

Korespondenční Seminář z Programování

ZAČÁTEČNICKÁ KATEGORIE

29. ročník

KSP-Z

Březen 2017

Milí řešitelé, milé řešitelky!

Blíží se jaro, Velikonoce, KSPřácké jarní soustředění (pojetel)... ale před tím vším vám přinášíme autorská řešení úloh ze třetí série. Doufáme, že vnesou světlo do skončené série a pomohou (nejen) se sériemi příštími. Také vás tradičně vyzýváme, pokud máte jakoukoliv otázku či nejasnost, přejte se! Obrátit se na nás můžete přes fórum na našich stránkách nebo e-mailem na ksp@mjf.cuni.cz.

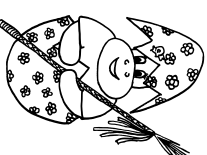


Řešení třetí série začátečnické kategorie 29. ročníku KSP

29-Z3-1 Želva na dvořku

Úloha byla opět jen drobným rozšířením Krocení zlé želvy z druhé série. Želva se chovala úplně stejně, přišly pouze překážky. Přesto ale řešení neodbudeme a ukážeme si ještě jeden, trochu jiný, způsob simulace polohy želvy.

V minulém řešení jsme si pamatovali aktuální souřadnice želvy jako dvě čísla a směr želvy jako číslo třetí. Pak jsme měli připravená pole, která říkála, v jakém směru máme o kolik změnit jednotu ze souřadnic.



Tento přístup se hodí, pokud například chceme procházet všech 8 políček, na které můžeme skočit šachovým koněm z aktuálního políčka. Souřadící programu bude statické pole obsahující políčka, na které může kůň skočit ze souřadnic [0, 0]. Pokud zovra nestojí na tomto políčku, stačí souřadnice patřičně posunout.

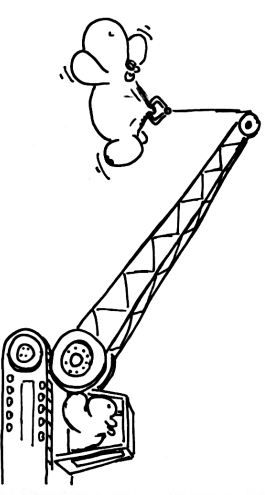
Nám ale stačí pamatovat si aktuální směr želvy, a proto můžeme program na pohled trochu zjednodušit. Nebudeme si směr pamatovat jako nějaké magické číslo, ale jako směrový vektor. Slovní spojení je to děsivé, ale vysvětlení je prosté. Směrový vektor je rozdíl příštího a aktuálního políčka – pokud tedy želva udělá krok. Želva je na začátku otočená na sever, tomu odpovídá vektor (0, 1).

Na povel *krok* zareagujeme snadno, prostě směrový vektor přičteme k aktuální souřadnici. Otáčení je složitější na pochopení, ale o to snazší na napsání.

Pokud je směrový vektor (dx, dy) a želva se otáčí za levou rukou, vezmeme jako nový směrový vektor $(-dy, dx)$. Vyzkoušejte, funguje to. Můžete si zkusit napsat směrové vektory pro všechny čtyři směry a uvidíte, že vždy obsahují jednu nulu a jednu (mínus) jedničku. Prohleďte si si, jak otáčením jednička „cestuje“.

¹ <http://ksp.mff.cuni.cz/viz/kucharky/zakladni>

² <http://ksp.mff.cuni.cz/viz/29-21-5/reseni>



Teď ale to hlavní: překážky. Potřebujeme zavítat, aby želva nedochlázla staré ně. Představme si na chvíli, že umíme snadno poznat, jestli na daném políčku je překážka. Potom si můžeme před každým krokem spočítat pozici cílového políčka a počítat se, jestli je obsazeno překážkou. Pokud ano, započítáme naražení do překážky a želva se nepohne. Pokud ne, změníme souřadnice želvy na ty vypočtené.

Jak ale poznat, jestli je na zvoleném políčku překážka? Pokud si všichni překážky vložíme za sebe jen tak do pole, budeme jej muset před každým krokem celé proйт. To by mělo časovou složitost lineární s počtem překážek, $O(P)$ pro každý krok želvy. Tedy až $O(P \cdot N)$ pro celý běh programu. Pokud si ale souřadnice překážek po načtení lexikograficky seřadíme, můžeme v nich hledat binárním vyhledáváním.¹ Tím se poměrně snadno dostaneme na časovou složitost $O(N \log P + P)$, což je rozhodně o něco lepší. Pokud přemýšlíte proč $+P$, pak protože musíme překážky také načíst, což trvá $O(P)$ (a P může být větší než $N \log P$, takže se to do prvního členu neschová).

Naprogramovat či popsat binární vyhledávání je dobré cvičení, které bychom po vás chtěli v teoretické domáči úloze. Pokud ale přiseté program, můžete využít knihovnu všeho programovacího jazyka. Existuje totiž často používaná datová struktura, která umí prvky rychle přidávat a testovat jejich existenci. Běžné se jí říká množina, *set*.

Zejména v Pythonu je její použití extrémně jednoduché. Opravdu stačí místo hranatých závorek (případně funkce `list`) použít funkci `set`, která vrátí množinu prvků z parametru. Testování na existenci funguje úplně stejně jako s polem pomocí operátoru `in`.

Program (Python 3):

<http://ksp.mff.cuni.cz/viz/29-23-1.py>

29-Z3-2 Písemka z angličtiny

Uvedomiti si přimocáré řéšení není vřbec těžké. Stačí si uložit všechna slova do pole, pro každé jednotlivé slovo pole projít a počítat si jeho prefixy. Pak vypíšeme slovo s nejvyšším počtem nalezených prefixů. Celý výpočet proběhne v $\mathcal{O}(N^2)$, kde N je součet délek všech slov.

Zkusíme najít něco lepšího. Pokud jste si vzpomněli na řešení úlohy 29-Z1-5,² správně vás napadlo použít *trie*. Pro připomenutí: *trie* je zakoreněný strom, kde vrcholy reprezentují písmena a každé slovo ve slovníku představuje cestu z kořene do listu. Každý vrchol tedy můžeme chápat jako prefix; musíme si ale označit vrcholy, kde nějaké slovo končí.

Vytrobíme si *trie* ze všech slov na vstupu. Ke každému vrcholu chceme určit S , počet slov ze slovníku, která jsou jeho prefixem. Proto projdeme *trie* do hloubky: do zásobníku si kromě vrcholu uložíme i jeho S . Na začátku bude zásobník obsahovat kořen *trie* a $S = 0$. Cílem je najít slovo s nejvyšším S .

Jeden krok příchodu se bude skládat z vyjmutí vrcholu a jeho S ze zásobníku. Pokud ve vrcholu končí slovo, zvýšíme S o 1. Pak do zásobníku vložíme všechny potomky vrcholu s výpočtem S .

Po celou dobu si ukládáme vrchol s nejvyšším S a jakmile skončíme příchod, vypíšeme slovo, které vrcholu odpovídá. Abychom to mohli udělat, musíme se z vrcholu vrátit až do kořene a po cestě si ukládat písmena.

To, že ve vrcholu s nejvyšším S končí slovo ze slovníku, nemusíme vůbec kontrolovat. Stačí si uvědomit, že to vždy bude list stromu, tedy vrchol bez potomků (a v něm vždy nějaké slovo končí).

Paměťová složitost je jistě $\mathcal{O}(N)$, vrcholů nebude více, než je součet délek slov. Časová složitost však závisí na tom, jakým způsobem budeme ve vrcholech ukládat odkazy na potomky.

Používáme anglickou abecedu s malými znaky, proto je potomků nejvýše 26. Můžeme ve vrcholech vytvořit pole odkazů s touto délkou: k vrcholu sice dokážeme přistoupit v konstantním čase, ale spotřeba paměti nezohledí narost (pro slovník „normálních“ slov bude obvykle obsazena jen malá část pole).

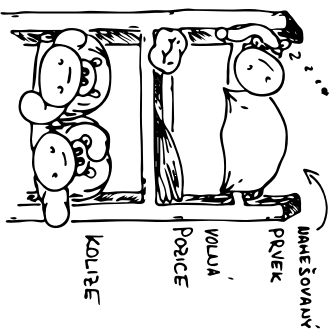
Lepší možností je si ukládat ukazatele do vyváženého binárního stromu, případně využít hešování. U první možnosti bude vytvoření *trie* i příchod v $\mathcal{O}(N \cdot \log 26) = \mathcal{O}(N)$, u druhé také v $\mathcal{O}(N)$.

Autorský zdrojový kód využívá hešování. V Pythonu se hešování přimocáré, obvykle se ale této struktúře říká *slovník* (*dictionary*). Hoří se však vědět, jak funguje uvnitř, což popisujeme v naší hešové kuchařce.³

Program (Python 3):
<http://ksp.mff.cuni.cz/viz/29-Z3-2.py>

Kuba Maroušek

³ <http://ksp.mff.cuni.cz/viz/kucharka/hesovani>



29-Z3-3 Šestkové čísla

Řešení úlohy si rozdělíme na dvě části. Nejprve šestkový zápis převedeme na desítnou hodnotu a poté pro výsledné číslo vypíšeme jeho hlavní zápis.

Pokud by šestkové číslce byly pouze v sestupném pořadí, stačilo by projít vstup a sečíst jejich hodnoty. Může se ale stát, že skupin stejných šestkových čísel následuje číslce vyšší hodnoty a my musíme předchozí odečíst.

Budeme si proto držet dvě pomocné proměnné h a p . První h bude označovat hodnotu poslední číslce, kterou jsme viděli, druhá p pak kolik těchto čísel jsme viděli nepřerušovaně za sebou.

Vstup projdeme číslci po číslci a budeme průběžně aktualizovat pomocné proměnné h a p , která drží celkový výsledek. Pokud je aktuálně zpracovávaná číslce hodnoty h , stačí p zvýšit o jedna.

Pokud je nižší hodnoty, k celkovému výsledku přičteme hodnoty čísel reprezentovaných v pomocných proměnných. Poté nastavíme p na jedničku a h na hodnotu aktuální.

Pokud je hodnota aktuální číslce vyšší, přičteme k výsledku hodnotu aktuální číslce a odečteme hodnotu těch, které jsou reprezentovány pomocnými proměnnými. Vímě, že v kontu připadá následující číslce určitě bude menší než aktuální, proto stačí nastavit pomocné proměnné p na nula a h na hodnotu aktuální číslce. Díky tomu se při příštím příchodu pomocné proměnné nastaví na nové hodnoty.

V druhé části máme číslo (pojmenujme jej c), které chceme vypsat v hlavním šestkovém zápisu. Představme si, že nemůžeme odečítat. Pak stačí procházet číslce od největšího po nejmenší. V každém příchodu pak, dokud je c větší než aktuální číslce, odčítáme od c tuto číslci a zároveň ji vypisujeme na výstup.

My ale můžeme i odečítat, což potřebujeme, abychom se vyhnuli zakázaným čtyřem stejným číslcím za sebou. Označme si hodnotu aktuální číslce jako a . Pokud je $c \geq a$ zachováme se stejně jako v předchozím odstavci.

Dále si všimneme, že nejmenší hodnotou, kterou můžeme reprezentovat aktuální číslci spolu s odčítáním, je $\frac{2}{3}a$. Pokud je tedy $c < \frac{2}{3}a$, nic neděláme a jdeme zpracovávat další číslci v pořadí.

Pokud je $\frac{1}{6}a \leq c < a$ vypíšeme aktuální číslci a před ní jednu ($c < \frac{5}{6}a$) či dvě ($c \geq \frac{5}{6}a$) číslce o jedna menšího

řádku. Zároveň od c odečteme a a přičteme vypsané menší číslke. Všimněme si, že potom nutně $c < \frac{1}{4}a$, a proto se nám nestane, že bychom nějakou číslíci odečeli a následně opět přičeli.

Program (Python 3):

<http://ksp.mff.cuni.cz/viz/29-23-3.py>

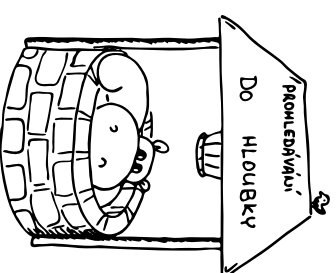
Honzá Križák

29-23-4 Zdobení stromčeku

Rozmyslíme si, že si můžeme vánoční stromček představit jako graf, kde jednotlivé větve stromčeku jsou vrcholy. Potom každý přidávaný rezež bude tvořit v tomto grafu hranu. Chceme zjistit, přidáním které hrany vznikne v grafu kružnice.

Pojďme si nejprve vyřešit jednodušší úlohu: dostali jsme nějaký graf a chceme zjistit, jestli v něm kružnice je. Na to nám stačí použít prohlédávání do šířky či hloubky.⁴ Ke každému navštívenému vrcholu si uložíme informaci, odkud jsme přišli. Pak se podíváme na všechny jeho sousedy. Pokud některý z nich už byl navštívený a není to ten, ze kterého jsme právě přišli, znamená to, že jsme objevili cyklus. Tohle ověření zvládneme v lineárním čase, $O(N + M)$, kde N je počet vrcholů a M je počet hran.

Nyní se nabízí jednoduché řešení celé úlohy: postupně přidáváme do grafu hrany a po každém přidání ověříme, jestli už v grafu není cyklus. Ale pak ověření provádíme celkem M -krát, časová složitost je tedy $O(M \cdot (N + M))$. To je celkem pomalé.



Zkusme to lépe. Všimněme si jedné věci: v průběhu přidávání hran nějakou dobu graf pořád neobsahuje cyklus, pak se tam objeví a od té doby už graf stále obsahuje cyklus.

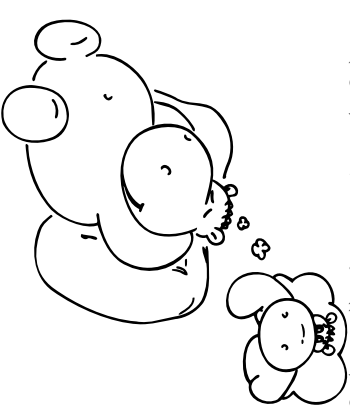
My hledáme předěl mezi těmito dvěma stavy: K tomu můžeme použít binární vyhledávání. Zkusíme do grafu přidat prvích $M/2$ hran a podíváme se, jestli obsahuje cyklus. Pokud ano, musel vzniknout přidáním některé z prvích $M/2$ hran, jinak musí vzniknout přidáním některé z druhých $M/2$.

Předpokládejme, že nastal první případ. Rozpíšme tedy interval od 1 do $M/2$ – vytvoříme nový graf, do kterého přidáme jen prvích $M/4$ hran. Opět ověříme, jestli obsahuje kružnici, a podle toho víme, že proviniáá hrana leží buď v intervalu od 1 do $M/4$, nebo od $M/4$ do $M/2$. Takhle pokračujeme v půlení, dokud nám nezůstane jediná hrana – ta

hledaná.

Abychom nemuseli graf stavět pořád znovu a znovu, stačí si u každé hrany pamatovat „čas“, ve kterém vznikla. Binární vyhledávání pak odpovídá plnění časových intervalů. Samotné prohlédávání grafu na existenci cyklu upravené tak, aby ignorovalo hrany, které teprve vzniknou.

Jeden pokus nám trvá lineární čas a binární vyhledávání provede $O(\log M)$ kroků, dohromady $O((N + M) \log M)$.



Ukážeme si ještě jedno, avšak pokročilejší, řešení. K tomu se nám bude hodit pojem *komponenta souvislosti* (viz opět grafovou kuchařku).

Na začátku jsou všechny vrcholy izolované a každý z vrcholů tvoří svou vlastní komponentu souvislosti. Pojďme se podívat, co přidávání hran do grafu s těmito komponentami provede.

První možnost je, že jsme přidali hranu mezi dvěma vrcholy, které patří do různých komponent. Potom jsme mezi nimi vytvořili novou cestu, a tedy jsme tyto dvě komponenty spojili v jednu.

Jestliže však byly oba vrcholy součástí stejné komponenty, existovala již dříve mezi nimi nějaká cesta. Potom přidáním této hrany vznikne mezi oběma vrcholy druhá cesta, která spojí s první vytvoří cyklus.

Díky tomuto pozorování můžeme místo hledání cyklu v grafu kontrolovat, zda před přidáním každé hrany byly oba vrcholy ve stejné komponentě souvislosti. Jak takový stav ale zjistíme?

Mohli bychom jednoduše kontrolovat, zda jsou oba vrcholy součástí stejné komponenty pomocí prohlédání do hloubky nebo do šířky. Avšak takové prohlédání máže potenciálně projít celý graf. Časová složitost takové kontroly je tedy $O(N + M)$, která se pro každou hranu seče na časovou složitost algoritmu $O(M(N + M))$. To jsme si moc nepomohli.

Pojďme to zkusit lépe použitím učého, což už vymysleli informatiči před námi. V algoritmu používáme dvě základní operace, a to přidání hrany a zjišťování komponenty souvislosti dvou vrcholů. Tento operacím se v informatice hanýváce říká *union* a *find*. Existuje datová struktura zvaná *DFU*, která je umí provádět velmi efektivně. Je popsána v naší kuchařce o minimálních kostkách.⁵ Využitím této struktury se poté úloha změní na postupně volání *find* a *union* pro každou hranu. Časová složitost takového algoritmu poté bude $O(M \log N)$.

⁴ <http://ksp.mff.cuni.cz/viz/kucharcky/grafy>

⁵ <http://ksp.mff.cuni.cz/viz/kucharcky/kostky>

Program (Python 3):
<http://ksp.mff.cuni.cz/viz/29-23-4.py>

Václav Kováčik

29-23-5 Prvočíselné rozklady

V úloze jsme po vás chtěli najít postup, kterým rozložíte přirozené číslo na součin prvočísel. Odborněji řečeno se tak řešeným procesem říká *faktORIZACE* a patří k velmi složitým problémům matematiky. Dodnes nikdo nezná algoritmus, který by dokázal faktorizovat velká (řádově tisícířadová) čísla dost rychle na to, abylohm se dočkali výsledku. To v vlastnosti se využívá v mnoha šifrovacích algoritmech, mimo jiné i ve známém RSA.



Popíšeme si autorské řešení. Na začátku běh programu dostaneme číslo M a máme silibno, že jakékoli číslo x , které dostaneme k faktorizaci, bude menší: Jeho rozklad se totiž bude skládat z prvočísel menších než M . Pokud bychom je všechna dokázali najít, můžeme rozklad snadno určit: postupně projdeme prvočísla a každým z nich dělíme x tak dlouho, dokud je prvočíslem dělitelné.

Také si uvědomíme, že nám ve skutečnosti stačí prvočísla menší nebo rovna \sqrt{M} . Větší prvočísla se v rozkladu nachází nejvýše jednou a získáme ho jako to, co nám zbude z x , když už nejdě dělit žádným z prvočísel menších než \sqrt{M} (rozmyslete si).

Jeden rozklad bude trvat čas $O(k + \ell)$, kde k je počet prvočísel menších než \sqrt{M} a ℓ je délka výsledného rozkladu. Rozklad nebude obsahovat více než $O(\log M)$ prvočísel – každý čísel je alespoň dvojnásobek. Odhadnout k šlora pomocí M je těžké, spokojíme se tedy s $O(\sqrt{M})$.

Ale jak prvočísla najít? Použijeme Eratosthenovo síto, které možná znáte z hodin matematiky. S troškou a papírem funguje následovně: najdív si vypíšeme do řady všechna čísla od 2 do \sqrt{M} . Postupně budeme čítk zakrouzkovat všechna prvočísla a škrtnout všechna složená.

Jeden krok algoritmu vypadá takto: vezmeme první neznačené číslo (ani zakrouzkované ani škrtnuté) a zakrouzkujeme ho. Potom škrtneme všechny jeho násobky (pokud jsme zakrouzkovali k , škrtneme $2k, 3k, \dots$).

Takže v prvním kroku zakrouzkujeme dvojnásobky a škrtneme všechna sudá čísla. V druhém zakrouzkujeme trojnásobky a škrtneme všechny násobky tří. A tak dále. Vyzkoušejte a rozmyslete si, že to funguje.

Pokud ho chceme naprogramovat, místo papíru použijeme pole síto velikosti \sqrt{M} . V něm na i -té pozici bude jednička, pokud číslo i je škrtnuté, jinak nula. Navíc si budeme pamatovat poslední nalezené prvočísla p . Pak je postup jednoduchý:

1. Vypíšeme pole síto samými nulami (na začátku není nic škrtnuté).

⁶ <http://ksp.mff.cuni.cz/viz/24-3-5/resi>
⁷ <http://ksp.mff.cuni.cz/viz/kuchariky/grafy>

2. $p \leftarrow 2$
3. Opakujeme dokud $p \leq \sqrt{M}$:

4. Přidáme p do seznamu prvočísel.
5. Škrtneme všechny násobky p : nastavíme $sito[p]=1$, $sito[3p], sito[4p], \dots$ na 1.
6. Posuneme se na nejbližší neškrtnuté číslo napravo: zvyšujeme p , dokud nebude $sito[p] = 0$.

Časová složitost síta velikosti \sqrt{M} je $O(\sqrt{M} \cdot \log \log \sqrt{M})$: důkaz není úplně jednoznačný, takže ho zde nebudeme uvádět. Můžete si ho přečíst v řešení úlohy 24-3-5.⁶

Tímto způsobem vyřšíme úlohu s $O(\sqrt{M})$ pomocné paměti, $O(\sqrt{M} \cdot \log \log \sqrt{M})$ času na převýpočet a $O(\sqrt{M})$ času na jeden dotaz. Pokud si ale dovolíme použít více paměti a čas na převýpočet, dostaneme další velmi dobrý algoritmus.

Když už složené číslo x škrtneme, známe prvočísla p_i , které ho děl. Pokud si ho zapamatujeme jako p_{x_i} , bude spočítání rozkladu triviálně jednoduché: Dostaneme-li x , do rozkladu přidáme p_{x_i} . Opakujeme s x/p_{x_i} , dokud nám nezbude prvočísla. Takto rozklad spočítáme v $O(\log x)$, tedy $O(\log M)$.

Bohužel tím musíme rozšířit síto na všech M čísel. To nás bude stát $O(M)$ pomocné paměti a výpočtem síta strávíme $O(M \cdot \log \log M)$ času. I tak se to ale vyplatí, pokud bude počet dotazů N dostatečně velký.

Program (Python 3):
<http://ksp.mff.cuni.cz/viz/29-23-5.py>

Kruba Marušek

29-23-6 Table s dominem

Ačkoli to tak na první pohled nevyypadá, tato úloha je grafová. Pokud toho o grafech ještě nevíte dost, určité si přečtete naši kuchařku o grafech.⁷

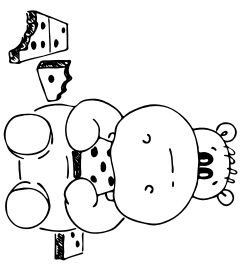
Jaký graf si tedy pod zadáním představíme? Obyčejný neorientovaný. Vrcholy nám budou tvořit symboly, které se vyskytují na domnu. Každá kostička obsahuje právě dva symboly, tedy nám v grafu budou tvořit hrany. Nebo lépe, hrany budou dlny kostiček, které máme k dispozici.

Nyní si vyberte vrchol v grafu, a zkusíte přes hrany přecházet do jiných vrcholů. Taková posloupnost vrcholů a hran tvoří *stez* v grafu. Čemní to odpovídá v domnové analogii? Možná jste uhodli že postaveným hadům z cukroví.

Každý had vypadá jako sled v našem grafu a naopak, každý sled jde postavit jako hada z domna. Naši kamarádi se v příběhu dostali do sítnace, ve které nemonli dva hady spojit žádným „mezihadem“. To znamená, že mezi koncovými symboly neexistoval v našem grafu žádný sled.

To se v grafu může stát pouze tehdy, pokud je graf *nesouvislý*. To znamená, že symboly jde rozřítit na (alespoň) dvě hrnadelky tak, aby každá kostička používala symboly pouze z jedné a žádná kostička dvě hrnadelky nepropojuje.

Mimo jiné to znamená, že dva hadi z příběhu nemohli žádný symbol společný – jinak by šlo hady propojit přes něj.



Řešením úlohy je tedy vytvořit graf a ověřit, zdali je souvislý. To jde provést velice snadno prohlédáním, například do

hloubky. Během prohlédávání budeme vrcholy obarvovat, a pokud nám na konci zbyl nějaký neobarvený, graf je nutně nesouvislý; a naši kamarádi si musí dát příšně větší pozor. Postavení grafu a jeho prohlédání stihneme v $O(N + M)$, kde N je počet vrcholů a M počet hran. Protože ale graf nemá izolované vrcholy (každý symbol se musel vyskytnout na nějaké kostičce), musí být $M \geq N$ a tedy je časová složitost pouze $O(M)$. Při rozumné reprezentaci v paměti nám bude stačit i $O(M)$ paměti.

Program (Python 3):
<http://ksp.mff.cuni.cz/viz/29-23-6.py>

Ondra Hlavatý



matfyz

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.

Webové stránky:

<https://ksp.mff.cuni.cz/>

E-mail:

ksp@mff.cuni.cz

Diskusní fórum:

<https://ksp.mff.cuni.cz/forum/>

Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – jeho SHA1 fingerprint je: E9:DB:EE:C6:62:BC:14:DE:09:E4:88:97:DC:36:0E:87:B3:50:B0:01.