dlouhý výraz, který pojme všechno.

Tento znak se také označuje jako „roura“ (*pipe*), protože to dávným programátorům připadalo, jako by mezi programy prostě natahovali potrubí.

Složitější výrazy se hodí psát mezi apostrofy: `'(\(|\))'`, jinak je může interpretovat ještě terminál samotný a z `a*` udělat seznam souborů, které začínají `a`, což rozhodně nechcete, a to je to nejmenší, můžou se stát daleko horší věci...

Dosud jsme regulárními výrazy pouze vyhledávali. Jde však o daleko mocnější zbraň, výrazně větší kladivo. Jak bylo zmíněno už v prvním dílu, regexy často používáme k systematickému nahrazování v textu.

Na nahrazování se hodí program `sed`. Jeho základní použití je podobné jako u programu `grep`.

Jak ale vypadá nahrazovací výraz? Začíná písmenkem `s`, pak následuje oddělovač (typicky `/` nebo `#`, ale může to být libovolný znak, klidně písmenko), pak hledaný řetězec, potom znovu stejný oddělovač, potom řetězec k nahrazení, a nakonec zase oddělovač. Například nahrazovací výraz `s/ahoj/nazdar/` nahrazuje slovo `ahoj` za `nazdar`.

Oddělovač se potom stává speciálním znakem a pokud jej chcete použít v původním významu, je potřeba to říct – obackslashovat jej. (`\/`, `\#`)

Jak to funguje jako příkaz? Když pustíte příkaz

```
sed -r 's/ahoj/nazdar/',
```

čte standardní vstup po řádkách, na každém řádku hledá výraz `ahoj`, jeho první nalezený výskyt změní na `nazdar` a změněný řádek vypíše na výstup.

Takže pokud vstup

```
moskyto@atrey.karlin.mff.cuni.cz
bagr
ksp@mff.cuni.cz bait@uu.ucw.cz
kombajn
```

proženeme třeba příkazem `sed -r 's/@/ (at) /'`, dostaneme výstup

```
moskyto (at) atrey.karlin.mff.cuni.cz
bagr
ksp (at) mff.cuni.cz bait@uu.ucw.cz
kombajn
```

Všimněte si na třetím řádku zbylého zavináče. Kdybyste potřebovali nahrazovat všechny výskyty, ne jen ten první, tak musíte za výraz ještě připsat `g` (od „globally“):

```
sed -r 's/@/ (at) /g'.
```

Pokud potřebujete „přišpendlit“ celý výraz na začátek nebo na konec řetězce, uveďte stříšku `^` na jeho úplném začátku nebo dolar `$` na jeho konci. To funguje jak pro `sed`, tak pro `grep`.

Úkol 1 [1b]: Napište nahrazovací výraz, který smaže všechny „trailing spaces“ – bílé znaky na koncích řádků (stačí asi mezera, `\t` a `\r`).

Jenom takhle by to ale bylo trapné. Co když potřebujeme z řádku jenom něco vytáhnout? Pak opravdu můžeme „najít“ třeba celý řádek jedním výrazem a vybrat si z něj jenom tu část, kterou potřebujeme.

`sed -r 's/^.*"(.*)".*$/\1/'` z celého řádku nechá jen řetězec v uvozovkách.

Takže vstup vlevo se přepíše na výstup vpravo.

`"`	`"`
`bagr`	`bagr`
`b"ag"r`	`ag`
`b"a"g"r`	`g`

Když tedy kus řetězce uzávorkujete, můžete se na tyto závorky pak odkazovat pomocí znaku `\` následovaného číslicí (`1` – `9`). Závorky jsou číslovány zleva doprava podle své otevírací závorky. Takže třeba `sed -r 's/(.)(.)/\2\1/'` na každém řádku prohodí první dva znaky.

Pamatujte si, že pokud `sed` na řádku nenajde žádný výskyt hledaného výrazu, vypíše řádek beze změny.

Také je potřeba si uvědomit „žravost“ některých operací. Znaky `*` i `+` jsou „nenažrané“. To znamená, že pokud by si `sed` měl vybrat mezi více podřetězci, které by přiřadil do oné hvězdičky, tak vybere ten nejdelší z nich. Přednost ve žravosti má dřívější `*`/`+`. Toho si můžete všimnout na čtvrtém řádku, kde byly troje uvozovky, že ty první byly polknuty do `.*` na začátku.

Podobně je žravý i výběr z více možností – bere první variantu, která se na dané místo hodí, takže `(.|..)` vždy uloží do `\1` jeden znak (a druhá půlka závorky je tedy zbytečná), kdežto `(..|.)` uloží do `\1` dva znaky, pokud to jenom trochu jde.

Pravidlo číslování u otevírací závorky se hodí u vnořených závorek:

```
sed -r 's/^(ab*cd(ef|gh)i+j)$/\1: \2/'
```

připíše za každý řádek, pokud je v onom poměrně složitém tvaru, dvojtečku a `ef` nebo `gh`...

Úkol 2 [4b]: V češtině je typografickou chybou napsat na konec řádku jednopísmennou předložku nebo spojku, výjimkou je spojka a, pokud se před ní nevyskytuje nějaké interpunkční znaménko (tečka, čárka, závorka, ...).

Různé programy to řeší různě, v TEXu se za předložku místo mezery vkládá vlnka (`~`). Napište výraz pro `sed`, který zařídí korektní „ovlnkování textu“ (případ, kdy je předložka/spojka přímo na začátku nebo na konci řádku, řešit nemusíte).

Tyhle „odkazy zpátky“ (*backreferences*) můžeme ale používat i ve vyhledávaném výrazu. `s/^(.*)\1$/\1/` tedy najde všechny řádky, na kterých je řetězec dvakrát za sebou, a nahradí tyto výskyty jen jedním opakováním řetězce. Na vstup

```
aa
ab
abeceda
bagrbagr
```

odpoví

```
a
ab
abeceda
bagr
```

Mimochodem, například také slova `sentence`, `sequence` nebo `statement` jsou nahrazovacími výrazy…

Úkol 3 [4b]: Napište vyhledávací výraz, který ze slovníku „vygrepuje“ všechna slova, která jsou validními nahrazovacími výrazy pro `sed`. I `grep` umí backreference (stejně jako `sed`). Bonusové 4 body dostane ten, kdo vyrobí takový výraz bez backreferencí (Chuck Norris je má jisté).

Výraz musí být samozřejmě nezávislý na použité abecedě, takže není správným řešením vygenerovat pro každý možný oddělovač separátní výraz…

Ve slovníku je na každém řádku právě jedno slovo (a vlevo a vpravo od něj nejsou žádné mezery).

Místo jednoho výrazu můžete použít až tři volání příkazu `grep` spojená za sebou rourou.

Nyní si zavedeme fiktivní programovací jazyk, budeme mu říkat třeba „ReProg“. Programem v ReProgu bude posloupnost nahrazovacích výrazů pro `sed`.

Nad vstupními daty se postupně spustí každý z výrazů. Když program doběhne na konec, zjistí ReProg, jestli nějaký z výrazů měnil data. Pokud ano, spustí se celý program od začátku znovu; pokud ne, data se prohlásí za výstup a program skončí.

Takový vzorový program je třeba

```
statement
sentence
samaria
```

Když mu předhodíme sebe sama jako vstup, pak po prvním průběhu budeme mít

```
steriencement
sencence
serienmaria
```

Po druhém průchodu máme

```
steriencerienc
sencence
serienriemenria
```

a tak dále, až pátý průchod data nezmění a výstupem je

```
steriencerienc
sencence
serienrierienrierien
```

Úkol 4 [7b]: Napište program pro ReProg, který na vstupu na každém řádku dostane dvě přirozená čísla ve dvojkovém zápisu oddělená mezerou a na výstupu bude na každém řádku to větší z nich. Například pro jednořádkový vstup `1011 110` bude výstup `1011`.

Za zmínku ještě stojí, že i `sed` podobné iterování umí, byť na jiném principu a s trochu jinou syntaxí. To už je ale trochu jiný příběh, třeba na nějakou soustředkovou přednášku.

A kdybyste se báli, že se na soustředko nedostanete, přečtěte si třeba manuálovou stránku (`man sed`), nebo dosti obsáhlou dokumentaci na internetu.[5]

To je pro tentokrát vše, v závěrečném dílu si ukážeme temnou a mocnou sílu jedné zajímavé mutace regexů – těch v Perlu. Prý se v nich dá napsat výraz, který nahradí dvojici čísel na vstupu za jejich podíl…

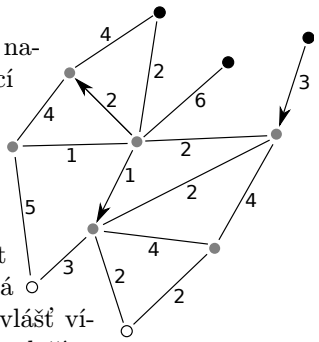
[5] `http://www.gnu.org/software/sed/manual/sed.html`

Recepty z programátorské kuchařky

Ukážeme si uměle znějící úlohu, kterou posléze zmatematizujeme, vyřešíme a dokážeme vlastnosti řešení. Nakonec přijdou četná užití, která ozřejmí, proč jsme se snažili.

Látka je lehce pokročilá, takže vězte, že budete potřebovat znát grafy.

Uměle znějící úloha



Ruský petrobaron vlastní ropná naleziště na Sibiři a trubky vedoucí do Evropy. Trubky vedou mezi nalezišti, uzlovými body a koncovými body, kde ropu přebírají odběratelé.

Každá trubka může a nemusí mít definováno, kterým směrem jí má téci ropa. Pro každou trubku zvlášť víme, kolik nejvýše jí za hodinu protlačíme.

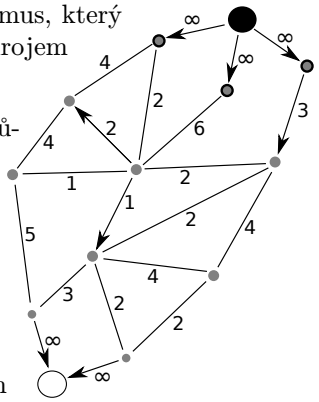
Naleziště jsou bezedná a mohou posílat neomezená množství ropy. Odběratelé také dokáží neomezená množství ropy z koncových bodů odebírat. Petrobaron čelí problému, jak protlačit danou distribuční sítí co nejvíce ropy za hodinu ze zdrojů k odběratelům.

Zapeklité je to hlavně proto, že v uzlových bodech nelze ropu hromadit, ani pálit – rozhodně tedy nejde bez rozmyslu přikázat, ať každou trubkou teče maximum, protože bychom poškodili cenná zařízení a v hnusu labutě zahubili.

Zmatematizování

V zadání vidíme graf, který obsahuje orientované i neorientované hrany, kde je nějaká podmnožina vrcholů označená jako zdroje a jiná jako… říkejme tomu třeba stoky.

Abychom měli situaci jednodušší, zbavíme se hned na úvod mnohočetnosti zdrojů a stoků. Přikreslíme si dva nové vrcholy – z nadzdroje budeme posílat ropu do všech zdrojů, do nadstoku budeme posílat ropu ze všech stoků. Kapacitu přikreslených hran pak nastavíme na nekonečno.



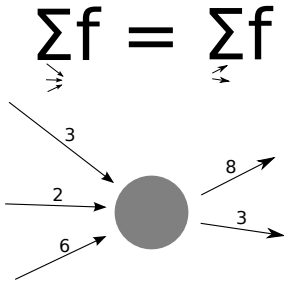
Teď nám stačí vymyslet algoritmus, který řeší problém s právě jedním zdrojem a právě jedním stokem.

Každý vstup totiž popsaným způsobem převedeme, pošleme ho algoritmu a z výstupu prostě jen odstraníme dva přidané vrcholy a připojené hrany.

Podobně se zbavíme neorientovaných hran.

Každou takovou hranu v každém zadání změníme na dvojici protisměrných orientovaných hran se stejnou kapacitou. V algoritmu pak už můžeme počítat jen s hranami orientovanými.

Dostáváme se nyní k nejdůležitějšímu – podmínkám na hledaný tok.



Na vstupu dostáváme ohodnocení hran nezápornými čísly a naším úkolem je sestavit jiné ohodnocení těch samých (všech) hran.

Je důležité, aby se nám to nepletlo – ohodnocení ze vstupu se říká kapacita a značí se $c(e)$, konstruované ohodnocení se jmenuje tok a říkáme mu $f(e)$.

Konstruované ohodnocení maximalizujeme, ale omezuje nás kapacita a Kirchhoffův zákon.

Tak budeme říkat podmínce na to, že součet toku na hranách, které do vrcholu vstupují, musí být stejný jako součet toku na hranách, které z vrcholu vystupují. Máte-li rádi fyziku nebo berete-li školu vážně, důvod k takovému pojmenování jistě chápete.

Formálně ony dvě podmínky vypadají takto:

$$\forall e \in E : f(e) \leq c(e)$$
$$\forall v \in V \setminus \{z, s\} : \sum_{\vec{uv} \in E} f(\vec{uv}) = \sum_{\vec{vu} \in E} f(\vec{vu})$$

Kirchhoffova podmínka se samozřejmě netýká ani zdroje, ani stoku – tam nám naopak jde o to ji co nejvíce porušit. Velikost toku je nejsnazší měřit na nich. Budeme ji definovat jako rozdíl mezi součtem odtoků a součtem přítoků ve zdroji.

K zamyšlení:

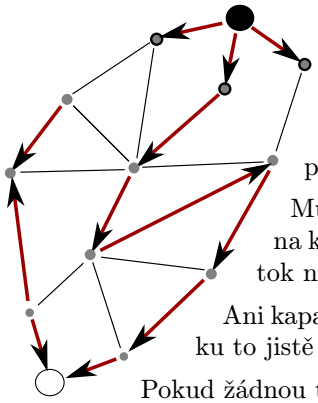
- Nastavit ohodnocení hrany (kapacitu) na skutečné nekonečno v našem programovacím jazyce nemusí jít. Pak se to řeší tím, že se zvolí dostatečně velké číslo. Jak co nejmenší, ale stále bezpečné, rychle ze zadání určit? Stejný problém se řeší třeba v Dijkstrově algoritmu, ale i ve spoustě dalších.
- Neorientované hrany, neboli obousměrné trubky, si zaslouží podrobnější rozbor, než jaký jsme jim věnovali v textu. Jak spolehlivě převedeme řešení algoritmu do původní sítě?
- Vymysleli jsme, jak vyřešit více zdrojů a stoků a jak ošetřit obousměrné trubky. Co kdyby bylo v zadání omezení na průtok vrcholy?
- Umíte dokázat, že je absolutní hodnota rozdílu přítoků a odtoků stejná na zdroji i na stoku? Tedy že bychom mohli velikost toku stejně tak dobře měřit i na stoku?

Řešení

Problém je velmi studovaný a k jeho řešení existují dva velké přístupy, které jsou humorně protikladné. Ten první vezme nulový tok a opatrně ho zlepšuje. Druhý si napíská veliké ohodnocení hran, které ani tokem není, a pak ho opravuje.

Předvedeme si onen první způsob a algoritmus, který se podle svých autorů jmenuje Fordův-Fulkersonův. Bude se nám odteď hodit tvářit se, jako že mezi každými dvěma vrcholy vede oběma směry hrana. Tam, kde ze vstupu nepřišla, si domyslíme jednu s nulovou kapacitou.

Představme si graf, na kterém počítáme tok a dejme tomu, že už nějaký tok máme – třeba prázdný. Představme si, že jsme ropný magnát a každý rozdíl mezi kapacitou potrubí a jejím využitím (tokem) nás stojí miliony dolarů. Už jsme se smířili s tím, že každá trubka nemůže být využita na maximum, ale ... zkusme si vyznačit ty hrany, kde $c(e) \neq f(e)$.

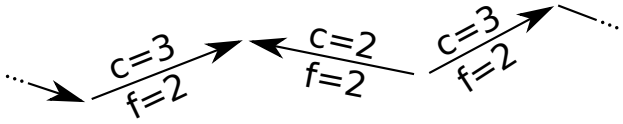


Co když existuje cesta z nadzdroje do nadstoku, která vede pouze po takových hranách?

Můžeme vzít minimum z rozdílů na každé hraně a o toto číslo navýšit tok na každé z nich!

Ani kapacitní, ani Kirchhoffovu podmínku to jistě nepoškodí.

Pokud žádnou takovou cestu nevidíme, znamená to, že tok vylepšit nejde? Ne úplně. Představte si následující situaci:



Copak nejde zlepšit? Jde! Není na to první pohled úplně jasné, ale můžeme zlepšovat výsledný tok i tím, že ho na protisměrné části cesty snížíme. Samozřejmě však nesmíme nastavovat tok záporný.

(Je smutné, že si teď trochu kazíme grafovou terminologii – co je to za cestu v orientovaném grafu, která nemusí respektovat orientaci hran?)

Takže jaká je přesně podmínka pro „vyznačení“ hrany $\vec{uv}$? Nastává $f(\vec{uv}) < c(\vec{uv})$ nebo $f(\vec{vu}) > 0$. Potom ji lze zlepšit o $c(\vec{uv}) - f(\vec{uv}) + f(\vec{vu})$.

Hledání všech vhodných („zlepšujících“) cest tedy můžeme dělat prostým prohledáváním do šířky přes vyznačené hrany. Budeme to dělat opakovaně znovu a znovu, až žádnou takovou nenajdeme, a pak vrátíme získaný tok jako výsledek.

Analýza algoritmu

Správnost

Zavolali jsme algoritmus na prázdný tok, ten ho zlepšil do situace, ve které neexistuje zlepšující cesta.

Znamená tato neexistence, že je výsledný tok maximální? Opačná implikace je jasná – maximální tok zlepšit žádným způsobem nepůjde, takže ani přes zlepšující cestičky.

Když zkusíme algoritmus pustit na graf, kde už žádná taková cesta není, můžeme si poznamenat všechny vrcholy, kam jsme se pomocí prohledávání zlepšitelných hran ještě dostali. Tato množina bude jistě obsahovat zdroj (tam jsme začali) a jistě nebude obsahovat stok (to by existovala zlepšující cesta).

Na hranách mezi touto množinou a jejím doplňkem nemůžeme zlepšovat, jinak by se po nich náš program pustil dál a množinu vrcholů, kam se dostal, by rozšířil. Všechny hrany směřující ven tedy mají $f(e) = c(e)$, pro všechny hrany směřující dovnitř platí $f(e) = 0$.

Tyto hrany tvoří *řez* naším grafem. Dovolám se v tuto chvíli na vaši intuici – tok nemůže být větší než libovolný řez. Z toho už dostáváme, že náš algoritmus našel tok maximální, protože našel také řez, který zaručuje, že nemůže existovat tok větší.

Formálnější předvedení najdete ve skriptíčkách z kombinatoriky.[6]

[6] `http://kam.mff.cuni.cz/~valla/kg.html`

Časová složitost

Je možné dobu běhu omezit počtem vrcholů a hran? Výše uvedeným postupem na grafu s celočíselnými kapacitami každou nalezenou cestou zvýšíme tok alespoň o jednotku, takže program nebude běžet déle, než je součet všech kapacit. Ale to není moc uspokojivý odhad, protože záleží na ohodnocení.

Pokud budeme hledat cesty skutečně prohledáváním do šířky, bude počet kroků v $\mathcal{O}(nm^2)$, protože se dá ukázat, že se hrany, které při zlepšování cesty tvoří minimum, postupně vzdalují od zdroje. Pak máme $\mathcal{O}(m)$ času k nalezení cesty a m hran, které se nejvýše n-krát mohou vzdálit. Že to tak skutečně je, je lehce zdlouhavé intelektuální cvičení. Nechat si prozradit postup můžete třeba v druhém vydání Introduction to Algorithms na straně 662.

O vylepšení daného postupu si můžete přečíst v záznamu[7] z jedné Medvědovy přednášky předmětu ADS2, ukázka druhého přístupu k řešení hledání maximálního toku je na záznamu[8] jejího pokračování.

K zamyšlení:

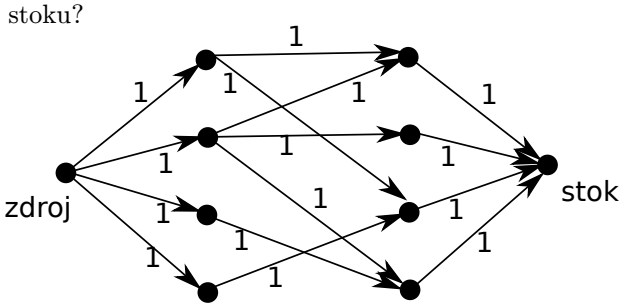
- Důležitou vlastností algoritmu je, že když dostane celočíselné kapacity, vrátí celočíselný tok. Bude se nám to hodit v aplikacích. Dokážete to?
- Rozdíl mezi Fordem-Fulkersonem, který hledá cesty obecným způsobem, a takovým, který to dělá prohledáváním do šířky, je ze složitostního hlediska docela velký, a proto se tomu druhému občas říká Edmondsův-Karpův. Najděte malý graf a nevhodnou posloupnost cest, která způsobí, že F-F poběží skutečně v závislosti na velikosti kapacit.
- Můžete dokonce zkusit využít zlatého řezu k nalezení grafu s reálnými kapacitami, na kterém F-F pro danou (nešikovnou) posloupnost cest nikdy neskončí.
- Skončí algoritmus v konečném čase, jsou-li kapacity čísla racionální?

Užití

Párování v bipartitních grafech

Máme-li za úkol najít na plese co nejvíce tanečnicím tanečníka, kterého znají, stojíme před zásadním a nelehkým úkolem.

Co třeba postavit na základě známosti bipartitní graf mezi partitou tanečníků a partitou tanečnic, přidat zdroj za kluky a stok za holky, tyto k nim připojit hranami s jednotkovou kapacitou, hranám v bipartitním grafu také nastavit jednotkové kapacity a nakonec všechno zorientovat směrem do stoku?



Maximální celočíselný tok, který na tomto grafu získáme, nám hrany bipartitního grafu rozdělí na nevybrané s tokem 0 a vybrané s tokem 1. Můžou vybrané hrany sdílet tanečníka? Těžko, když do něj teče nejvýše jednotkový tok a musí platit Kirchhoffův zákon. A podobně s tanečnicemi.

Vybrané hrany nám proto vytvoří párování. A protože jsme našli maximální tok, jde o párování největší. Kdyby existovalo párování větší, dokázali bychom z něj zvětšit tok.

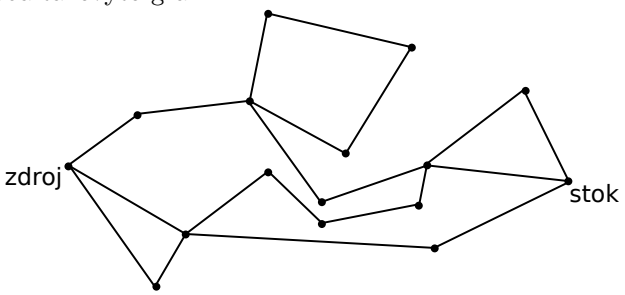
Hledání hranově a vrcholově disjunktních cest

Chceme-li se v grafu G dostat z vrcholu u do vrcholu v, může nás zajímat (třeba kvůli spolehlivosti, s jakou se umíme dopravit do cíle), kolik mezi nimi existuje cest, které

- nesdílí hrany, nebo
- nesdílí vrcholy. (Tato podmínka je silnější. Když dvě cesty nesdílí vrcholy, nesdílí hrany.)

Oba tyto problémy lze převést na hledání maximálního toku. V obou případech nastavíme u jako zdroj a v jako stok. V prvním případě nastavíme jednotkové kapacity všem hranám, v druhém navíc všem vrcholům.

Ford-Fulkerson nastavil některým hranám jednotkový tok, některým nulový. Nulové nyní z grafu vyhodíme. Pokud jsme hledali hranově disjunktní cesty, můžeme nyní získat třeba takovýto graf:



Jak z něj vykřesat kýžený výsledek? Začneme procházet ze zdroje zbylé hrany. Vždy, když se dostaneme do vrcholu, ve kterém už jsme v tom samém průchodu byli, vyhodíme z grafu všechny hrany cyklu, který jsme tímto objevili. (Hodnota toku se tím nezmění.)

Průchodem grafu se vždy můžeme dostat až do stoku (všude jinde budeme moci podle Kirchhoffova zákona jít dál – dost to připomíná úvahu o eulerovských tazích)[9] a protože jsme mezitím agilně odstraňovali cykly, dostali jsme cestu. Vrátíme ji jako jeden výsledek, smažeme její hrany a pokud ještě tok není nulový, pokračujeme dál.

Počet cest je tedy velikost toku. Podle Mengerovy věty je navíc počet hranově/vrcholově disjunktních cest roven stupni hranové/vrcholové souvislosti grafu – máme tedy nyní algoritmus, který ji najde.

K zamyšlení:

- Úvaha nebyla naprosto přímočará kvůli cyklům v nalezeném toku. Říká se jim cirkulace. Je jasné, že v případě hledání hranově disjunktních cest vzniknout mohou. Co v případě vrcholově disjunktních, tedy v situaci, kdy jsme omezili tok vrcholy?
- Nepracuje náhodou neupravený Edmondsův-Karpův algoritmus rychleji, pokud je graf, jak jsme teď opakovaně viděli, ohodnocený toliko nulami a jedničkami?

[7] `http://mj.ucw.cz/vyuka/0910/ads2/2-dinic.pdf`
[8] `http://mj.ucw.cz/vyuka/0910/ads2/3-goldberg.pdf`
[9] `http://ksp.mff.cuni.cz/viz/kucharky/eulerovske-tahy`

23-3-1 Úsporný kořen

Řešitelé, kteří mají dobrou grafovou intuici nebo dostatečně naposloucháno, si uvědomili, že jde dokázat, že kýžené vrcholy najdou uprostřed nejdelší cesty stromu. Jan Bok si dobře všimnul, že v dávné úloze 18-1-3 Keřík už jsme dokonce obecnější variantu problému nejdelší cesty ve stromu řešili.

Vezmeme zavděk algoritmem, který takové pozorování nevyužívá. Bude se zakládat na opakovaném obírání stromu o listy. Nejdřív ale několik otázek:

Může být list stromu na alespoň třech vrcholech úsporný kořen? Nemůže, protože soused takového listu je ke všem ostatním vrcholům o jednotku bližší (každá cesta z listu k dalšímu vrcholu totiž vedla přes něj), takže bude mít o jednotku menší hlobuku.

Změní se množina úsporných kořenů odstraněním všech listů stromu na alespoň třech vrcholech? Ne, protože takovou operací zmenšíme hloubku všech zbylých vrcholů právě o jedničku – vrcholy s minimální hloubkou zůstanou tytéž.

Proč právě o jedničku? Hloubka každého vrcholu je dána vrcholy, které jsou od něj nejdál. To ale musí být listy, jinak by šlo onu vzdálenost měřící cestu protáhnout a hloubku zvětšit.

Tím, že odstraníme všechny listy, tedy odstraníme všechny důvody, proč by nemohla být hloubka o jednotku menší. O víc to být nemůže, protože sousedi odstraněných nejvzdálenějších listů svědčí o existenci cesty o jednotku kratší.

Je dobré si rozmyslet, kde argumentace selhává na stromech, které ani tři vrcholy nemají.

Teď už je zřejmá správnost algoritmu, který vrací výsledek sama sebe pro strom obraný o všechny své listy, je-li spouštěn na stromu s třemi a více vrcholy. Pro strom na jednom či dvou vrcholech je množina úsporných kořenů rovna množině vrcholů.

Algoritmus skončí, protože každý strom na alespoň dvou vrcholech obsahuje alespoň dva listy (jsou to třeba konce nejdelší cesty).

Abychom se vešli do lineární časové složitosti, předpočítáme si stupeň (počet sousedů) každého vrcholu a při každém odtrhávání listů si jej zaktualizujeme. Budeme si také udržovat seznam listů grafu – vrcholy z něj zanikají odtrháváním a přibývají snížením stupně na jednotku.

Odůvodněním lineárnosti pak budiž to, že odstranění každého vrcholu nám trvá konstantně času – nezapomeňme, že odstraňujeme listy, takže aktualizace seznamu sousedů a stejně tak stupně se týká jen jediného souseda tohoto odstraňovaného.

Vzorový program je na konci letáku.

Lukáš Lánský

23-3-2 Nejkratší cesta přes oceán

Nejprve se podíváme na to, jak vypadá ona nejkratší úsečka, kterou hledáme. Její krajní body leží na obvodech zadaných mnohoúhelníků, takže buďto na nějaké straně, nebo v nějakém vrcholu mnohoúhelníka.

Navíc, když si po chvíli uvědomíme, že řešením určitě bude kombinace vrchol-strana, nebo vrchol-vrchol, tak už není žádný problém vyzkoušet všechny takové kombinace v čase $\mathcal{O}(N^2)$. Může existovat i řešení strana-strana, ale pak existuje i jiné stejně dobré...

S trochou štěstí, třídění a binárního vyhledávání se dá takový algoritmus zrychlit až na $\mathcal{O}(N \log N)$, ale to stále není žádná sláva.

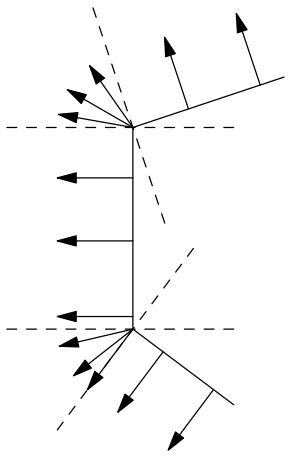
Naservírujeme si tedy trochu geometrických důkazů a vykoukáme z nich algoritmus ještě výrazně rychlejší.

První případ. Pokud je řešením kombinace strana-vrchol, pak ona hledaná úsečka bude na příslušnou stranu kolmá. Důkaz sporem. Uvažme stranu $\overline{XY}$ jednoho mnohoúhelníka, uvnitř které leží bod B (různý od X i Y); bod A je vrcholem druhého mnohoúhelníka. Hledanou úsečkou budiž $\overline{AB}$ a úhel ABY nechť není pravý.

Pak nechť B' je pata kolmice z bodu A na přímku XY. Pokud B' leží mezi X a Y, pak rozhodně $|AB'| < |AB|$, a tedy máme kratší úsečku a $\overline{AB}$ nebyla řešením.

Pokud by bod B' padl mimo $\overline{XY}$, pak na přímce XY leží body určitě v pořadí B', X, B, Y, nebo B', Y, B, X, přičemž vzdálenost od bodu A v tomto pořadí roste ($\overline{AB'}$ je nejkratší a $\overline{AY}$, resp. $\overline{AX}$ je nejdelší). Speciálně tedy jedna z $\overline{AY}$ a $\overline{AX}$ musí být kratší než úsečka $\overline{AB}$, která tedy není řešením. Spor. *QED*

Podobnou úvahou zjistíme, že pokud je řešením kombinace vrchol-vrchol, pak hledaná úsečka musí s oběma přilehlými stranami svírat alespoň pravý úhel, jinak na ni můžeme aplikovat argument z předchozích odstavců.



Takže pro každou hranu máme jen jeden směr, ve kterém z ní může vést hledaná úsečka, a pro každý vrchol interval směrů.

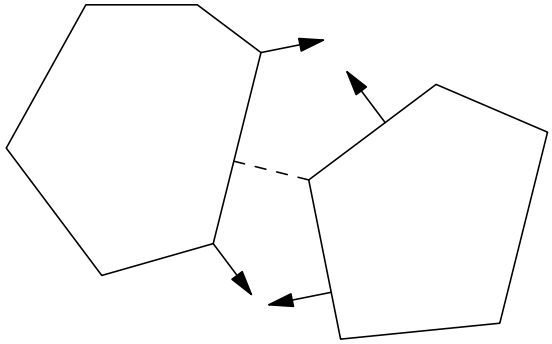
Co víc, mnohoúhelníky jsou konvexní, takže když si řekneme libovolný směr (úhel), tak nalezneme jen jedno místo na každém mnohoúhelníku, pro které tento směr připadá v úvahu.

Navíc jsme dostali ony mnohoúhelníky zadané jako body v pořadí na obvodu, takže můžeme jednoduše v lineárním čase postupně projít všechny možné směry.

Lze si to také představit tak, že máme dvě rovnoběžky, které otáčíme každou okolo jednoho z mnohoúhelníků stejným směrem (tak, abychom nepřeskočili žádný vrchol), a vždycky si ukládáme, kterého vrcholu se zrovna která ze přímek dotýká. Tak je také implementován vzorový program.

Jak zjistíme, že právě procházíme okolo řešení? Pokud je správným řešením kombinace vrchol-strana (A-BC), rozhodně se v jednu chvíli stane, že na jednom mnohoúhelníku máme zrovna vybraný bod A a na druhém přecházíme z B do C.

Navíc pro správné řešení jako jediné platí, že ABC je ostroúhlý trojúhelník, který se nepřekrývá se zadanými mnohoúhelníky. Důkaz je jednoduchý – od správného řešení se rozchází odpovídající si vrcholy na různé strany, viz obrázek.



Pro případ, že řešení je vrchol-vrchol, si ještě ukládáme vzdálenosti mezi projitými dvojicemi vrcholů. Pokud tedy doběhne cyklus bez toho, abychom vypsali výsledek a skončili, je správným řešením nalezené minimum vrchol-vrchol.

Lineární řešení (C):
`http://ksp.mff.cuni.cz/tasks/23/2332-1.c`

Čas je tedy $\mathcal{O}(N)$, paměť taktéž. Vyřešili jsme tedy úlohu tak rychle, jak rychle umíme načíst vstup.

Existuje drsnější řešení, které využívá modifikaci půlení intervalu. Jeho kompletní popis by vystačil na samostatný článek a jeho časová složitost je $\mathcal{O}((\log A)(\log E))$, kde A a E jsou počty vrcholů mnohoúhelníků. Nám však bohatě stačilo řešení lineární.

V úloze se masivně používá analytická geometrie a vektorový počet. Za zmínku stojí několik použitých faktů:

- Bod se dá považovat za vektor.
- Skalární součin dvou vektorů a, b je roven $|a||b|\cos\varphi$, tedy je kladný, pokud svírají ostrý úhel, záporný pro tupý úhel a nula pro pravý úhel φ.
- Normálový vektor a_N je kolmý na vektor a
- Skalární součin vektoru a a normálového vektoru b_N je kladný, je-li vektor b „na jedné straně“ od vektoru a, jinak záporný (a pro vektory opačného směru nulový). Kladná a záporná strana závisí na definici normálového vektoru (je-li to „ten kolmý vlevo“, nebo „ten kolmý vpravo“).

Jan „Moskyto“ Matějka & Jitka Novotná

23-3-3 Skok bez padáku

Úloha má přehršel parametrů a u takových se obvykle stává, že složitost různých řešení závisí na různých parametrech. Tak si je pojďme pojmenovat:

(x_0, y_0)	počáteční pozice
h_0	výška, ze které smíme spadnout
T	počet trampolín
W	šířka (pozice nejpravější trampolíny)

Ujasnění zadání. Zadání zarytě mlčí o dvou důležitých věcech:

- Jsou souřadnice celočíselné? Nikoho z řešitelů naštěstí nenapadlo, že by nemusely být, tak to předpokládejme také. (Jinak by totiž úloha byla mnohem zákeřnější – byla by vůbec řešitelná v konečném čase?)
- Co se stane, když padáme z výšky 1? Pak by měl následovat odraz do výšky 0. A pokud spadneme na jednu z několika sousedních trampolín, můžeme po nich pak volně chodit a na kraji seskočit dolů? Raději nulové odrazy zakážeme. (Kdybychom je opravdu chtěli, náš algoritmus půjde snadno upravit, aby s nimi počítal.)

Pár pozorování pro začátek. Předně, pokud spadneme z bodu (x, y) na trampolínu (x, t), odrazíme se do výšky $y' = \lfloor (y + t)/2 \rfloor$ a odtamtud se posuneme buďto do $(x - 1, y')$, nebo do $(x + 1, y')$. Jelikož $t < y$ (trampolína leží pod námi) a nulové odrazy jsme zakázali, musí být i $y' < y$. Takže postupně padáme z čím dál tím nižších bodů.

Proto ať už se odrážíme jakkoliv, po konečně mnoha odrazech spadneme na zem (živí či mrtví; se schrödingerovsky kočkovitými parašutisty nepočítáme). Dokonce víme, že odrazů je vždy nejvýše y_0.

Rekurzivní řešení. Nejprve se podíváme na první podúlohu. Chceme tedy naprogramovat funkci, která dostane počáteční polohu (x_0, y_0) a oznámí, jaký je minimální počet odrazů, chceme-li přežít (nebo $+\infty$, pokud nemáme šanci). Tato funkce si může spočítat, která trampolína leží pod zadaným bodem, odrazit se od ní, a vyzkoušet jak posunutí doleva, tak doprava.

Každá z těchto možností zase dává nějaký bod, ze kterého budeme padat. Který si vybrat? Nevíme. Tak zkusíme oba. Pro každý se zavoláme rekurzivně a zjistíme, která možnost dává menší počet odrazů. O 1 větší počet pak prohlásíme za svůj výsledek. Jak už víme, stále klesáme, takže výpočet se nemůže zacyklit.

Zbývá ošetřit triviální případ, totiž ten, že už pod námi žádná trampolína neleží. Pak podle toho, zda už jsme v bezpečné výšce, vrátíme buď 0 nebo $+\infty$.

Toto je jistě funkční řešení, bohužel ale poněkud hlemýždí – pro každý odraz se dvakrát rekurzivně voláme, takže pro nejvýše y_0 odrazů dostáváme exponenciální časovou složitost $\mathcal{O}(2^{y_0})$. (Náš odhad počtu odrazů je poněkud přemrštěný, ale i s tím správným, který časem dokážeme, vyjde exponenciála.)

Jak neopakovat výpočty. Čím všechen ten čas trávíme? Inu, počítáme pořád dokola totéž. Vstupem naší funkce je totiž dvojice souřadnic a různých dvojic existuje pouze $W \times y_0$.

Algoritmus tedy můžeme vylepšit tím, že si pořídíme pole („blbenku“) a budeme si v něm pamatovat, pro které počáteční polohy už známe výsledek a jaký je. Před každým voláním funkce se tam podíváme a pokud už hodnotu známe, použijeme ji. Jinak volání provedeme a výsledek si poznamenáme. Tím celkový počet volání snížíme na $\mathcal{O}(Wy_0)$.

Jak najít trampolínu. V předchozím rozboru jsme poněkud zamluvili, že potřebuje pro zadanou polohu zjistit, jaká je nejbližší nižší trampolína. Na to by se dalo jít všelijak chytře, třeba si souřadnice trampolín setřídit lexikograficky a pak v nich půlením intervalu hledat.

My na to ale půjdeme jinak: předpočítáme si „navigační tabulku“ tvaru $W \times y_0$, která nám pro každý bod řekne, jak hluboko pod ním je trampolína.

Nejprve tabulku vyplníme nulami, jen na pozice trampolín napíšeme jedničky. Pak pole projdeme zespoda nahoru a doplňujeme hodnoty. Jedničky zůstanou jedničkami, pro každou nulu se podíváme, co je pod ní. Pokud nula, ponecháme naši nulu. Pokud něco jiného, naše hodnota bude o 1 větší. Výpočet tabulky tedy bude trvat čas $\mathcal{O}(Wy_0 + T)$.

Každý krok našeho rekurzivního algoritmu s blbenkou teď už umíme provést v konstantním čase, celý algoritmus tedy poběží v čase $\mathcal{O}(Wy_0 + T)$.

Zespoda nahoru. Rekurzi s blbenkou obvykle můžeme zjednodušit na dynamické programování. Tím myslíme, že budeme blbenku rovnou počítat zespoda nahoru – pro výpočet

každé hodnoty potřebujeme jenom hodnoty z nižších řádků, které už budeme mít spočítané.

Přesněji řečeno, označíme si $P[x, y]$ minimální počet odrazů při pádu z bodu (x, y) a budeme zespoda nahoru provádět toto:

- Pokud pod (x, y) neleží žádná trampolína, položíme
$$P[x, y] = \begin{cases} 0 & \text{pro } y \leq h_0, \\ +\infty & \text{pro } y > h_0. \end{cases}$$
- Leží-li pod (x, y) trampolína (x, t), spočítáme výšku po odrazu $y' = \lfloor y + t \rfloor$ a položíme
$$P[x, y] = \min(P[x - 1, y'], P[x + 1, y']) + 1.$$
Ptáme-li se na hodnotu mimo tabulku, použijeme $+\infty$.

S předvýpočtem navigační tabulky seběhne tento algoritmus opět v čase $\mathcal{O}(Wy_0 + T)$, ale je daleko jednodušší. Proto jsme ukázkový program psali podle něj.

Podúloha b. Druhou podúlohu, totiž stanovení všech výšek, ze kterých spadnuvše bychom přežili, získáme jako vedlejší produkt právě popsaného algoritmu. Stačí se totiž do tabulky P podívat na x_0-tý sloupec a vypsat ta y, pro něž je $P[x_0, y]$ konečné. To stihneme v čase $\mathcal{O}(y_0)$, takže nám to časovou složitost nezhorší.

Pseudopolynomiální složitost. Algoritmus, který jsme si ukázali, má takzvaně pseudopolynomiální složitost. Tím se myslí, že složitost není polynom ve velikosti vstupu, nýbrž v hodnotách čísel obsažených na vstupu. U opravdového polynomiálního algoritmu by tedy směla záviset pouze na T, nikoliv na y_0, h_0 nebo W. Případně pokud bychom (jak se často činí) měřili velikost vstupu v bitech, byla by vzhledem k velikosti vstupu polynomiální také čísla $\log y_0$, $\log h_0$ a $\log W$. Neuměli bychom najít poctivé polynomiální řešení?

Lepší odhad na počet odrazů. Především si všimneme, že naše omezení počtu odrazů číslem y_0 bylo naprosto přemrštěné. Zaměřme se na jednu trampolínu a sledujme výšky, do kterých se dostaneme po jednotlivých odrazech. Kdyby žádné jiné trampolíny neexistovaly (a dovolili bychom si na chvíli po odrazu neuhnout doleva ani doprava), dělila by se po každém odrazu výška dvěma, takže po řádově $\log y_0$ odrazech by byla nulová. Teď vrátíme ostatní trampolíny do hry a všimneme si, že tím, že jsme si na ně odskočili (doslova), jsme si při dalším návratu na naši trampolínu mohli výšku jedině zmenšit. Takže i tehdy je počet odrazů o jednu trampolínu nejvýše $\log y_0$ a celkem se proto můžeme odrazit nejvýše $(T \log y_0)$-krát.

Odstranění závislosti na W. Závislosti na parametru W (šířce mapy) se můžeme zbavit snadno. Všimneme si totiž, že se ve vodorovném směru nikdy nedostaneme dál než o T kroků od počátku. Do vzdálenosti $T + 1$ musí přeci ležet aspoň jeden sloupec bez trampolíny a ten nemáme jak přeskočit. Stačí tedy pole P v našem algoritmu omezit na velikost $(2T + 1) \times y_0$ (sloupec odpovídající souřadnici x_0 bude uprostřed) a trampolíny ležící mimo ignorovat. Tím časovou složitost zlepšíme na $\mathcal{O}(Ty_0)$.

Závislost na y_0. Ve svislém směru to nedopadne tak skvěle. Nabízí se využít toho, že během jednoho seskoku spadneme na jednu trampolínu nejvýše $(\log y_0)$-krát, takže bychom políčka nad touto trampolínou mohli rozdělit na nějaké intervaly, uvnitř kterých je $P[x, y]$ konstantní, a pamatovat si pouze hranice intervalů a jednu hodnotu pro každý z nich. Takových algoritmů se dá vymyslet vícero, ale všechny selžou na tom, že v různých seskocích může být toto rozdělení na intervaly různé, takže intervaly se mohou množit a množit, až jich nakonec bude řádově y_0.

Je tato hrozba reálná? Bohužel ano – ukážeme konstrukci vstupu, který se v těchto ohledech chová značně ošklivě. Předem varujeme, že to nebude úplně snadné; čtenář neprahnoucí po dobrodružství nechť raději přeskočí k podpisu autora na konci řešení.

Ještě tu jste? Dobrá, jdeme na to. Nejdříve si uvědomíme, jak se mění souřadnice, když se během jednoho seskoku odrážíme postupně od trampolín ve výškách $t_1, t_2, \ldots, t_n$. Už víme, že po prvním odrazu vyskočíme do výšky $y_1 = (y_0 + t_1)/2$ (zaokrouhlení s dovolením zanedbáme a pak budeme volit výšky tak, aby vždy vyšlo celé číslo). Obecně $y_i = (y_{i-1} + t_i)/2$. Pokud tyto vztahy složíme dohromady, dostaneme:

$$y_n = \frac{y_0}{2^{n+1}} + \frac{t_1}{2^n} + \frac{t_2}{2^{n-1}} + \ldots + \frac{t_n}{2^1}. \qquad (*)$$

Naše konstrukce bude vypadat tak, že si zvolíme nějaká čísla $x_1, \ldots, x_T$ a rozmístíme T trampolín na souřadnice $(n-i, x_i)$. Uvažujme, do jakých výšek nad nejpravější trampolínou se můžeme dostat při různých způsobech seskoku. Ukážeme, že možných výšek je spousta, a to dokonce i tehdy, když se omezíme na některé speciální druhy seskoků.

Kterýkoliv seskok můžeme jednoznačně popsat posloupností rozhodnutí o směru doleva/doprava po jednotlivých odrazech. Nás budou zajímat pouze seskoky složené z úseků tvaru `PPLP` nebo `PLPP`. Všimněte si, že každý takový úsek nás posune přesně o 2 trampolíny doprava, takže po $T/2$ úsecích proskáčeme celou posloupnost trampolín; celkem se při tom odrazíme $2T$-krát.

Nyní použijeme vzoreček $(*)$ a uvážíme, jak k finální výšce přispějí trampolíny v i-tém úseku. Úseky přitom očíslujeme od nultého úplně vpravo, takže i-tý úsek bude složený z trampolín $(n-2i-2, x_{2i+2})$ a $(n-2i-1, x_{2i+1})$ a navštívíme ho ve skocích s vahami (to jsou ty mocniny dvojky ve vzorečku) 2^{4i+4} až 2^{4i+1}.

Pokud ho proskáčeme způsobem `PPLP`, přispěje k součtu hodnotou

$$A_i = \frac{x_{2i+2}}{2^{4i+4}} + \frac{x_{2i+1}}{2^{4i+3}} + \frac{x_{2i}}{2^{4i+2}} + \frac{x_{2i+1}}{2^{4i+1}}.$$

Při `PLPP`:

$$B_i = \frac{x_{2i+2}}{2^{4i+4}} + \frac{x_{2i+1}}{2^{4i+3}} + \frac{x_{2i+2}}{2^{4i+2}} + \frac{x_{2i+1}}{2^{4i+1}}.$$

Rozdíl těchto dvou hodnot označíme

$$C_i = B_i - A_i = \frac{x_{2i+2} - x_{2i}}{2^{4i+2}}.$$

Finální výšku tedy můžeme vyjádřit jako součet všech A_i, ke kterému přičteme ta C_j, která odpovídají úsekům typu `PLPP`.

Uvažujme nyní nějakou obecnou posloupnost přirozených čísel $z_0, \ldots, z_K$ $(K = T/2 - 2)$. V naší konstrukci nastavíme $x_i = 0$ pro všechna lichá i, dále položíme $x_0 = 0$ a $x_{2j+2} = x_{2j} + z_j \cdot 2^{4j+2}$ pro všechna j. Navíc zvolíme počáteční výšku y_0 tak, aby byla větší než $2^{2T+1} \cdot \max_i x_i$ – tím zařídíme, že se během seskoku délky $2T$ nemůžeme dostat pod žádnou z navržených trampolín.

Touto volbou hodnot x_i jsme zařídili, že rozdíly C_i z předchozího výpočtu jsou rovny právě z_i. Jinými slovy, výšky dosažitelné zkoumanými druhy seskoků se dají napsat konstanta plus součet nějaké podmnožiny čísel z_i.

K dokončení stačí klasický trik: z mocnin dvojky $2^0, \ldots, 2^K$ se dají nasčítat všechna čísla od 0 do $2^{K+1} - 1$ (tak funguje dvojková soustava). Pro volbu $z_i = 2^i$ tedy existuje alespoň 2^{K+1} dosažitelných výšek, což je exponenciálně mnoho vzhledem k T.

Navíc počty použitých trampolín odpovídají počtu jedniček v binárním zápisu čísla, což se mění příliš rychle na to, aby intervalů mohlo být řádově méně. EPA.[10]

Vzorový program je na konci letáku.

Martin „Medvěd“ Mareš

23-3-4 Psaní písmen

Poznámka redakce: Zadavatel této úlohy do CodExu ji pozměnil. Oproti zadání v letáku a na webu byl na vstupu zadán graf explicitně rozsekaný na komponenty. Navíc zadání v CodExu vyžadovalo optimalizaci na paměť. Tomu odpovídá i zdrojový kód.

Abychom mohli úlohu vyřešit, měli bychom vědět, co jsou to eulerovské tahy a jaké podmínky splňují grafy, které je obsahují (nahlédnout můžete do našich grafových kuchařek). To, že jde obrázek nakreslit jedním tahem, znamená, že obsahuje uzavřený či otevřený eulerovský tah.

Pokud souvislý graf obsahuje pouze vrcholy sudého stupně, je v něm možno nalézt uzavřený eulerovský tah.

Co se stane, pokud neobsahuje pouze vrcholy sudého stupně? Mezi dvojici lichých vrcholů přidáme hranu (opakujeme, dokud máme vrcholy lichého stupně), takto postupně dostaneme graf, ve kterém jsou všechny vrcholy sudého stupně, tedy obsahuje uzavřený eulerovský tah.

Nyní odebereme hrany, které jsme přidali, a tento eulerovský tah se nám rozpadne na několik hranově disjunktních tahů, které vždy začínají a končí v nějakém vrcholu lichého stupně (jeden počáteční lichý vrchol a jeden koncový lichý vrchol pro každý tah), tudíž celkový počet těchto tahů je počet lichých vrcholů děleno dvěma.

Žádný vrchol lichého stupně nemůže být uprostřed tahu, tudíž tahů nemůže být méně, než jsme našli.

Stačí nám vědět, kolik takových tahů potřebujeme, není tedy potřeba je konstruovat, stačí nám určit počet lichých vrcholů (a dát si pozor na grafy bez lichých vrcholů).

Samotné řešení úlohy (provedeme pro každou komponentu souvislosti samostatně):

Potřebujeme pole délky n (počet vrcholů), při načítání si v něm udržujeme stupně jednotlivých vrcholů. Po načtení projdeme toto pole a určíme počet lichých vrcholů, který vydělíme 2. Dostaneme, kolikrát musíme zvednout pero při kreslení grafu.

Paměťová složitost: $\mathcal{O}(n)$.

Časová složitost: $\mathcal{O}(m + n)$, kde m je počet hran grafu.

Vzorový program je na konci letáku.

Martin Böhm & Lucie Mohelníková & CodEx

23-3-5 Rozházené EWD

Úkolem bylo setřídit zadaný jednosměrný spojový seznam co nejrychleji, ale v konstantní paměti, což znamená jen s předem daným počtem proměnných, bez rekurze a dalších pomocných polí, tedy pouze přepojováním původního spojového seznamu.

Určitě bylo dobrým nápadem podívat se do naší (tradiční české) kuchařky o třídění. A co s tak malou pamětí? Bublinkové třídění (bubble sort) bude zcela jistě fungovat, protože v průběhu algoritmu prohazujeme jen dva sousední prvky, což lze udělat jednoduše.

Bublinkové třídění má navíc pěknou vlastnost, že třídění již setříděných dat trvá pouze $\mathcal{O}(N)$. Jenže nejhůře a dokonce i průměrně vyjde asymptotická složitost $\mathcal{O}(N^2)$. Je to nejrychlejší možný výsledek za daných podmínek, nebo ne?

Než si řekneme řešení, uveďme si dolní odhad složitosti. Jelikož stáří záznamů EWD můžeme akorát tak porovnávat (nic o nich nevíme), platí důkaz uvedený na konci kuchařky o třídění, a tedy určitě nevymyslíme algoritmus s průměrnou složitostí lepší než $\mathcal{O}(N \log N)$.

Takový algoritmus skutečně existuje. My si ukážeme, jak modifikovat třídění sléváním (Mergesort) se zachováním složitosti v nejhorším případě i v průměru $\mathcal{O}(N \log N)$, na což přišlo i několik řešitelů. Nevylučuji však, že nepůjde upravit jiný algoritmus, i když třídění haldou ani Quicksort nejspíš převést na řešení úlohy nelze.

Jak funguje takový běžný Mergesort na třídění pole? Ten si nejprve rozdělí pole na dvě půlky, ty setřídí stejným algoritmem (zavolá se na každou rekurzivně) a pak je „slije“: odebírá vždy menší z prvků na začátku obou setříděných půlek pole a vkládá je do nového pole. Podrobnější popis opět v kuchařce.

Nyní upravíme Mergesort pro potřeby naší úlohy. Jelikož nesmíme použít rekurzi, nebudeme postupovat „odshora dolů“ (postupně půlíme data na co nejmenší části), ale „odspoda nahoru“ (spoustu malých setříděných částí sléváme postupně do jedné).

V prvním kroku se podíváme na všechny dvojice sousedních prvků (každý prvek je nejvýše v jedné dvojici), porovnáme prvky dvojice a případně je prohodíme, což v případě spojového seznamu znamená přepojení odkazů. V druhém kroku sléváme vždy dvě sousední dvojice prvků do setříděné čtveřice, v třetím dvě čtveřice do osmice…

Obecně v k-tém kroku slijeme dvě sousední části o 2^k prvcích. Až slijeme všechny prvky do jedné setříděné posloupnosti, máme vyhráno.

Často se může stát, že poslední slévaný úsek v k-tém kroku nemusí mít 2^k prvků, ale to vůbec nevadí (jeden slévaný úsek bude menší). Podobně lichý počet slévaných úseků (nemůžeme je spárovat do dvojic) ošetříme prostým ignorováním posledního úseku. V nějakém pozdějším kroku musí být tento úsek slit se zbytkem, třeba pro $2^n + 1$ prvků se bude poslední prvek slévat až v posledním kroku.

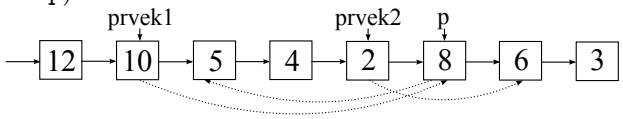
Nyní pojďme na implementaci slévání dvou setříděných úseků ve spojovém seznamu (ne nutně stejné délky) s konstantní pomocnou pamětí. Budeme si pamatovat odkaz na prvek před prvním úsekem (tedy poslední prvek již slité části) v proměnné `prvek1` a odkaz na prvek před druhým úsekem v proměnné `prvek2`.

Na začátku slévání dvou úseků nejprve posuneme odkaz `prvek2` o délku prvního úseku za odkaz `prvek1`. Abychom mohli kontrolovat, jestli v nějakém úseku nedošly prvky, vytvoříme si dvě proměnné `delka1` a `delka2`, v nichž budou počty zbývajících prvků v úsecích.

[10] Est post aves. To je něco jako „Quod erat demonstrandum.“, ale znamená to „A je po ptákách.“

Pak postupně bereme prvky ze začátku obou úseků (následníky prvku `prvek1` a `prvek2`) a menší z nich přepojíme za prvek `prvek1`. Je-li to prvek z prvního seznamu, stačí posunout odkaz `prvek1` o jeden prvek dopředu, jinak je to následník `prvek2` (označme ho `p`), který přepojíme za `prvek1` takto: následníkem `p` bude následník `prvek1`, následníkem `prvek1` bude `p`, následníkem `prvek2` bude původní následník `p`.

Jestli vás předchozí odstavec zmátl, vůbec se nedivím a raději předkládám obrázek (tečkované šipky ukazují přepojení prvku `p`):



Je vidět, že potřebujeme jen konstantně mnoho pomocné paměti. Co se týče časové složitosti, bude pro jakákoliv data $\mathcal{O}(n \log n)$, kde n je počet prvků. V k-tém kroku totiž sléváme úseky o 2^k prvcích, a bude-li $2^k > n/2$, získáme po tomto kroku celý setříděný spojový seznam. Odtud zlogaritmováním dostaneme, že stačí $\log_2 n$ kroků, přičemž v každém provedeme $\mathcal{O}(n)$ operací.

Vzorový program je na konci letáku.

Pavel Veselý

23-3-6 Výzkum veřejného mínění

Tato úloha měla spoustu možností, jak ji řešit. My si ukážeme jedno kvadratické řešení a pak řešení v čase $\mathcal{O}(N \log N)$. Nejdříve se podíváme na kvadratické řešení.

Vstupní posloupnost si načteme do dvou polí. V poli X budeme mít posloupnost, tak jak přišla na vstupu, a do pole Y uložíme posloupnost setříděnou podle velikosti.

Nyní si všimneme, že každá dvě po sobě jdoucí čísla v poli X nám určují intervaly mezi vstupními období (kde popularita klesá/stoupá) a dvě po sobě jdoucí čísla v poli Y určují intervaly hodnot, které budou mít stejnou četnost výskytů.

My tedy z každého intervalu v poli Y vezmeme libovolnou hodnotu, (například prostřední), a spočítáme, kolikrát se vyskytuje v intervalech pole X.

Nyní k řešení pracující v čase $\mathcal{O}(N \log N)$. Existuje spousta způsobů, jak na úlohu jít. My si ukážeme techniku zvanou Zametání přímkou (line sweep), pomocí které se mimo jiné dají řešit i některé geometrické úlohy.

Představme si, že se ke grafu popularity blíží přímka rovnoběžná s osou x. Tato přímka začne v minus nekonečnu, projde grafem od zdola nahoru a skončí v plus nekonečnu. Nás v každém okamžiku bude zajímat, kolikrát přímka protíná graf.

Všechny okamžiky ale testovat nemůžeme, tak se budeme věnovat jen těm, ve kterých se počet průsečíků s přímkou mění. Takovým okamžikům budeme říkat události a tyto události budeme zpracovávat v pořadí, v jakém nastanou při průchodu od zdola nahoru.

V našem případě jsou události všechny body, ve kterých se mění počet průsečíků s přímkou. Všimneme si, že tento počet se nám bude měnit pouze v lokálních maximech a minimech (tam, kde je špička). V maximu nastanou dvě události: nejdříve se počet průsečíků zmenší o jedna (došli jsme do špičky) a poté špičku opustíme a počet průsečíků se znova zmenší o jedna. Podobné budou i události u minima.

My si tedy pro každý bod vytvoříme příslušné události (pozor, u krajních bodů je pouze událost opuštění/přidání špičky) a tyto události si setřídíme primárně podle výšky a sekundárně podle jejich priority.

Priorita událostí je:

1. změna na špičku maxima
2. přidání špičky minima
3. opuštění špičky maxima
4. rozdvojení špičky u minima

Zkuste si rozmyslet, proč jsou priority událostí právě takto a v jakém případě může nastat problém, kdyby žádné priority nebyly.

Teď už jen postupně zpracováváme všechny události a po každém zpracování zkontrolujeme, jestli nejsme v maximálním počtu průsečíků. Po zpracování všech událostí vypíšeme výsledek.

Na první pohled to vypadá docela složitě, ale vlastně je to jednoduché. Viz zdrojový kód.

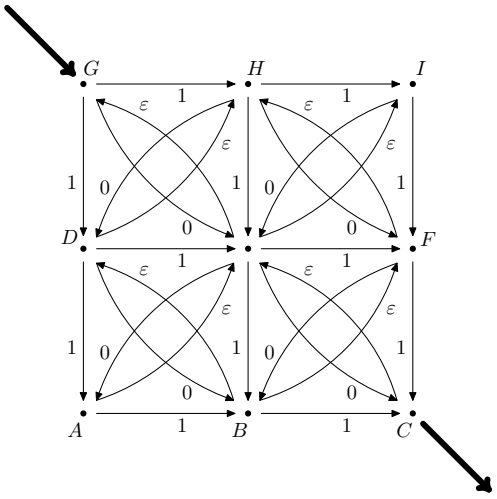
Vzorový program je na konci letáku.

Karel Tesař

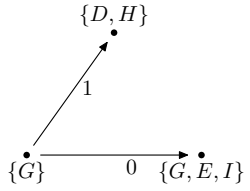
23-3-7 Automaty stokrát jinak

Třetí sérií uzavíráme seriálovou odbočku k automatům. Ještě nám chybí vysvětlit převod NKA na DKA a redukci automatu.

Nejprve si ukážeme převod NKA na DKA třeba na zadání **úkolu 1**. Označíme si jednotlivé stavy třeba písmeny A až I jako na obrázku (ten stav uprostřed je E, jen se to tam nevešlo).

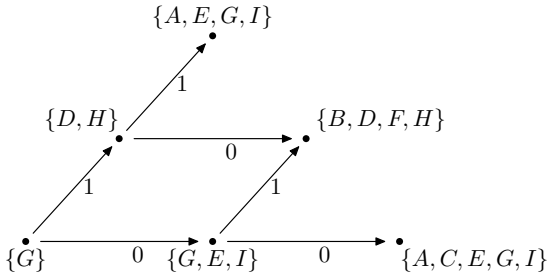


Nyní budeme konstruovat DKA, ve kterém budou jako stavy množiny stavů původního NKA. Vstupní stav je G. Z něj se můžeme dostat přečtením 1 do $\{D, H\}$ a přečtením 0 do $\{G, E, I\}$.



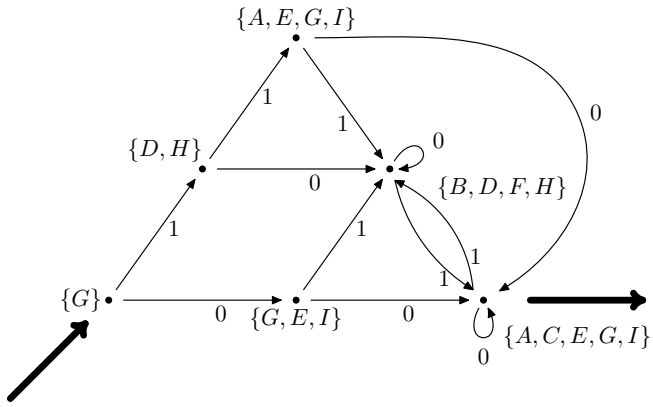
Kam se nyní můžeme dostat z $\{D, H\}$ přečtením 0? Ze stavu D jde jít do B (a pak po ε hranách do D, F a H), z H jde jít do D a F (a po ε hranách do H), tedy z $\{D, H\}$ vede hrana popsaná 0 do stavu $\{B, D, F, H\}$.

Analogicky z $\{D, H\}$ přečtením 1 dojdeme do $\{A, E, G, I\}$. Z $\{G, E, I\}$ pak vedou hrany 0 a 1 do $\{A, C, E, G, I\}$ a $\{B, D, F, H\}$.



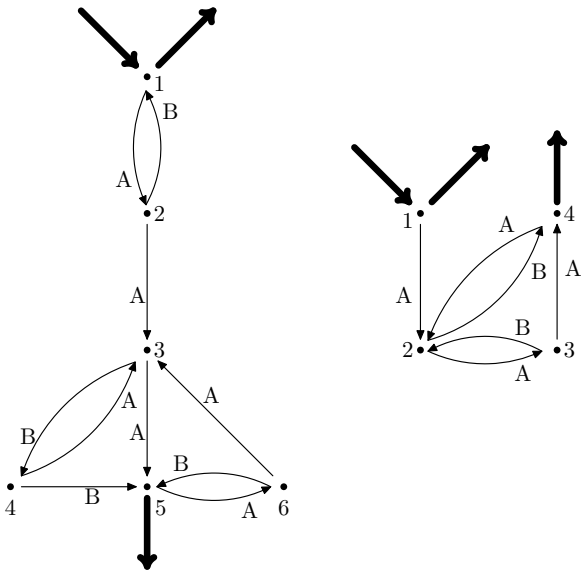
Stejným způsobem pak ještě doplníme hrany z nově vzniklých tří stavů (další už nevzniknou, ale teoreticky by mohly – výsledný DKA může mít až 2^N stavů oproti NKA s N stavy).

Výstupní stavy jsou pak všechny ty, v jejichž množinách se vyskytuje alespoň jeden výstupní stav původního NKA. V tom byl v našem případě výstupním stavem jen C, který se i ve výsledném automatu vyskytuje v jediném stavu. Ten je tedy výstupním.



Na obrázku vidíte kompletní zkonstruovaný DKA a zároveň řešení úkolu 1.

Když jste převedli oba dva výrazy z **úkolu 2** na NKA (postupem z minulé série) a touto metodou na DKA, dostali jste přibližně tyto dva automaty:



Věřili byste, že jsou ekvivalentní, tedy že přijímají stejný jazyk? Na první pohled to tak rozhodně nevypadá, ale jsou.

Jak na to přijdeme?

Ukážeme si postup zvaný „redukce automatu“, kdy nalezneme všechny stavy, které jsou ekvivalentní, a sloučíme je.

Například si můžeme všimnout, že u řešení úkolu 1 by šlo sloučit (nerozlišitelné) stavy $\{A, E, G, I\}$ a $\{G, E, I\}$. Ať přečtu cokoli, skončím na stejném místě.

	A	B	
→ 1	2	–	ω
2	3	1	α
3	5	4	α
4	3	5	α
5	6	–	ω
6	3	5	α

Zapíšeme si levý automat tabulkou. Se šipkou je vstupní stav, podtržen je výstupní. Ve sloupci vpravo jsou pak zapsány kategorie stavů – jak by automat z tohoto stavu pokračoval, kdyby na vstupu už nebyl žádný znak.

Tabulku budeme dále rozšiřovat. Předpokládejme, že je na vstupu o znak víc:

	A	B		A	B	
→ 1	2	–	ω	α	–	ω
2	3	1	α	α	ω	α
3	5	4	α	ω	α	β
4	3	5	α	α	ω	α
5	6	–	ω	α	–	ω
6	3	5	α	α	ω	α

Zjistili jsme, jak se automat chová po přečtení jednoho znaku. Vidíme, že stavy 2, 4 a 6 jsou nerozlišitelné, pokud přečteme maximálně jeden znak ze vstupu. Taktéž stavy 1 a 5.

Třetí, separátní kategorii jsme museli zavést pro stav 3, který se začal lišit od stavů 2, 4 a 6.

Provedeme ještě jeden krok a zjistíme, že se už kategorie stavů nezměnily.

	A	B		A	B		A	B	
→ 1	2	–	ω	α	–	ω	α	–	ω
2	3	1	α	α	ω	α	β	ω	α
3	5	4	α	ω	α	β	ω	α	β
4	3	5	α	α	ω	α	β	ω	α
5	6	–	ω	α	–	ω	α	–	ω
6	3	5	α	α	ω	α	β	ω	α

Jedna hezká věta říká, že jakmile se jednou nezmění kategorie stavů, nezmění se nikdy. To je docela jasně vidět, když si uvědomíte, že by se vlastně pořád dokola opakovaly stejné trojice sloupečků.

	A	B
→ ω	α	–
α	β	ω
β	ω	α

Další hezká věta říká, že poslední trojice sloupečků nám popisuje tzv. redukovaný automat, který je ekvivalentní s tím původním. Duplicitní řádky vynecháme.

Když pak provedeme totéž pro druhý automat, dostaneme podobnou tabulku.

	A	B		A	B		A	B	
→ 1	2	–	α	β	–	α	β	–	α
2	3	4	β	β	α	β	γ	α	β
3	4	2	β	α	β	γ	α	β	γ
4	2	–	α	β	–	α	β	–	α

$\longrightarrow$

	A	B
→ α	β	–
β	γ	α
γ	α	β

Automaty tedy jsou ekvivalentní, neboť jejich redukované verze jsou ekvivalentní (stačí tabulky přepísmenkovat). Proto i dva zadané regexy jsou ekvivalentní, tedy popisují stejný jazyk.

Jedno obtížně dokazatelné tvrzení říká, že pokud jsou dva automaty ekvivalentní, pak je lze zredukovat tímto postupem na stejný DKA, až na isomorfismus.

Isomorfní DKA jsou takové, že pokud správně přečíslujeme stavy jednoho z nich, tak dostaneme druhý automat. Ač se to nezdá, na problém neznáme polynomiální algoritmus, ale ani nevíme, jestli je NP-úplný.

A jaké bylo správné řešení **úkolu 3**? `10101010101201` je nejmenším násobkem devíti vyhovujícím zadanému výrazu. Bylo potřeba si všimnout, že všechny ohvězdičkované

trojice jsou násobky 3, takže přinejhorším nějakou vložíme na konec.

Na začátku byla povinně trojice 101 s ciferným součtem 2. V každé iteraci velké závorky musela být zase trojice 101 a navíc buďto 0 nebo 202. První varianta měla ciferný součet 2, druhá 6.

Krátkým rozborem případů pak došlo na to, že nejkratší násobek 3 vyhovující regexu je 10101010101. Přilepením 201 za něj pak vypadl kýžený nejmenší násobek 9.

Většina řešitelů obdržela téměř plný počet bodů, nejčastější chybou bylo opomenutí popisu, které stavy budou vstupní a které výstupní po převodu DKA na NKA. Obecně však byla vaše řešení hezká a bylo mi potěšením je opravovat.

Jan „Moskyto“ Matějka

23-4-3 Zadání (Zabugovaný program) Python

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# Čtení vstupu (zde není chyba)
# N je počet nugetů
# ceny je pole s jejich cenami
N = long(raw_input())
s = raw_input()
ceny = map(long,s.split(" "))
if len(ceny) != N:
    print "(Po)Chybný vstup"
    exit(1)
# Vstup úspěšně přečten

# Setřídíme ceny
ceny.sort()

A = 0
B = 0
LA = []
LB = []

# a rozházeme
while len(ceny) > 0:
    c = ceny.pop()
    if A > B:
        LB.append(c)
        B += c
    else:
        LA.append(c)
        A += c

# Výpis (zde není chyba)
if A == B:
    print " ".join(map(str,LA))
    print " ".join(map(str,LB))
else:
    print "Nelze spravedlivě rozdělit."
```

23-4-3 Zadání (Zabugovaný program) C

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

#define MAX 1000
int N;                 // počet nugetů
int ceny[MAX];         // ceny nugetů
int la[MAX], lb[MAX]; // co dostane kdo

// Vypíše zadané pole o N prvcích
void vystup(int *pole, int N) {
  for (int i=0; i<N; i++) {
    if (i>0)
      printf(" ");
    printf("%d", pole[i]);
  }
  printf("\n");
}

// Porovnávací funkce pro qsort()
int cmp(const void *a, const void *b) {
  return (*((int *)b) - *((int *)a));
}

int main(void) {
  // Přečteme vstup
  scanf("%d", &N);
  for (int i=0; i<N; i++)
    scanf("%d", &ceny[i]);

  // Setřídíme vstup
  qsort(ceny, N, sizeof(int), cmp);

  int a=0, ai=0, b=0, bi=0;
  for (int i=0; i<N; i++) {
    // Který zlatokop má zatím méně?
    if (a < b) { // A -> přidáme A
      la[ai] = ceny[i];
      ai++;
      a += ceny[i];
    } else {     // B -> přidáme B
      lb[bi] = ceny[i];
      bi++;
      b += ceny[i];
    }
  }

  if (b == a) { // Povedlo se rozdělit
    // Tak to vypíšeme
    vystup(la,ai);
    vystup(lb,bi);
  } else        // Nepovedlo se rozdělit
    printf("Nelze spravedlivě rozdělit.\n");
  exit(0);
}
```

23-3-1 Program (Úsporný kořen) **Python**

```
N = input()
sousedi = []
for i in range(N) : sousedi.append([])
stupne = [0] * N
listy = []

for i in range(N-1) :  # strom: n vrcholů => n-1 hran
    radka = raw_input().split()
    a = int(radka[0])-1
    b = int(radka[1])-1

    sousedi[a].append(b)
    sousedi[b].append(a)

for i in range(N) :
    stupne[i] = len(sousedi[i])
    if stupne[i] == 1 :
        listy.append(i)

aktN = N
while aktN > 2 :
    noveListy = []
    for list in listy :
        for sous in sousedi[list] :
            stupne[sous] -= 1
            if stupne[sous] == 1 :
                noveListy.append(sous)

# Prohlédněte si pozorně, co se stalo. Mazání prvku
# z prostředku seznamu tu trvá lineární čas, takže sousedy neudržuju,
# nestihnul bych to: fakt, že vrchol neexistuje, značim toliko
# nulovým stupněm, který je mu nastaven v kroku po odtrhnutí.

    aktN -= len(listy)
    listy = noveListy

# Rozmyslete si, že program za list označí
# i případný jediný zbylý vrchol.

for list in listy :
    print list+1
```

23-3-3 Program (Skok bez padáku) C

```
#include <stdio.h>

#define MAXW 100    // max. šířka
#define MAXH 100    // max. výška
#define INFTY 1000000   // toto je nekonečno

int x0, y0;     // počáteční pozice
int h0;         // bezpečná výška
int T;          // počet trampolín
int W;          // šířka

// Jak dlouho padáme, než narazíme na trampolínu
// 0=žádná pod námi není, 1=jsme přímo na ní
int tramp[MAXW][MAXH];

// kroky[x][y] = min. počet odrazů při skoku
// z políčka x,y nebo INFTY, když nelze přežít
int kroky[MAXW][MAXH];

int main(void)
{
  // Přečteme vstup a vyznačíme si trampolíny
  scanf("%d%d%d%d", &x0, &y0, &h0, &T);
  for (int i=0; i<T; i++)
    {
      int tx, ty;
      scanf("%d%d", &tx, &ty);
      // Víme, že C globální proměnné nuluje
      tramp[tx][ty] = 1;
      if (tx > W)
    W = tx;
    }

  // Předpočítáme si vzdálenosti k trampolínám
  for (int y=1; y<=y0; y++)
    for (int x=0; x<=W; x++)
      if (!tramp[x][y] && tramp[x][y-1])
    tramp[x][y] = tramp[x][y-1] + 1;

  // Dynamika zespoda nahoru
  for (int y=0; y<=y0; y++)
    for (int x=0; x<=W; x++)
      {
    kroky[x][y] = INFTY;
    if (!tramp[x][y])
      {
        // Není pod námi žádná trampolína
        if (y <= h0)
          kroky[x][y] = 0;
      }
    else if (tramp[x][y] > 1)
      {
        // Spadneme na trampolínu (x,yt)
        // Odrazíme se do bodu (x,yo)
        int yt = y - tramp[x][y] + 1;
        int yo = (y + yt) / 2;
        if (x > 0 &&
            kroky[x-1][yo]+1 < kroky[x][y])
          kroky[x][y] = kroky[x-1][yo] + 1;
        if (x < W &&
            kroky[x+1][yo]+1 < kroky[x][y])
          kroky[x][y] = kroky[x+1][yo] + 1;
      }
      }

  // Vypíšeme výstup
  if (kroky[x0][y0] < INFTY)
    {
      printf("Odrazů: %d\n", kroky[x0][y0]);
      printf("Ještě přežijeme z výšek:");
      for (int y=y0; --y >= 0;)
    if (kroky[x0][y] < INFTY)
      printf(" %d", y);
      putchar('\n');
    }
  else
    printf("R.I.P.\n");
  return 0;
}
```

23-3-4 Program (Psaní písmen) C

```
#include <stdio.h>
#include <string.h>
#define MAXV 10000
struct polozka
{
  int komponenta;
  int stupen;
} pole_vrcholu[MAXV];

int main()
{
  int vrcholy;
  int hrany;
  int max_vrchol=0;
  int v1=0;
  int v2=0;
  int komp=0;

  while (!feof(stdin))
  {
    komp++;
    char pom[2];
    pom[0]=getchar();
    if (pom[0]=='-')
    {
      scanf("%c%c\n",&pom[1],&pom[2]);
      scanf("%d %d\n",&vrcholy,&hrany);
    }
    else
    {
      ungetc(pom[0],stdin);
      scanf("%d %d\n",&vrcholy,&hrany);
    }
    for(int i=0;i<hrany;i++)
    {
      scanf("%d %d\n",&v1,&v2);
      pole_vrcholu[v1].stupen++;
      pole_vrcholu[v1].komponenta=komp;
      pole_vrcholu[v2].stupen++;
      pole_vrcholu[v2].komponenta=komp;
      if (max_vrchol<v1) max_vrchol=v1;
      if (max_vrchol<v2) max_vrchol=v2;
    }
  }
    int pole_komponent[komp+1];
    for(int i=1;i<=komp;i++)
    {
      pole_komponent[i]=0;
    }
    //printf("pocet komp: %d\n",komp);
    //printf("max_vrchol: %d\n",max_vrchol);

    for(int i=1;i<=max_vrchol;i++)
    {
      if (pole_vrcholu[i].stupen%2 ==1)
        pole_komponent[
          pole_vrcholu[i].komponenta]++;
    }
    int vystup=0;
    for (int i=1;i<=komp;i++)
    {
      vystup = pole_komponent[i]/2;
      printf("vystup: %d\n",vystup+!vystup);
    }

return 0;
}
```

23-3-5 Program (Rozházené EWD) C

```
// prvek spojového seznamu
typedef struct prvekSeznamu  {
  // záznam EWD (jen pro ilustraci, že s daty EWD se nehýbe -- v celém programu se jinak nepoužívá)
  EWD zaznam;

  // stáří záznamu (čím starší, tím vyšší číslo)
  int stari;

  // ukazatel na další prvek spojového seznamu (nebo hodnota NULL, pokud je poslední)
  struct prvekSeznamu *dalsi;

} PrvekSeznamu;

// nerekurzivní funkce, která dostane ukazatel na první prvek nesetříděného seznamu
// a vrátí ukazatel na první prvek setříděného seznamu
PrvekSeznamu *EWDSort(PrvekSeznamu *zacatek) {
  if (zacatek == NULL) return NULL; // algoritmus dostal prázdný seznam

  int delkaUseku = 1; // délka slévaných úseků
  int pocetPrvku = 0; // počet prvků spojového seznamu, spočte se v prvním kroku

  PrvekSeznamu *novyZacatek = (PrvekSeznamu *)malloc(sizeof(PrvekSeznamu));
  novyZacatek->dalsi = zacatek;
  PrvekSeznamu *prvek1; // ukazatel na poslední prvek již slité části seznamu
  // při slévání je prvek1 poslední prvek před 1. slévaným úsekem
  PrvekSeznamu *prvek2; // prvek před aktuálně zpracovávaným prvkem
  // prvek2 je poslední prvek před 2. slévaným úsekem
```

```
  do { // cyklus dle kroků algoritmu
    prvek1 = novyZacatek; // nastav ukazatele pred začátek
    prvek2 = novyZacatek;

    do { // cyklus slévání dvou úseků
      int delka1 = delkaUseku; // délka prvního slévaného úseku
      int delka2 = delkaUseku; // délka druhého slévaného úseku

      for (int i = 0; i < delkaUseku; i++) { // projdi první úsek
        prvek2 = prvek2->dalsi;
        if (delkaUseku == 1) pocetPrvku++; // v prvním kroku počítáme prvky
        if (prvek2 == NULL) break; // konec seznamu
      }

      if (prvek2 == NULL) break; // druhý úsek je prázdný, nemá cenu slévat

      // dokud nejsou slity všechny prvky v obou úsecích
      while (delka1 > 0 || (delka2 > 0 && prvek2->dalsi != NULL)) {
        // došly prvky v 2. úseku, nebo je 1. prvek 1. úseku starší než 1. prvek 2. úseku
        if (delka2 <= 0 || prvek2->dalsi == NULL
         || (prvek1->dalsi != NULL
             && prvek1->dalsi->stari > prvek2->dalsi->stari)) {
          delka1--;
          prvek1 = prvek1->dalsi; // zatřiďujeme prvek z 1. úseku, stačí tedy posunout ukazatel
        }
        else if (delka1 <= 0) { // došly prvky v 1. úseku
          delka2--;
          prvek1 = prvek1->dalsi; // je třeba posunout oba ukazatele
          prvek2 = prvek2->dalsi;
          if (delkaUseku == 1) pocetPrvku++;
        }
        else { // je třeba přehodit 1. prvek 2. úseku před 1. prvek 1. úseku
          PrvekSeznamu *usek1 = prvek1->dalsi;
          prvek1->dalsi = prvek2->dalsi;
          prvek2->dalsi = prvek2->dalsi->dalsi;
          prvek1->dalsi->dalsi = usek1;
          prvek1 = prvek1->dalsi;
          if (delkaUseku == 1) pocetPrvku++;
          delka2--;
        }
      }

      prvek1 = prvek2;

    } while (prvek2->dalsi != NULL); // dokud nejsme na konci seznamu

    delkaUseku = delkaUseku * 2; // zdvojnásob délku slévaných úseků

  } while (delkaUseku < pocetPrvku); // dokud jsme neslili všechny prvky

  return novyZacatek->dalsi;
}
```

23-3-6 Program (Výzkum veřejného mínění) **C++**

```
#include <cstdio>
#include <cstdlib>
#include <algorithm>

using namespace std;

struct Bod {
  double hodnota;
  int typ;

  bool operator<(const Bod &b) const {
    return (hodnota < b.hodnota || (hodnota == b.hodnota && typ < b.typ));
  }
};

int n;
double hodnoty[100100];
Bod udalosti[200200];
int pocet;

int main(void) {
  scanf("%d", &n);
  for (int i=0; i<n; i++)
    scanf("%lf", &hodnoty[i]);
  if (n==1)
    printf("%lf\n", hodnoty[0]);
  else {
    // vytvoření událostí
    pocet = 0;
    // první bod
    udalosti[pocet++] = (Bod){hodnoty[0], (hodnoty[0] < hodnoty[1])?2:3};
    // poslední bod
    udalosti[pocet++] = (Bod){hodnoty[n-1], (hodnoty[n-1] < hodnoty[n-2])?2:3};
    for (int i=1; i<n-1; i++) {
      // maximum
      if (hodnoty[i] > hodnoty[i-1] && hodnoty[i] > hodnoty[i+1]) {
        udalosti[pocet++] = (Bod){hodnoty[i], 1};
        udalosti[pocet++] = (Bod){hodnoty[i], 3};
      }
      // minimum
      else if (hodnoty[i] < hodnoty[i-1] && hodnoty[i] < hodnoty[i+1]) {
        udalosti[pocet++] = (Bod){hodnoty[i], 2};
        udalosti[pocet++] = (Bod){hodnoty[i], 4};
      }
    }
    sort(udalosti, udalosti + pocet);
    // zpracování událostí
    int max = 0;
    double h = 0.0;
    int pruseciky = 0;
    for (int i=0; i<pocet; i++) {
      if (pruseciky > max) {
        max = pruseciky;
        h = (i>0)?((udalosti[i].hodnota + udalosti[i-1].hodnota)/2):(udalosti[i].hodnota);
      }
      switch(udalosti[i].typ) {
        case 1: pruseciky--; break;
        case 2: pruseciky++; break;
        case 3: pruseciky--; break;
        case 4: pruseciky++; break;
      }
    }
    printf("Procenta %lf%%, pocet pruseciku %d\n", h, max);
  }
  return 0;
}
```

Výsledková listina dvacátého třetího ročníku KSP po třetí sérii

		Škola	*ročník*	*sérií*	*2331*	*2332*	*2333*	*2334*	*2335*	*2336*	*2337*	*série*	*celkem*
1.	Jakub Zíka	GNAlejíPH	4	3	9	14		9			12	44,7	134,6
2.	Vojtěch Hlávka	GŠlapanice	2	8	9	11,5	8	10	7	6,7	11	42,8	129,3
3.	Juda Kaleta	GKlatovy	2	4	9	3	12	9			9	41,9	126,6
4.	Lukáš Folwarczný	GKomHavíř	3	4	3	6,5		10			11	36,3	123,0
5.	Filip Hlásek	GMikulášPL	4	18	9		12	10		8,3		37,6	119,2
6.	Michal Anderle	GTim_Lučen	4	3	9			9	3	9		32,5	114,5
7.	David Bernhauer	GZborovPH	3	3		3,5		10	3	4,6	8,5	34,0	109,9
8.	Peter Zeman	GAnVra	4	3	9			10	7		11	37,7	108,8
9.	Vojtěch Sejkora	SPSE_Pard	2	3	7	3			2	0	11	29,6	105,0
10.	Martin Raszyk	G_Karvina	1	3	5			10	2,5		12	33,4	102,2
11.	Jan Hadrava	GZborovPH	3	3	9			8				18,3	99,8
12.	Matěj Kocián	GLesníZlín	4	5		14		9				23,6	99,2
13.	Michal Pokorný	SŠkybernHK	3	3					2,5	6,3	12	24,2	94,4
14.	Jan Bok	GJungmanLT	4	4	9			10	3		8	33,7	84,0
15.	Ondřej Hübsch	GArabskáPH	1	8	9			10				19,0	83,7
16.	Jerguš Greššák	GRaymanaPV	2	4				10				10,0	82,6
17.	Jindřich Pilař	GBroumov	3	4	5	6,5	1		3	1,5		24,1	80,2
18.	Štěpán Šimsa	GJungmanLT	2	11								0,0	79,6
19.	Vojtěch Kletečka	GHavlBrod	3	3	4	3					3,5	18,6	71,2
20.	Ondřej Fiedler	GJungmanLT	4	4	9			9	3,5	4,7		30,4	69,7
21.	Ondřej Mička	GJírovcČB	2	7					7	4,7		12,9	65,8
22.	Ondřej Cífka	GNAlejíPH	2	5					3		2	8,0	59,5
23.	Daniel Stahr	GJungmanLT	4	5	9	6,5		10	3			32,7	59,4
24.	Matouš Kozma	BiGyBBHK	4	2	9				3	4,7	12	33,1	50,2
25.	Jiří Setnička	G25březnPH	4	13	9					6,4		14,9	46,9
26.	Andrej Mariš	PriorPC	3	2								0,0	45,2
27.	Jiří Eichler	SlovanGOL	3	7								0,0	43,9
28.	David Krška	GJirsíkaČB	4	1								0,0	40,3
29.	Jan Paštyka	SPSKutHora	2	2								0,0	39,8
30.	Rastislav Rabatin	GJHroncaBA	2	1								0,0	37,4
31.	Filip Matzner	GJirsíkaČB	4	2								0,0	34,5
32.	Matěj Židek	GBroumov	3	4					2,5			4,1	34,0
33.	Robin Mana	GValašKlob	4	2								0,0	32,3
34.	Milan Berka	G_Krumlov	4	1								0,0	29,8
35.	Tereza Hulcová	GKlatovy	2	2					2	2,8		9,1	28,9
36.	Mária Mrocková	GJHroncaBA	4	4				9				9,6	28,6
37.	Daniel Švec	SPŠERožnov	3	1								0,0	27,9
38.	Jakub Kulhan	G_Kralupy	3	1	7	4			4	2,3		27,3	27,3
39.	Jonatan Matějka	GJírovcČB	1	6						3,2		4,7	26,4
40.	Jitka Fürbacherová	GKlatovy	2	2					2			3,9	24,3
41.	Michal Punčochář	GJírovcČB	1	1								0,0	24,2
42.	Tomáš Varga	GMost	-1	2								0,0	22,0
43.–44.	Anna Dresslerová	GJHroncaBA	4	2								0,0	19,0
	Milan Mikuš	GĽŠtúraTN	3	2								0,0	19,0
45.	Filip Štědronský	GMikulášPL	4	1				9		9		18,8	18,8
46.	Tomáš Velecký	GBezručeFM	0	2								0,0	17,3
47.	Jan Škoda	GMikulášPL	4	5	5					9		15,7	15,7
48.	Jiří Šebele	GArabskáPH	1	1								0,0	14,3
49.	Pavel Kratochvíl	VOŠGSvětlá	3	11				5				5,0	14,1
50.–51.	Martin Mach	GJírovcČB	3	4								0,0	10,0
	Alexander Mansurov	GNVPlániPH	2	4								0,0	10,0
52.	Josef Klesa	GKlatovy	3	1								0,0	9,5
53.	Martin Holec	GSlavičín	4	8								0,0	8,7
54.	Jan Lejnar	GKlatovy	1	1								0,0	8,6
55.	Tomáš Turlík	GRaymanaPV	2	1								0,0	8,4
56.	Petr Pecha	SPŠsVsetín	4	10	7							7,2	7,2
57.	Barbora Hourová	G_Brandýs	4	1								0,0	5,7
58.	Patrik Jung	GKlatovy	1	1								0,0	4,5
59.	Radim Cajzl	GNoMěsNMor	4	23						4,7		1,7	1,7