Milí řešitelé a řešitelky!

Zde je třetí série 23. ročníku KSP. Každá série obsahuje 7 úloh, z toho 4 nejlépe vyřešené se započítávají do celkového bodového hodnocení. Nezapomeňte, že k vyřešení některých úloh stačí prostudovat vhodné kuchařky.

Termín odevzdání třetí série je stanoven na pondělí 31. ledna v 8:00 SEČ, což znamená, že papírové řešení byste měli podat na poštu do středy 26. ledna.

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.

Třetí série třiadvacátého ročníku KSP

Edsger Dijkstra byl slavný holandský myslitel. Byl to samotářský, konzervativní člověk, kterému se jen velmi těžko určuje obor, jímž se zabýval. Narodil se roku 1930 v Rotterdamu, kde zůstal až do svých vysokoškolských studií teoretické fyziky. Později žil a učil v Eindhovenu, kde se věnoval praktické i teoretické matematice a informatice.

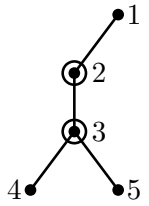
Zabýval se mimo jiné i grafy. Jedním z nejznámějších algoritmů, které vymyslel, je hledání nejkratší cesty v ohodnoceném grafu, jenž po něm nese jméno a můžete si ho přečíst třeba v naší kuchařce.[1] *My bychom po vás teď chtěli vymyslet jeden trochu jednodušší, ale zdánlivě podobný algoritmus.*

23-3-1 Úsporný kořen — 9 bodů

Na vstupu dostanete neohodnocený strom[2] zadaný počtem vrcholů a seznamem hran. Vrchol s hloubkou x bude takový vrchol, od kterého je každý jiný vrchol vzdálený maximálně x. Úsporný kořen je vrchol s nejmenší možnou hloubkou. Nalezněte všechny úsporné kořeny stromu.

Příklad vstupu:

```
5
1 2
2 3
3 4
3 5
```



Výstup: `2 3`

Jako správný samotář bydlel na vesnici, takže do školy jezdil jen v úterý. Vedl tam i seminář, kterému se příhodně říkalo Tuesday Afternoon Club, kde se řešily úlohy podobné těm z KSP. Kdykoliv tam či jinde studenti příliš hlasitě šuškali (to asi dobře znáte), nekřičel, ale naopak začal šeptat. To mělo ohromný efekt – všichni ztichli. Takový měl respekt.

Když se později stěhoval do Ameriky a cesta mu připadala dlouhá, vymyslel úlohu, kterou pak zadal americkým studentům při jejich prvním semináři Tuesday Afternoon Club.

23-3-2 Nejkratší cesta přes oceán — 14 bodů

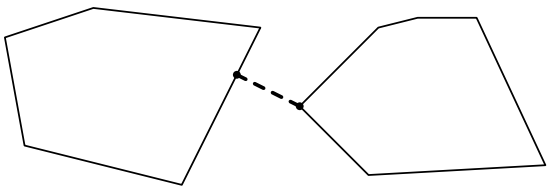
⚠ Pro zjednodušení si severní Ameriku i Evropu představme jako dva konvexní n-úhelníky, které se neprotínají. Chcete nalézt nejkratší cestu mezi nimi. Na vstupu nejdříve dostanete souřadnice Ameriky a potom souřadnice Evropy jako vrcholy v pořadí, v jakém leží na obvodu.

Vaším úkolem je najít takové dva body, jeden na obvodu Ameriky a druhý na obvodu Evropy, aby jejich vzdálenost byla co nejmenší. Pokud je víc možností, stačí vypsat libovolnou z nich.

Vstup (znázorněný obrázkem):

```
5
0 75
10 20
90 0
130 80
45 90

6
185 5
275 10
240 85
210 85
190 80
150 40
```



Výstup (čárkovaně):

```
118 56
150 40
```

Krom tisíce jiných věcí přemýšlel, jak počítače naučit počítat, třeba odpovědět na zadání `1+2*3`. K tomu účelu znovu objevil postfixový zápis a objevil, jak na něj běžný infixový zápis rychle převést. Pokud to chcete umět taky, můžete se podívat třeba do Wikipedie.[3]

Abychom nezůstali jen u teorie, podílel se na vývoji programovacího jazyka ALGOL 60 a později i jeho prvního překladače. Tento jazyk vznikl pro snadný zápis algoritmů a jako konkurence tehdy mohutně nasazovaného BASICu.

Edsger Dijkstra vůbec velmi brojil proti příkazu `goto` a zasazoval se o strukturované programování. Příkaz `goto` po-

[1] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`
[2] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`
[3] `http://cs.wikipedia.org/wiki/Shunting-yard_%28algoritmus%29`

važoval za nepřehledný. Samozřejmě, že na úrovni procesoru se stále používá, ale programátor by od něj měl být odstíněn, pokud to jen jde.

Měl rád programování rovnou na čisto, nejdřív si program rozmyslet a pak jej plynule psát. Lepší je chyby nedělat, než je hledat.

Následující úlohu vymyslete rovnou bez chyb a tak, aby šla zapsat bez `goto`. Pokud nevíte, co to `goto` je, máte to snazší, buďte jen rádi.

23-3-3 Skok bez padáku — 13 bodů

Z letadla vyskočil Američan, leč až po výskoku si uvědomil, že místo padáku si vzal batoh spolucestujícího Čecha. Naštěstí pro něj je na stráni pod ním rozmístěno N trampolín.

Představme si stráň jako rovinu postavenou svisle, tedy souřadnice x určuje horizontální pozici a souřadnice y je výška.

Každá trampolína je určena dvojicí souřadnic (x, y). Parašutista má nějakou počáteční pozici (x_0, y_0). Padá ve směru osy y, dokud nenarazí na trampolínu, která je o d níže než on. Od ní se odrazí a vyletí o $d/2$ výše, pak si může vybrat, jestli se posune o 1 vlevo, nebo vpravo (posunout se musí) a posune se podle toho.

Toto se opakuje, dokud nedopadne na zem. Na vstupu také dostanete výšku h; pokud spadne na zem z větší výšky, způsobí si zranění neslučitelná se životem a přivolaný lékař konstatuje smrt.

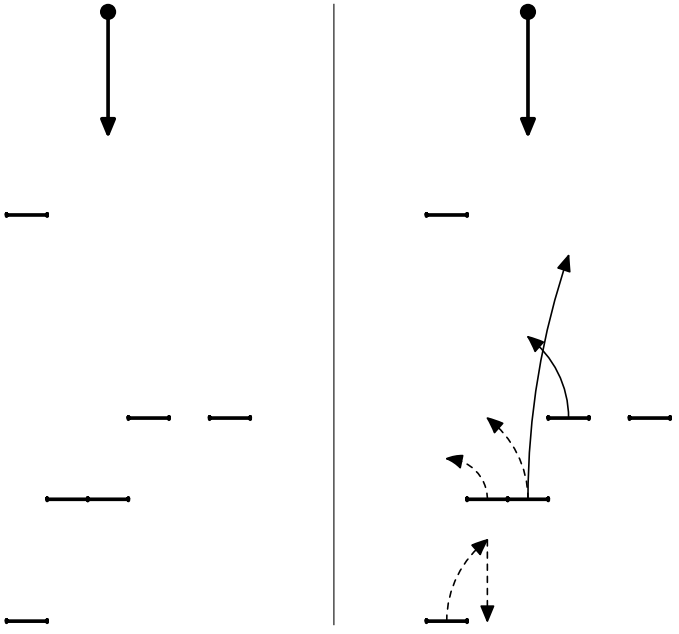
a) Určete, jestli má šanci přežít, a pokud ano, jak má postupovat. Nalezněte nejkratší možné řešení (nejméně odrazů). (5 bodů)

b) Nalezněte všechna možná $y \leq y_0$, pro která přežije pád s počáteční pozicí (x_0, y). (8 bodů)

```
2 15 3
6
0 0
0 10
1 3
2 3
3 5
5 5
```

Příklad vstupu: Na prvním řádku jeho počáteční souřadnice a h_0, na druhém řádku K, na dalších K souřadnice trampolín.

Na obrázku vlevo náčrt zadání, vpravo možné řešení (nejprve skáče po plných, následně po čárkovaných šipkách).



Zasazoval se o eleganci nejen zdrojových kódů, ale i matematických důkazů. Ty jeho byly zvlášť pěkné. Málokdy přesáhly 16 stran a každou větu pečlivě vybíral, aby nebyla zbytečná, nudná ani nepochopitelná. Důkazy psal jako pohádky. Neměl rád dlouhé formální řady implikací ani důkazy sporem, zato zvládl i třeba v geometrii nebo algebře použít algoritmické důkazy a jiné překvapivé finty z programátorského světa.

Jeho konzervativní přístup k vědě se projevoval třeba tím, že nerad jezdil na konference, ale raději vykládal v menší skupině lidí. Většinu své práce psal na stroji. Osobní počítač si pořídil až ke konci života, ale i tehdy ho používal minimálně a preferoval psaní rukou. Měl krásné technické tiskací písmo, které se používá i jako počítačový font.

23-3-4 Psaní písmen — 10 bodů

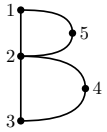
Každé písmeno se skládá z bodů a linií, které je spojují. V jednom bodě může začínat i končit více linií. Při psaní perem lze psát víc navazujících linií jedním tahem, nejde-li to, musí se pero zvednout a začít jinde. Kolikrát nejméně je potřeba pero zvednout?

Na vstupu dostanete neorientovaný graf o N vrcholech a M hranách a vypište, kolika nejméně tahy lze nakreslit.

Samozřejmě šetříme, takže je zakázáno jakoukoli hranu nakreslit více než jednou. Jinak řečeno, nesmíte se vracet po již nakreslených liniích.

Příklad vstupu:

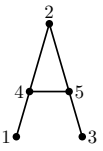
```
5 6
1 2
2 3
3 4
4 2
2 5
5 1
```



výstup: 1

Jiný příklad:

```
5 5
1 4
4 2
2 5
5 3
4 5
```



výstup: 2

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[4] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

Jeden z mnoha jeho textů (EWD 1250) – známý problém dvou párů, loďky a řeky. K řece přišly dva páry a potřebují překonat řeku tak, aby nikdy nezůstal sám pán z jednoho páru s dámou z druhého páru. Loďka unese maximálně dva lidi.

The couples, the river, and the little boat

Here is the problem:

"Two husbands and two wives have to cross a river in a boat which can hold only two people. How can they cross so that no woman is in the company of a man unless her husband is also present?"

(Copyright © 1967 by Morris Kline)

---

[4] http://ksp.mff.cuni.cz/zaciname/codex.html

Pamatoval i na ty, kteří už úlohu znali, nebo hned vyřešili. „Žádost: Pokud vám připadá tento problém příliš jednoduchý na to, abyste na něj plýtvali svým časem, hledejte prosím místo toho počet různých řešení. Děkuji. ⟨Konec žádosti.⟩“

Request If you think this problem too trivial to waste your time on, please state instantaneously the number of different solutions. Thank you. (End of Request.)

Mimochodem, někteří organizátoři KSP mají velmi podobné písmo… také vám připadá výrazně čitelnější než klasické psací písmo?

Někdy mu bylo vyčítáno, že neuváděl žádné nebo málo zdrojů. Ale co měl dělat, když něco vymyslel jen tak? Navíc většinu své práce nepublikoval v časopisech, ale posílal přátelům a známým. Tyto články jsou označovány EWD (jeho iniciálami) a číslem, třeba EWD 1250, a dají se stáhnout volně na internetu.[5]

Analogii můžeme najít v hudbě, kde se takhle označují díla slavných skladatelů, asi nejznámější jsou BWV (Bach-Werke-Verzeichnis) a HWV (Händel-Werke-Verzeichnis). Zásadní rozdíl je ovšem, že Dijkstra si to čísloval sám, kdežto hudebníci to neřešili a udělal to za ně někdo jiný o mnoho let později.

EWD shromažďuje pan Ham Richards. Jednou, když je nesl, zakopl a rozsypaly se mu po podlaze.

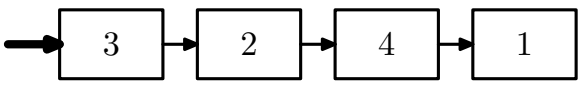
23-3-5 Rozházené EWD — 7 bodů

Chudák pan Richards má jen svou zapomětlivou hlavu a pár papírů, tak budete muset vymyslet, jak setřídit EWD v konstantní paměti. To jest, že si může udělat třeba 1000 záznamů, ale ne pro každou z N EWD jeden. Vámi spotřebovaná paměť prostě na N vůbec nesmí záviset (a N může být libovolně velké – argument, že EWD je konečně mnoho, vám neprojde). Dávejte si pozor na rekurzi, spotřebovává tolik paměti, jak hluboko je zanořená.

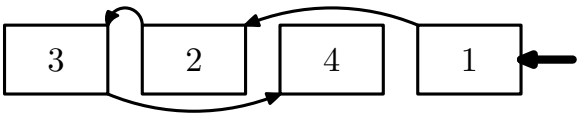
Přeházená EWD budeme reprezentovat jako spojový seznam. V programu dostanete ukazatel na první prvek spojového seznamu, kde je číslo EWD a ukazatel na další. Vaším úkolem je ho setřídit a vrátit ukazatel na první prvek (nejstarší EWD).

Spojový seznam už máte v paměti, vaším úkolem je přepojit jej do setříděného stavu.

Příklad před setříděním:



A po setřídění:



Z úplně jiného soudku je jeho návrh operačního systému THE multiprogramming system, zaměřeného na sekvenční zpracování úloh a s podporou multitaskingu a paralelizace. Velkého rozšíření se sice nedočkal, nicméně myšlenky vytvořené pro něj se ujaly. Jestli vás paralelní svět zajímá, měli jsme o něm kdysi seriál[6] *a také o něm byla série úloh před pár lety v Matematické olympiádě.*

Jako už starý se vrátil do Holandska do svého původního domu ve vesnici Nuen, kde roku 2002 zemřel na rakovinu.

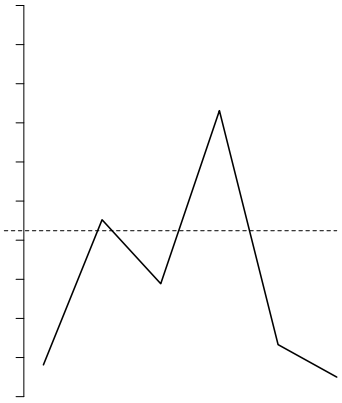
Pozůstalí pak přemýšleli, jestli mu na hrob napsat, že byl matematik, nebo informatik. Hodilo by se jim k rozhodování vědět, kolik procent lidí si nejčastěji myslelo, že byl informatik.

23-3-6 Výzkum veřejného mínění — 9 bodů

Za jeho života se dělalo N výzkumů veřejného mínění, jestli byl informatik. Výsledkem bylo vždy číslo v procentech s přesností na M desetinných míst. N je řádově tolik jako 2^M. Můžeme předpokládat, že výzkumy odpovídaly realitě a mezi jednotlivými výzkumy procento lidí, kteří si myslí, že ano, buď jen stoupalo, nebo jen klesalo. Vaším úkolem je zjistit, jaké procento bylo během jeho života nejčastější. Případně určit libovolné z nejčastějších. Nezapomeňte, že to nutně nemuselo být v době, kdy se konal výzkum.

Příklad vstupu:

```
6 5
8.124 45.223 28.8723 73.117 13.3 5.0
```



Výstup (vyznačen čárkovanou čarou): `42.42` (4×)

23-3-7 Automaty stokrát jinak — 12 bodů

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

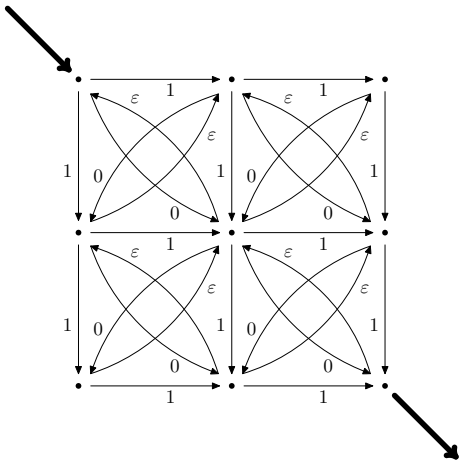
Minule jsme si popsali nedeterministický konečný automat (NKA) s ε-přechody. Definovali jsme, že vstup přijme, pokud v něm existuje alespoň jedna možnost, jak při čtení onoho vstupu skončit ve výstupním stavu.

Běžný program ale neumožňuje paralelně zkoumat všechny větve, kterými by mohl procházet. To bychom si museli pořídit třeba paralelizátor jako v jednom ze starých ročníků olympiády.

Mohli bychom však zkusit převést NKA na DKA, který programem simulovat velmi jednoduše umíme. Dá se dokázat, že to jde vždycky (i když výsledný automat může mít až exponenciální velikost), ale obvykle se to dělá ukázáním obecného postupu, takže si to ukážeme až v řešení.

Úkol 1 [6b]: Vymyslete, jak simulovat NKA a jak převést NKA na DKA. Pokud vám to nepůjde ve vší obecnosti, máte šanci získat 2 body za převod tohoto konkrétního automatu:

[5] `http://www.cs.utexas.edu/users/EWD/welcome.html`
[6] `http://ksp.mff.cuni.cz/viz/12`



Samozřejmě, pokud vymyslíte úlohu obecně, tak ji nemusíte řešit konkrétně na tomto automatu, nicméně snad nic nebrání jeho použití třeba k názornému ilustrování vašeho postupu. Za pomoci mašinérie, kterou jsme si zatím ukázali, by pro vás neměl být problém zjistit následující (ale můžete to dělat i jinak, jestli chcete):

Úkol 2 [5b]: Zjistěte, jestli tyto dva regexy popisují stejný jazyk (tedy vyhovují jim právě stejné řetězce):

```
(AB)*(AA(BA)*(A|BB)(AB)*)*
(A(AB)*(B|AA))*
```

Úkol 3 [1b]: Nalezněte nejmenší násobek devíti, jehož desítkový zápis vyhovuje tomuto výrazu:

```
(102)*101(201)*((0|202)(102)*101(201)*)*
```

Nezapomeňte, že součástí každého úkolu je přesvědčit opravujícího, že vaše řešení je správně. Regex bez jakéhokoli vysvětlení není úplné řešení a nemá nárok na plný počet bodů. Stejně tak konstatování „NE“, „10202010102“ nebo „nelze“...

Tím jsme uzavřeli kapitolu konečných automatů. Příště se vrátíme zpátky k regexům, ukážeme si, jak se programem `sed` nahrazují řetězce za jiné, co dělat, když regex pro náš požadovaný řetězec prostě neexistuje a jak s tím vším souvisí Chuck Norris.

Vzorová řešení druhé série

23-2-1 Balíčky balíčků

Naše úloha se docela podobá problému batohu (viz kuchařka), takže by nás mohlo napadnout použít modifikovanou verzi algoritmu, kterým se řeší.

Postupně procházíme celá čísla od nuly vzhůru a pokud jsme právě na hodnotě, kam se umíme dostat, tak projdeme všechny nabídky a pro každou z nich si poznačíme, že se umíme dostat na hodnotu, která je součtem této nabídky a hodnoty, na které právě jsme. Na začátku víme jenom to, že se umíme dostat do čísla nula. Takhle postupujeme, dokud se nedostaneme do čísla, které je větší nebo rovno H, a máme řešení.

Tenhle postup sice funguje, ale dosti pomalu. K rychlejšímu algoritmu dojdeme, když si uvědomíme, co to znamená, že každou nabídku můžeme použít, kolikrát chceme – to, že kdykoliv umíme poslat x kg, tak umíme poslat i $x + kN$ kg pro jakékoliv nezáporné celé číslo k (N kg je totiž hmotnost nejmenší nabídky).

Díky tomu si můžeme pole hmotností přeuspořádat do tabulky o N sloupcích. Políčko na i-tém řádku j-tého sloupce pak představuje $(i \cdot N + j)$ kg.

K vyplňování této tabulky bychom mohli použít stejný postup jako před chvílí, ale my si ho upravíme tak, že když jsme na nějakém políčku a umíme se dostat do nějakého políčka nad ním (číslo sloupce je stejné, číslo řádku menší), tak si poznačíme, že se umíme dostat i do aktuálního políčka, ale už nemusíme zjišťovat, kam se odsud můžeme dostat s použitím různých nabídek.

To proto, že pokud se na nějaké políčko umíme dostat z aktuálního použitím nabídky x kg, tak se tam umíme dostat i ze zmíněného políčka nad ním. A to nejdříve použitím nabídky x kg a následně několikanásobným použitím nabídky N kg.

Tuto tabulku si ale nemusíme pamatovat celou. Stačí si pro každý sloupec pamatovat, který je první řádek v tomto sloupci, na který se umíme dostat.

Tento seznam sloupců pak procházíme dokola podobně, jako jsme předtím procházeli celou tabulku – jeden průchod seznamem odpovídá průchodu jedním řádkem v tabulce.

Navíc ani nemusíme procházet seznamem slouců tolikrát, kolik řádků bychom prošli v tabulce. Jakmile se jednou umíme dostat do sloupce, který obsahuje cílové políčko, tak víme, že se umíme dostat až tam.

Pro určení výsledné kombinace balíčků si musíme pro každý sloupec zapamatovat, s použitím jakého balíčku jsme se tam dostali.

Samotnou výslednou kombinaci určíme tak, že nejdříve započítáme nabídku N kg tolikrát, kolik řádků by činil rozdíl v tabulce mezi cílovým políčkem a políčkem, kam se umíme dostat. Následně procházíme sloupce podle toho, pomocí kterého balíčku jsme se do něj dostali, dokud se nedostaneme do nultého sloupce. Všechny balíčky, které jsme na této cestě použili, započítáme také a máme kýžený výsledek.

Jakou má tento algoritmus složitost? Paměťová je $\mathcal{O}(N)$ – nejvíce zabírá seznam sloupců a těch je N.

S časovou složitostí je to složitější. Procházení nabídek provádíme nejvýše jedenkrát pro každý sloupec, což nám dává $\mathcal{O}(N^2)$. Protože se ale může stát, že budeme procházet seznamem opakovaně, dokud se neumíme dostat do všech sloupců, potřebujeme zjistit, kolikrát nejvýše to uděláme.

Stačí se podívat na jedinou nabídku: $2 \cdot (N - 1)$. Pokud budeme používat jenom tuto nabídku, tak se v případě lichého N po N krocích dostaneme do každého sloupce. Došli jsme tedy až do čísla $N \cdot 2 \cdot (N - 1)$, a počet průchodů seznamem je tedy $2 \cdot (N - 1) = \mathcal{O}(N)$.

V případě sudého N se do lichých sloupců nedá dostat žádným způsobem a použitím stejné nabídky jako v předchozím případě se po $N/2$ krocích dostaneme do všech dostupných sloupců. Prošli jsme tedy seznamem opět $\mathcal{O}(N)$-krát.

V obou případech tedy musíme projít v nejhorším případě $\mathcal{O}(N^2)$ políček. Zpětný průchod pro zjištění výsledku projde každým sloupcem nejvýše jednou a složitost nám tedy nezhorší. Celková časová složitost tedy je $\mathcal{O}(N^2+N^2+N) = \mathcal{O}(N^2)$.

Vzorový program je na konci letáku.

Petr Onderka & CodEx

23-2-2 Zastavení

Zkusme nejprve generovat čísla 1 až 120. Hodíme jednou šestistěnkou a jednou dvacetistěnkou, máme tedy 6 možností, jak dopadne hod první kostkou a 20 možností, jak dopadne hod druhou kostkou. Celkem tedy máme 120 různých možností a pro každou možnost odpovíme jiným číslem.

Kdybychom chtěli generovat čísla od 1 do 50, hodíme dvakrát desetistěnkou a dostaneme 100 různých možných výsledků ($(3,1)$ a $(1,3)$ jsou rozdílné výsledky). Všechny výsledky jsou stejně pravděpodobné, každá kostka je dokonale náhodná a jednotlivé hody se neovlivňují.

Pokud tedy program odpoví jedničkou pro první dva výsledky (pro libovolné uspořádání), dvojkou pro další dva atd., umí správně generovat požadovaná čísla, protože generuje každé se stejnou pravděpodobností a potřebuje konečný počet hodů.

Nyní obecnější případ, chceme generovat N čísel a N dělí nějaký násobek počtů stěn našich kostek P – to je ve skutečnosti počet možných výsledků, které můžou nastat po hodech těmito kostkami. Rozdělíme všechny možné výsledky na N (disjunktních) částí o P/N prvcích a použijeme předchozí postup.

Zbývá ukázat, že pro jiná N nedokážeme na zaručeně konečný počet hodů vždy vygenerovat správný výsledek. Nejmenší N takové, že nedělí žádné možné P, je 7. V prvočíselném rozkladu žádného počtu stěn našich kostek totiž není 7.

Napřed rozeberme špatné postupy. Zahození některých výsledků – hodím osmistěnkou, pokud padne 8, hodím znova. Takovému algoritmu by mohla padat pořád 8 a nezastavil by se, leda by nám stačil průměrně konečný počet hodů, viz úloha 16-1-5[8]

Když nemůžeme dostat vhodné P násobením, zkusíme sčítat – sečtu dvě padlá čísla po hodu čtyřstěnkou a od toho odečtu 1. Tento postup ale nedává stejné pravděpodobnosti všech čísel.

Jednička může vzniknout jen poté, co padne $(1,1)$, ale trojka může vzniknout po pádu $(1,3)$, $(2,2)$ nebo $(3,1)$, takže trojkou by algoritmus odpověděl s třikrát větší pravděpodobností. Některým číslem odpovím i jindy – pokud padne 8, odpovím jedničkou, ale toto triviálně nedává stejnou pravděpodobnost všem číslům.

Jak tedy dokázat, že žádný algoritmus si nemůže vystačit s konečným počtem hodů?

Pro spor budeme předpokládat, že existuje nějaký algoritmus, který správně generuje pro N, která nedělí žádné možné P. Po nějakém konečném počtu hodů program proběhne jedním z P různých způsobů (všechny jsou stejně pravděpodobné) a na konci každého odpoví nějakým z N požadovaných čísel.

Kdyby ale všechny odpovědi měly stejnou pravděpodobnost, znamenalo by to, že jsme dokázali P celočíselně a bez zbytku vydělit číslem N, což je spor s předpokladem.

Umíme tedy generovat jen pro taková N, která dělí nějaké P.

Martin Böhm & Karel Král

[8] `http://ksp.mff.cuni.cz/tasks/16/tasks1.html#task5`

23-2-3 Projížďka

Trocha magie

Milý čtenář mi jistě pro jednou odpustí, pokud si zahraji na kouzelníka a vytáhnu jednoho králíka z klobouku.

Napřed, zadání šlo chápat různými způsoby, avšak příliš neměnilo podstatu řešení. Předpokládejme tedy například, že všechny cesty jsou jednosměrky a že „z rozcestí vychází sudý počet cest“ znamená, že právě polovina tohoto sudého počtu je v příchozím a právě polovina v odchozím směru.

Opravdu nám stačí taková podmínka pro orientovaný graf. V neorientovaném jsme potřebovali sudý počet, protože kdykoliv jsme vešli do vrcholu, také z něj někudy musíme odejít. Stejně to funguje pro orientovaný, jen musíme přijít po vstupní hraně a odejít po výstupní. Že jde o podmínku postačující, lze nahlédnout také zcela stejně jako v neorientovaném grafu. Jediné, na co si musíme dát pozor, je, že při vypisování dostáváme hrany pozpátku.

Na grafu na vstupu (rozcestí jsou vrcholy a cesty jsou hrany) si najdeme uzavřený eulerovský tah (to již za nás vyřešila kuchařka). Nyní jej projdeme a budeme si udržovat průběžný součet prošlých hran (říkejme tomu součtu odpočatost). Rozeberme dva případy.

Jako první případ vezmeme situaci, kdy po projití celého tahu dostaneme záporné číslo. Potom je součet všech hran záporný a takový zůstane, ať je vezmeme v libovolném pořadí. Proto úloha nemá řešení.

Pokud průšvih popsaný v minulém případu nenastane, vezmeme místo v tahu, kde se nachází minimum ze všech odpočatostí (místem v tahu není myšlen jen vrchol, ale i který průchod tímto vrcholem máme na mysli, neboť při různých průchodech můžeme mít různé hodnoty odpočatosti). V tomto místě v tahu začneme (jakoby jej pootočíme).

Složitost

Máme hezké lineární řešení (jak pamětí, tak časem), neboť již kuchařka nám ukázala, že eulerovský tah v dané složitosti zvládneme najít, a přidali jsme jen dva průchody vzniklým cyklem (jeden na průběžné počítání, druhý na výpis „pootočené“ verze).

Proč to funguje

Nyní už jen zbývá zdůvodnit, proč tento algoritmus vlastně počítá, co má. První případ je nezajímavý (neboť jsme jej již zdůvodnili výše). Dále tedy předpokládejme, že nám nastal druhý případ. Protože máme uzavřený eulerovský tah, projedeme každou cestou právě jednou. Zbývá dokázat, že odpočatost v pootočeném tahu nikde neklesne do záporných čísel.

Předpokládejme tedy, že v místě $š$ na tahu máme zápornou odpočatost. Minimum máme v místě m. Pokud by v původním neotočeném tahu bylo $š$ až za m, pak by muselo být také s menším číslem než m a m by tedy nebylo minimum. Tento případ tedy nenastal.

Takže $š$ je před m. Představme si, že jsme prošli tahem dvakrát místo jednou, tedy při druhém průchodu $š$ jsme na nižším čísle, než při prvním průchodu m (proto nám po pootočení v $š$ vyšlo něco záporného). Ale protože druhý průchod nezačíná od nuly, ale od něčeho nezáporného, odpočatost druhého průchodu $š$ je alespoň tak velká, jako první. Tedy i při prvním průchodu $š$ jsme měli nižší číslo než u m, což je opět ve sporu s výběrem minima.

Jak na to přijít

Jednak, kdyby na to bylo jednoduché přijít, nebyla by úloha za 12 bodů. Ale přesto si řekneme způsob, jak na to přijít.

Můžeme si představit, že jsme řešení již našli a koukat na jeho vlastnosti. To, že je to uzavřený eulerovský tah, je vidět celkem jednoduše. Dále si všimneme, že vybráním jiného začátku se nám všechna čísla posouvají jen nahoru a dolů, rozdíly zůstávají stejné (s výjimkou rozpojeného konce – začátku). No a dále víme, že nejmenší číslo je 0 a to je na počátku.

Program (C):
`http://ksp.mff.cuni.cz/tasks/23/2323.c`

Michal „Vorner“ Vaner

23-2-4 Plánování

Budeme hladově přiřazovat letadla událostem tak, jak nám ze vstupu (seřazené dle počátků) přijdou pod ruku.

Podrobněji řečeno: v každé chvíli běhu programu si budeme udržovat hypotézu „stačí nám L letadel“, kde L navýšíme jen tehdy, ukáže-li se býti flagrantně špatná tím, že nebudeme mít při zpracování počátku události žádné volné letadlo.

Pokud volné letadlo mít budeme, prostě ho dané události přidělíme – k uchovávání volných letadel můžeme mít zásobník, do kterého budeme házet jejich pořadová čísla. Nebo frontu, pokud toužíme vytvářet zdání spravedlivého rozvrhování vůči pilotům. (Rozmyslete si.)

Tímto jistě dojdeme ke správnému minimu počtu letadel, protože pokud nám po poslední události zbyla hypotéza „stačí nám L letadel“, jistě se někdy stalo, že $L-1$ letadel nedokázalo pokrýt probíhající události. Zároveň je jasné, že tak umíme vytvořit správné rozvrhy, protože jsme si celou situaci de facto odsimulovali.

Z tohoto popisu to vypadá, že nám stačí lineární čas, ale celá věc má jeden háček: potřebujeme zpracovávat konce událostí (uvolňovat letadla), ale kdy?

Musí to být před dalšími odlety, abychom zbytečně nezvýšili L, musí to být po předchozích odletech, abychom nenabyli zdání, že letadel potřebujeme méně – asi nám nezbyde nic jiného, než tyto konce v $\mathcal{O}(N \log N)$ zatřídit do vstupní posloupnosti, jejíž počáteční setřízení podle počátků nám nakonec z hlediska časové složitosti k ničemu nebylo.

Paměťová složitost je samozřejmě lineární.

Úloha souvisí se specifickými grafy, kterým se říká intervalové, ale jejich přímé použití by program nevyhnutelně zpomalilo a vzhledem k jednoduchosti algoritmu by nám nijak nepomohly ani ve výše provedené úvaze.

Vzorový program je na konci letáku.

Lukáš Lánský

23-2-5 Zaměřování

Obsah trojúhelníku lze spočíst jako $S = av_a/2$, kde a je základna a v_a odpovídající výška. Délka základny je známa, a tedy pokud existuje bod, který spolu s kanóny tvoří trojúhelník s obsahem právě S, bude ležet na průsečíku mnohoúhelníku a rovnoběžek spojnice kanónů ve vzdálenosti v_a od nich.

V programu si můžete všimnout, že jsou ignorovány hrany rovnoběžné se spojnicí kanónů. To si můžeme dovolit, neboť u nich záleží jen na okrajových bodech a ty se uváží nejpozději v kroku, kdy nenalezneme průsečík.

Pokud takový průsečík nenalezneme, tak bod, který by spolu s kanóny tvořil trojúhelník s obsahem právě S, neexistuje. Potom je třeba hledat bod, který spolu s kanóny vytvářel trojúhelník s obsahem co nejbližším k zadání. Jeden z takových bodů bude určitě vrchol mnohoúhelníku.

To se snadno nahlédne sporem. Pokud by existoval takový bod X uprostřed hrany AB, tak mohou nastat dvě možnosti. Buď je hrana AB rovnoběžná se spojnicí kanónů (a pak je jedno, který bod z této hrany uvážíme – všechny budou vytvářet trojúhelník se stejným obsahem), nebo není rovnoběžná, a pak pokud s X hneme, tak na jednu stranu bude obsah trojúhelníku růst, na druhou klesat. A jelikož víme, že trojúhelník s obsahem přesně S neexistuje, tímto posunutím jsme našli bod s menším rozdílem obsahů a máme spor.

Projdeme tedy všechny vrcholy mnohoúhelníku a najdeme ten, který odpovídá úloze.

Časová složitost je $\Theta(N)$ – seznam vrcholů projdeme právě $2\times$ – a paměťová také $\Theta(N)$ – někde musí být uložen vstup. Pokud bychom uvážili, že vstup nám bude někdo zadávat postupně, dal by se program upravit, aby potřeboval jen další $\Theta(1)$ paměti.

Následuje pár poznámek k užité analytické geometrii, kterou jsme hojně využívali ve zdrojovém kódu.

Skalární součin vektorů $\vec{a}$ a $\vec{b}$ se určí jako

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y.$$

Platí pro něj $\vec{a} \cdot \vec{b} = |\vec{a}| \cdot |\vec{b}| \cos\theta$, kde θ je úhel, který spolu vektory $\vec{a}$ a $\vec{b}$ svírají a $|\vec{a}|$, resp. $|\vec{b}|$ jsou velikosti vektorů $\vec{a}$ a $\vec{b}$. Speciálně platí $\sqrt{\vec{a} \cdot \vec{a}} = |\vec{a}|$ a $\vec{a} \cdot \vec{b} = 0$, pokud jsou na sebe vektory $\vec{a}$ a $\vec{b}$ kolmé.

Dále potřebujeme popsat úsečku a přímku. Nejjednodušší je parametrický popis. Uvažme, že úsečka je mezi body, jejich polohu zapíšeme jako vektor od počátku souřadnic. Pro body $\vec{X}$ ležící na ní platí

$$\vec{X} = \vec{A} + t\left(\vec{B} - \vec{A}\right),$$

kde t je reálný parametr nabývající hodnot mezi nulou a jedničkou. Zřejmě nula odpovídá bodu $\vec{A}$, jednička bodu $\vec{B}$ a ostatní hodnoty bodům mezi okraji. Pokud bychom z tohoto chtěli přímku, stačí vynechat omezení $t \in \langle 0, 1\rangle$.

Pro přímku však existuje i jiný způsob popisu. Uvažme, že známe vektor $\vec{n}$ kolmý na $\vec{B} - \vec{A}$. Pokud jím skalárně vynásobíme parametrický zápis přímky, dostaneme rovnici $\vec{X} \cdot \vec{n} + c = 0$, kde $c = -\vec{A} \cdot \vec{n}$ (tedy nějaká konstanta) a $\vec{X}$ obecný bod. Této rovnici se říká implicitní zápis přímky. Lze také ukázat, že každé řešení této rovnice je popsáno odpovídajícím parametrickým zápisem.

U implicitního zápisu ještě chvíli zůstaneme. Označme $\vec{s}$ vektor spojující body A a B, kterými prochází přímka, $\vec{s} = \vec{B} - \vec{A}$. Normálový vektor k němu zvolme $\vec{n} = (-s_y, s_x)$. Snadno nahlédneme, že opravdu $\vec{n} \cdot \vec{s} = 0$. Implicitní tvar rovnice přímky procházející body A a B tak může být zapsán jako $\vec{n} \cdot \vec{x} + c = 0$.

Nyní však uvažme, co se stane, pokud do ní dosadíme bod, který na přímce neleží. Podívejme se podrobněji, co dostaneme na pravé straně. Bod $\vec{X}$ lze zapsat jako $\vec{X} = \vec{A} + \alpha\vec{n} + \beta\vec{s}$, kde α a β jsou nějaká jednoznačně určená čísla (α určuje posun po přímce od bodu A a β posun kolmo k ní). Po dosazení do implicitní rovnice dostaneme

$$\vec{n} \cdot \left(\vec{A} + \alpha\vec{n} + \beta\vec{s}\right) - \vec{n} \cdot \vec{A} = \beta\vec{n} \cdot \vec{n} = \beta|\vec{n}|^2.$$

Vzhledem k výše popsané konstrukci $\vec{n}$ si snadno čtenář ověří, že $|\vec{n}| = |\vec{s}|$, tedy že velikost normálového vektoru je rovna vzdálenosti bodů A a B. Kromě toho víme, že $\beta\vec{n}$ je takový posun směrem kolmým na přímku, abychom se z přímky dostali do bodu X. Tedy velikost tohoto vektoru (rovná $|\beta| \cdot |\vec{n}|$) je výška trojúhelníku ABX kolmá na stranu AB.

Proto výraz $|\beta| \cdot |\vec{n}|^2$ popisuje dvojnásobek obsahu trojúhelníku ABX. Ten v programu budeme určovat vzorcem $2S = |\vec{X} \cdot \vec{n} + c|$. Funguje jak pro určení obsahu trojúhelníku ABX, tak i pro určení rovnoběžky - zjevně stačí upravit konstantu c o $\pm 2S$.

Nakonec budeme potřebovat určit průsečík přímky a úsečky. Předpokládejme, že přímku máme implicitně zadanou ve tvaru $\vec{n} \cdot \vec{X} + c = 0$ a úsečku mezi body P a Q zadanou parametricky $\vec{X} = \vec{P} + t(\vec{Q} - \vec{P})$. Dosazením těchto rovnic do sebe a vyjádřením t dostaneme

$$t = -\frac{c + \vec{n} \cdot \vec{P}}{\vec{n} \cdot \left(\vec{Q} - \vec{P}\right)}.$$

Pokud platí, že takto spočtené $t \in \langle 0, 1 \rangle$, průsečík existuje, jinak ne.

Vzorový program je na konci letáku.

Pavel Čížek

23-2-6 Testovací

Nejjednduší řešení, které se nám na první pohled nabídne, je vyzkoušet všechny možné trojice (a_i, a_j, a_k), pro které platí $i < j < k$, a pro každou takovou trojici otestovat, zda platí rovnost $a_j - a_i = a_k - a_j$. Tím získáme jednoduché řešení pracující v $\mathcal{O}(N^3)$.

Jak si spousta z vás všimla, tento jednoduchý algoritmus můžeme urychlit tím, že využijeme setříděnosti posloupnosti a použijeme binární vyhledávání (o kterém se můžete dočíst v jedné z našich kuchařek). V naší úloze binární hledání využijeme k nalezení třetího prvku.

Tedy pro všechny dvojice (a_i, a_j) si spočítáme

$$a_k = a_j + (a_j - a_i)$$

a pokusíme se a_k vyhledat v intervalu a_{j+1} až a_{N-1}. Tím dostaneme řešení se složitostí $\mathcal{O}(N^2 \log N)$. Ale ani to ještě není optimálním řešením.

Optimální řešení pracuje v čase $\mathcal{O}(N^2)$ a využívá jak setříděnosti posloupnosti, tak toho, že ke každé dvojici (a_i, a_j) existuje nejvýše jedno a_k splňující podmínku. Jak na to?

Nejdříve si všimneme, že pokud $a_{k_0} - a_j < a_j - a_i$ platí pro nějaké k_0, tak tato nerovnost bude platit i pro všechna $k < k_0$. Naopak pokud $a_{j_0} - a_i < a_k - a_{j_0}$ platí pro nějaké j_0, tak stejná nerovnost platí i pro všechna $j < j_0$. Není těžké si na papíře rozmyset, proč.

A jak toho využijeme v našem řešení? Pro všechna možná i zvolíme $j = i+1$ a $k = j+1$ (následující prvky), pokud tedy $i + 2 < N$ (musí existovat), a dále opakujeme následující postup.

Pokud $a_j - a_i = a_k - a_j$, nalezli jsme řešení, vypíšeme jej a k a j zvýšíme o jedna (pro jedno a_j nemůže existovat více a_k).

Pokud $a_j - a_i > a_k - a_j$, zvýšíme k o jedna. Je důležité si uvědomit, že tuto operaci můžeme udělat a nepřijdeme tak o žádné řešení, protože pro všechna nižší k řešení už také neexistuje, nebo jsme jej už vypsali.

Zbývá nám možnost $a_j - a_i < a_k - a_j$. V tomto případě zvýšíme j o jedna, protože pro tohle j už žádné řešení nebude.

Celý postup opakujeme, dokud $k < n$.

Nyní si jen stačí uvědomit, že u neklesající posloupnosti, kde můžou být bloky stejných čísel, se nám nic hrozného nestane – jen když po zvýšení nějakého indexu $x \in \{i, j, k\}$ zjistíme, že $a_x = a_{x-1}$, zvýšíme jej ještě jednou.

Složitost je $\mathcal{O}(N^2)$, protože pro každé i maximálně N-krát iterujeme j i k o jedna. Toto řešení si můžete přečíst i jako zdrojový kód.

Nyní ještě dokážeme, že lepší časové složitosti v nejhorším případě nemůžeme dosáhnout. Uvažme jednoduše posloupnost $1, 2, \ldots, N$. V takové posloupnosti existuje $N-2$ trojic s diferencí 1, $N-4$ s diferencí 2, $N-6$ s diferencí 3 atd., až nakonec 1 trojice s diferencí $(N-1)/3$.

Počet všech trojic je tedy $(N^2-1)/4$, což je vzhledem k N kvadraticky mnoho, takže algoritmus může mít až kvadraticky velký výstup a nemůžeme dosáhnout lepší složitosti v nejhorším případě než $\Theta(N^2)$.

Vzorový program je na konci letáku.

Karel Tesař

23-2-7 Regulomaty

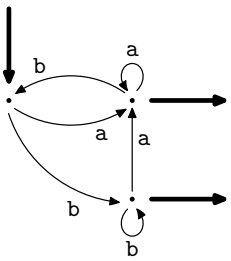
Převeďte automat na obrázku na regulární výraz. To se drtivé většině z vás podařilo, strhával jsem body za chybějící vysvětlení.

`(1+2+3+)*` bylo správné řešení, někteří z vás zapomněli, že existuje operátor `+` a zapsali to jako `(11*22*33*)*`, za což jsem strhával řádově desetiny bodu.

Jak se ovšem **úkol 1** řeší obecně? Jak dostanete z každého automatu regex, když jsem se v zadání chvástal, že to umím pro všechny? Existuje univerzální postup, který si tu předvedeme.

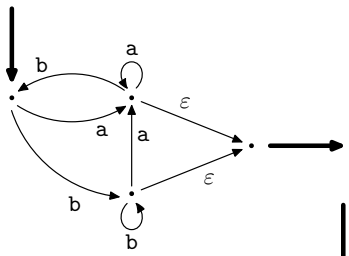
Postupně se budeme zbavovat vrcholů automatu, až nám jich zbyde jen pár, konkrétně ty vstupní a výstupní. Budeme na to pořád dokola používat tři operace:

1) spojení paralelních hran
2) odstranění smyček
3) odstranění vrcholu

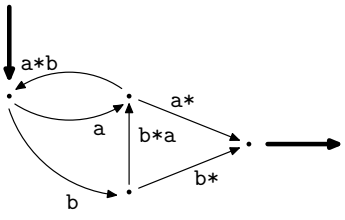


Celou věc si budeme ilustrovat na jiném, názornějším automatu, zde na obrázku.

Před započetím ještě musíme automat upravit tak, aby měl jen jeden výstupní stav. K tomu použijeme ε-hrany, které si teď na chvíli povolíme.

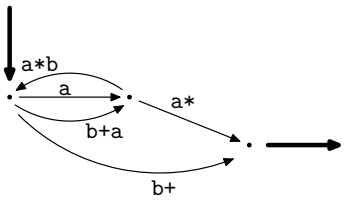


Vytvoříme tedy jeden stav navíc, který bude oním jediným výstupním, a ze všech bývalých výstupních stavů do něj natáhneme ε-hrany.



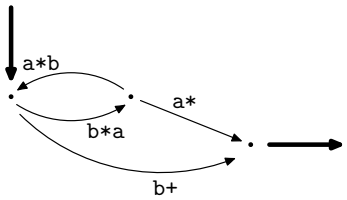
Nyní budeme cyklit pořád dokola naše tři body. Paralelní hrany zatím nemáme,

ale smyčky se nějaké vyskytují, tak je zrušíme. Obalíme je hvězdičkou a připojíme na začátek výstupních hran. Teď už nebudou hrany označeny znakem, ale regexem. Nakonec zbydou dva stavy – vstupní a výstupní – a jediná hrana mezi nimi, která bude označena výsledným regexem.

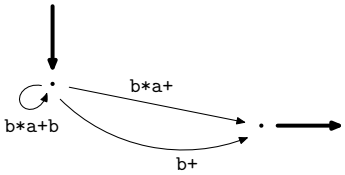


Vybereme si nějaký vrchol a ten odstraníme. Jednoduše vytvoříme všechny možné kombinace hran, jež bylo možné použít, abychom prošli tímto vrcholem. Samozřejmě neodstraňujeme vstupní a výstupní vrchol! Všimněte si, že na obrázku už jsou spojené výrazy `bb*a` do `b+a` a `bb*` do `b+`.

A zase od začátku. Spojení paralelních hran, tentokrát tady jeden případ máme, tak pryč s nimi! Výrazy `a` a `b+a` se mi spojí do `(a|b+a)`, což můžeme upravovat postupně na `(b+)?a` a `b*a`, což je výsledný výraz.

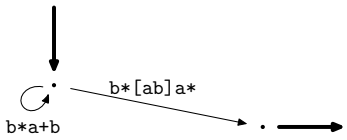


Eliminaci smyček pro tentokrát vynecháme, žádné v automatu zrovna nemáme, znovu budeme odstraňovat vrchol, teď už jediný odstranitelný.

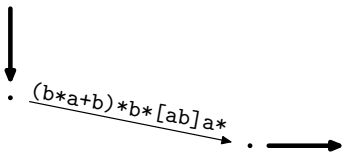


Zase jsme mohli nově vzniklé výrazy zjednodušit. Z výrazu `b*aa*` máme `b*a+` a na smyčce z původního `b*aa*b` vznikl `b*a+b`.

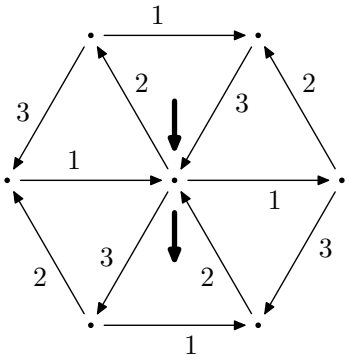
Přichází na řadu spojení hran, při kterém vznikne ze dvojice výrazů `b*a+` a `b+` postupně `(b+|b*a+)`, `b*(b|a+)` a `b*[ab]a*`. Poslední přeměna už nebyla úplně mechanická – výrazu totiž odpovídá libovolný řetězec nenulové délky, který nejdřív obsahuje jen `b` a potom jen `a`.



A po eliminaci smyček jsme u konce, na jediné hraně mezi vstupním a výstupním stavem máme výsledek.

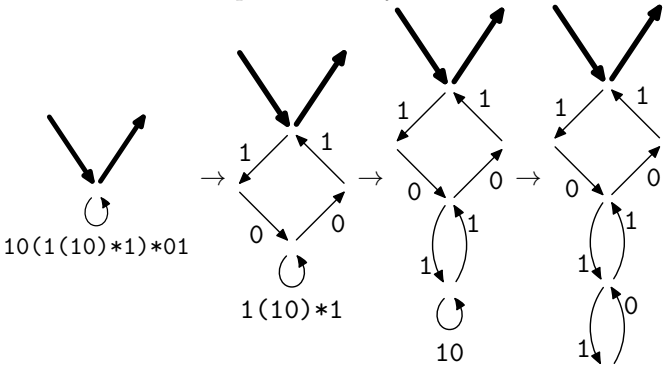


Správné řešení druhého úkolu bylo velice jednoduché, drtivá většina řešitelů za něj dostala plný počet bodů (s občasným stržením nějakých bodů za chybějící slovní popis, co že to je zač). Za nakreslení tohoto přehledného tvaru jsem uděloval malý bodový bonus.

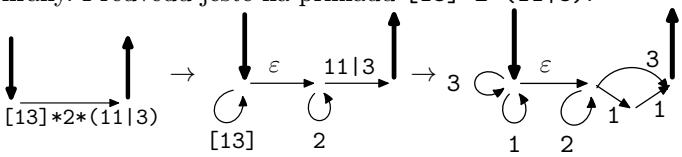


Uvedenému výrazu po krátkém zkoumání vyhovují všechny řetězce sestavené z permutací `123`, na obrázku jedničky symbolizují posun doprava, dvojky vlevo dolů a trojky vlevo nahoru. Abych tedy po každém třetím znaku byl v počátečním stavu, musím projít nějakou permutací `123`...

Poslední úkol nebyl tak jednoduchý na analýzu. Bylo potřeba aplikovat postup, který jsme si právě předvedli, jenže obráceně. Víc asi napoví obrazový materiál.



Postup je tedy jednoduchý. Hvězdičku rozbalím na cyklus, znaky ze začátku a konce výrazu převedu na samostatné hrany. Více hvězdičkovaných výrazů oddělím ε-hranou. Kdyby se objevilo více možností, udělám z nich paralelní hrany. Předvedu ještě na příkladu `[13]*2*(11|3)`:



Nejasnosti a upřesnění řeším standardně na fóru, nebojte se zeptat. Můžete si také stáhnout zdrojové kódy obrázků (Metapost).[9]

Jan „Moskyto“ Matějka

23-2-1 Program (Balíčky balíčků) C

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

// struktura, reprezentující sloupec, do kterého se umíme dostat
struct Solution
{
  // první řádek, na který se umíme dostat
  int Level;
  // číslo balíčku, který jsme použili na přístup do aktuálního sloupce
  int LastBundle;
};
```

[9] `http://ksp.mff.cuni.cz/tasks/23/s2327.mp`

```
// N ze zadání
int N;
// počet použitých druhů balíčků
// (druhá polovina balíčků obsahuje jen balíčky s hmotností, která už v první polovině je)
int bundlesLength;
// pole předpočítaných velikostí balíčků
int* bundles;
// pole řešení pro sloupce
struct Solution* solutions;
// číslo aktuálního průchodu, tedy čísla řádku
int currentLevel;

// funkce, která označí sloupce dostupného ze zadaného
void newSolutions(int pos)
{
  for (int i = 1; i < bundlesLength; i++)
  {
    int newPos = (pos + bundles[i]) % N;
    int newLevel = currentLevel + (pos + bundles[i]) / N;
    if (newPos != 0 && (solutions[newPos].Level == 0 || solutions[newPos].Level > newLevel))
    {
      solutions[newPos].Level = newLevel;
      solutions[newPos].LastBundle = i;
    }
  }
}

int main(void)
{
  // H ze zadání
  int H;

  FILE* input = fopen("balicky.in", "r");
  fscanf(input, "%d %d", &N, &H);

  int evenN = N % 2 == 0;

  bundlesLength = (N + 1) / 2;

  bundles = (int*)malloc(sizeof(int) * bundlesLength);

  // předpočítání velikostí balíčků
  for (int i = 0; i < bundlesLength; i++)
  {
    bundles[i] = (N - i) * (i + 1);
    if (evenN)
      bundles[i] /= 2;
  }

  // pokud je N sudé, sloupce s lichým číslem nejsou dostupné a můžeme je tedy zcela ignorovat,
  // což uděláme tak, že N i H snížíme na polovinu
  if (evenN)
  {
    H = (H + 1) / 2;
    N /= 2;
  }

  // číslo řádku cílového políčka
  int quotient = H / N;
  // číslo sloupce cílového políčka
  int remainder = H % N;

  solutions = (struct Solution*)calloc(N, sizeof(struct Solution));

  currentLevel = 0;

  solutions[0].LastBundle = 0;
  solutions[0].Level = 1;

  newSolutions(0);

  int done = 0;
  int currentPosition;
```

```
  // průchod seznamem sloupců
  while (!done)
  {
    currentLevel++;
    for (currentPosition = 0; currentPosition < N; currentPosition++)
    {
      if (solutions[currentPosition].Level > 0 && solutions[currentPosition].Level <= currentLevel)
      {
        if ((currentPosition == remainder && solutions[currentPosition].Level <= quotient)
          || (currentLevel > quotient)
          || (currentLevel == quotient && currentPosition >= remainder))
        {
          done = 1;
          break;
        }
      }
      if (solutions[currentPosition].Level == currentLevel)
        newSolutions(currentPosition);
    }
  }

  int* counts = (int*)calloc(bundlesLength, sizeof(int));

  // započítání balíčků velikosti N
  if (quotient > currentLevel)
    counts[0] = quotient - currentLevel;

  // započítání ostatních balíčků zpětným průchodem
  int current = currentPosition + currentLevel * N;
  while (current > 0)
  {
    struct Solution solution = solutions[current % N];
    int bundle = solution.LastBundle;
    counts[bundle] += 1;
    current -= bundles[bundle];
  }

  FILE* output = fopen("balicky.out", "w");

  for (int i = 0; i < bundlesLength; i++)
    if (counts[i] != 0)
      fprintf(output, "%d %d\n", counts[i], i + 1);

  return 0;
}
```

23-2-4 Program (Plánování) — Python

```
import queue

vstup = input()
dleZacatku = list()

i = 0
for dvojice in vstup.split(" "):
    dleZacatku.append(dict(
            zacatek = int(dvojice.split("-")[0]),
            konec = int(dvojice.split("-")[1]),
            id = i   ))
    i += 1

dleKoncu = sorted(dleZacatku, key=lambda dvojice: dvojice["konec"])

volnaLetadla = queue.Queue(0)
potrebnychLetadel = 0

letovePlany = list()
udalostiNaLetadla = [0]*i
```

```
i = 0; j = 0
while len(dleZacatku) > i :
# Pokud jsme odeslali všechna letadla, nepotřebujeme dál simulovat přistávání.
    if dleZacatku[i]["zacatek"] < dleKoncu[j]["konec"] :
        if volnaLetadla.empty() :
            volneLetadlo = potrebnychLetadel
            potrebnychLetadel += 1
            letovePlany.append([dleZacatku[i]])
        else :
            volneLetadlo = volnaLetadla.get()
            letovePlany[volneLetadlo].append(dleZacatku[i])
        udalostiNaLetadla[dleZacatku[i]["id"]] = volneLetadlo
        i += 1
    else :
        pristavajiciLetadlo = udalostiNaLetadla[dleKoncu[j]["id"]]
        volnaLetadla.put(pristavajiciLetadlo)
        j += 1

i = 1
print(potrebnychLetadel)
for plan in letovePlany :
    print(str(i) + ": ", end="")
    for udalost in plan :
        print(str(udalost["zacatek"]) + "-" + str(udalost["konec"]), end=" ")
    print()
    i += 1
```

23-2-5 Program (Zaměřování) — Pascal

```
const
   MaxN = 1000;
   Nekonecno = 1E10;  {"nekonečné" t u průsečíku - prostě nenalezen}

type Vektor = record
   X,Y:real;
   end;

var
   Kanon:array[1..2] of Vektor;
   Vrcholy:array[0..MaxN] of Vektor;
   Vrcholu:integer;
   PozadovanyObsah:real;

   Spojnice:Vektor;
   Normala:Vektor;
   C:real; {poslední parametr k popisu přímky spojující kanóny}
   t:real; {t průsečíku}

   i:integer; {index ...}

   NejlepsiNalezenyObsah,SoucasnyObsah:real;
   VhodnyVrchol:integer; {vrchol, který zatím dosáhl nejpřesnějšího obsahu}

function SkalarniSoucin(V1,V2:Vektor):real;
begin
   SkalarniSoucin:=V1.X*V2.X+V1.Y*V2.Y;
end;

function Prusecik(P,Q:Vektor;Normala:Vektor;c:real):real;
{spočtě t průsečíku úsečky spojující body P a Q s přímkou určenou normálou a c}
var Jmenovatel:real;
    SmerovyVektor:Vektor;
begin
   SmerovyVektor.X:=Q.X-P.X;
   SmerovyVektor.Y:=Q.Y-P.Y;
   Jmenovatel:=SkalarniSoucin(SmerovyVektor,Normala);
   if Jmenovatel=0 then Prusecik:=Nekonecno {rovnoběžky - nezajímavé}
   else  Prusecik:=-(c+SkalarniSoucin(P,Normala))/Jmenovatel;
end;
```

```
procedure NactiVstup;
var i:integer;
begin
   readln(Kanon[1].X,Kanon[1].Y);
   readln(Kanon[2].X,Kanon[2].Y);
   readln(PozadovanyObsah);
   if PozadovanyObsah<0 then PozadovanyObsah:=0; {záporný obsah by dělal dále potíže}
   readln(Vrcholu);
   for i:=1 to Vrcholu do readln(Vrcholy[i].x,Vrcholy[i].Y);
   Vrcholy[0]:=Vrcholy[Vrcholu];
{speciální příapad úsečky spojující první a poslední vrchol}
end;

begin
   NactiVstup;
   Spojnice.X:=Kanon[2].X-Kanon[1].X;
   Spojnice.Y:=Kanon[2].Y-Kanon[1].Y;
   Normala.X :=-Spojnice.Y;
   Normala.Y :=+Spojnice.X;
   C:=-SkalarniSoucin(Kanon[1],Normala);
   for i:=1 to Vrcholu do begin
      t:=Prusecik(Vrcholy[i],Vrcholy[i-1],Normala,C+2*PozadovanyObsah); {průsečík s rovnoběžkou}
      if (t>=0) and (t<=1) then begin {průsečík existuje}
         writeln(Vrcholy[i].X*(1-t)+Vrcholy[i-1].X*t,',',Vrcholy[i].Y*(1-t)+Vrcholy[i-1].Y*t); exit;
      end;
      t:=Prusecik(Vrcholy[i],Vrcholy[i-1],Normala,C-2*PozadovanyObsah); {průsečík s rovnoběžkou}
      if (t>=0) and (t<=1) then begin {průsečík existuje}
         writeln(Vrcholy[i].X*(1-t)+Vrcholy[i-1].X*t,',',Vrcholy[i].Y*(1-t)+Vrcholy[i-1].Y*t); exit;
      end;
   end;
   {žádný průsečík nenalezen, projdeme tedy vrcholy znovu}
   NejlepsiNalezenyObsah:=abs(SkalarniSoucin(Vrcholy[1],Normala)+C)/2;
   VhodnyVrchol:=1;
   for i:=2 to Vrcholu do begin
      SoucasnyObsah:=abs(SkalarniSoucin(Vrcholy[i],Normala)+C)/2;
      if abs(SoucasnyObsah-PozadovanyObsah)<abs(NejlepsiNalezenyObsah-PozadovanyObsah) then begin
         NejlepsiNalezenyObsah:=SoucasnyObsah;
         VhodnyVrchol:=i;
      end;
   end;
   writeln(Vrcholy[VhodnyVrchol].X,',',Vrcholy[VhodnyVrchol].Y);
end.
```

23-2-6 Program (Testovací) C

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

int posl[10000];
int N;

// iterování proměnné přes úseky stejné hodnoty
int zvys(int x) {
  do { x++; } while(x<N && posl[x]==posl[x-1]);
  return x;
}

int main(void) {
  // načtení vstupu
  scanf("%d", &N);
  for (int i=0; i<N; i++)
    scanf("%d", &posl[i]);

  // výpočet
  for (int i=0; i<N-2; i = zvys(i)) {
    int j = i+1;
    int k = i+2;
    while(k<N) {
      if (posl[j]-posl[i] == posl[k]-posl[j]) {
        printf("%d-%d-%d\n",
                posl[i], posl[j], posl[k]);
        j = zvys(j);
        k = zvys(k);
      }
      else if (
          posl[j]-posl[i] > posl[k]-posl[j])
        k = zvys(k);
      else {
        j = zvys(j);
        if (j>k)
          k = zvys(k);
      }
    }
  }
  return 0;
}
```

Výsledková listina dvacátého třetího ročníku KSP po druhé sérii

		Škola	*ročník*	*sérií*	*2321*	*2322*	*2323*	*2324*	*2325*	*2326*	*2327*	*série*	*celkem*
1.	Jakub Zíka	GNAlejíPH	4	2		9,5	12	9		10	12	43,9	89,9
2.	Lukáš Folwarczný	GKomHavíř	3	3		10		10		10	12	42,0	86,7
3.	Vojtěch Hlávka	GŠlapanice	2	7	2	6	12	10	6	10	11	43,4	86,5
4.	Juda Kaleta	GKlatovy	2	3	9		11	10		7	12	43,4	84,7
5.	Michal Anderle	GTim_Lučen	4	2				10	8	6	12	38,4	82,0
6.	Filip Hlásek	GMikulášPL	4	17				10	8	10	12	40,0	81,6
7.	Jan Hadrava	GZborovPH	3	2		10	2,5	10			11,5	37,1	81,5
8.	Štěpán Šimsa	GJungmanLT	2	11	2	10		10	8	7		35,0	79,6
9.	David Bernhauer	GZborovPH	3	2		7		6	0	2,5	11	34,1	75,9
10.	Matěj Kocián	GLesníZlín	4	4	10	9			8	4		33,9	75,6
11.	Vojtěch Sejkora	SPSE_Pard	2	2	0	6		9	7	4,5	11,8	37,8	75,4
12.	Jerguš Greššák	GRaymanaPV	2	3		6		10			11,3	30,0	72,6
13.	Peter Zeman	GAnVra	4	2	2			7		5	12	32,9	71,1
14.	Michal Pokorný	SŠkybernHK	3	2		7		9			10,5	30,3	70,2
15.	Martin Raszyk	G_Karvina	1	2	7	2		6		6	6,7	35,6	68,8
16.	Ondřej Hübsch	GArabskáPH	1	7	10		0,5			6	4	23,5	64,7
17.	Jindřich Pilař	GBroumov	3	3		2	2	8	8	3,5		27,4	56,1
18.	Ondřej Mička	GJírovcČB	2	6				10			6,2	18,2	52,9
19.	Vojtěch Kletečka	GHavlBrod	3	2		2			5	3	8,3	27,5	52,6
20.	Ondřej Cífka	GNAlejíPH	2	4		7					10,5	20,1	51,5
21.	Jan Bok	GJungmanLT	4	3	0						12	12,0	50,3
22.	Andrej Mariš	PriorPC	3	2	2							4,2	45,2
23.	Jiří Eichler	SlovanGOL	3	7	10						10,7	21,2	43,9
24.	David Krška	GJirsíkaČB	4	1								0,0	40,3
25.	Jan Paštyka	SPSKutHora	2	2				6		3	6,7	23,9	39,8
26.	Ondřej Fiedler	GJungmanLT	4	3	2			10	7	9,5		31,6	39,3
27.	Rastislav Rabatin	GJHroncaBA	2	1								0,0	37,4
28.	Filip Matzner	GJirsíkaČB	4	2								0,0	34,5
29.	Robin Mana	GValašKlob	4	2				6		2		12,6	32,3
30.	Jiří Setnička	G25březnPH	4	12				10			12	22,0	32,0
31.	Matěj Židek	GBroumov	3	3				6		2,5		12,9	29,9
32.	Milan Berka	G_Krumlov	4	1								0,0	29,8
33.	Daniel Švec	SPŠERožnov	3	1								0,0	27,9
34.	Daniel Stahr	GJungmanLT	4	4		7		10	8			26,7	26,7
35.	Michal Punčochář	GJírovcČB	1	1								0,0	24,2
36.	Tomáš Varga	GMost	-1	2								0,0	22,0
37.	Jonatan Matějka	GJírovcČB	1	5							10,5	11,3	21,7
38.	Jitka Fürbacherová		2	1			10			6,5		20,4	20,4
39.	Tereza Hulcová		3	1			10			5,5		19,8	19,8
40.–42.	Anna Dresslerová	GJHroncaBA	4	2								0,0	19,0
	Milan Mikuš	GĽŠtúraTN	3	2				10				10,0	19,0
	Mária Mrocková	GJHroncaBA	4	3								0,0	19,0
43.	Tomáš Velecký	GBezručeFM	0	2							10,6	11,6	17,3
44.	Matouš Kozma		4	1				7	8			17,1	17,1
45.	Jiří Šebele	GArabskáPH	1	1								0,0	14,3
46.–47.	Martin Mach	GJírovcČB	3	4								0,0	10,0
	Alexander Mansurov	GNVPlániPH	2	4								0,0	10,0
48.	Josef Klesa	GKlatovy	3	1				8				9,5	9,5
49.	Pavel Kratochvíl	VOŠGSvětlá	3	10		9						9,1	9,1
50.	Martin Holec	GSlavičín	4	8								0,0	8,7
51.	Jan Lejnar		1	1						6		8,6	8,6
52.	Tomáš Turlík	GRaymanaPV	2	1								0,0	8,4
53.	Barbora Hourová	G_Brandýs	4	1								0,0	5,7
54.	Patrik Jung		1	1				2				4,5	4,5