Korespondenční Seminář z Programování

37. ročník | **KSP** | **Prosinec 2024**

Vzorová řešení druhé série třicátého sedmého ročníku KSP

37-2-1 Urychlovač částic

K řešení úlohy můžeme použít dynamické programování.[1] To trochu připomíná důkaz indukcí – Nejprve triviálně vyřešíme nějakou zmenšenou verzi problému. Poté toto řešení nějak rozšíříme, aby řešilo o něco větší verzi problému, poté ještě větší, a tak dále, dokud se nedostaneme k původnímu problému.

Úlohu si nejprve zjednodušíme tak, že budeme chtít jen spočítat součet úžasností v nejlepší posloupnosti přeměn, aniž bychom tuto posloupnost nalezli. Zato řešení spočítáme pro každý cílový prvek, ne jen pro hélium.

Nejprve musíme najít nějaký způsob, jak úlohu zmenšit. Nabízí se tři možnosti:

1. Vyřešit úlohu pro menší počet prvků P.
2. Vyřešit úlohu pro menší počet nastavení urychlovače N.
3. Vyřešit úlohu pro menší počet provedených přeměn K.

První dvě možnosti sice vypadají lákavě, ale není snadné existující řešení rozšířit přidáním prvku nebo nastavení. Zaměříme se tedy na třetí možnost.

Součet úžasností nejlepší posloupnosti k přeměn, která z vodíku vyrobí prvek s protonovým číslem p, si označíme jako $dp[k][p]$. Pokud taková posloupnost neexistuje, pak bude $dp[k][p] = -\infty$. Naším konečným úkolem je spočítat $dp[K][2]$.

Pro $k = 0$ úlohu vyřešíme snadno. Pomocí nula přeměn můžeme získat jen vodík, a to se součtem úžasností přeměn 0, proto bude $dp[0][1] = 0$. Ostatní prvky vůbec vyrobit nemůžeme, tedy pro $p \geq 2$ bude $dp[0][p] = -\infty$.

Teď si musíme rozmyslet, jak řešení pro k přeměn rozšířit na řešení pro $k+1$ přeměn. Posloupnost $k+1$ přeměn získáme tak, že vezmeme posloupnost k přeměn a přidáme další přeměnu. Pro každý prvek se tedy podíváme na všechna nastavení n, která ho umí vyrobit. Pro každé z nich spočítáme součet jeho úžasnosti a úžasnosti nejlepší posloupnosti délky $k-1$, která vyrobí potřebný vstupní prvek. Úžasnost chceme maximalizovat, takže z těchto potenciálních úžasností vezmeme tu největší. Získali jsme tedy tento vzorec:

$$dp[k][p] = \max_{\text{přeměna } a_n \to p} dp[k-1][a_n] + u_n$$

Maximum z prázdné množiny definujeme jako $-\infty$.

Získali jsme tím tento jednoduchý algoritmus:

1. Pro k od 0 do K:
2. Pro p od 1 do P:
3. $dp[k][p] \leftarrow -\infty$
4. $dp[0][1] \leftarrow 0$
5.
6. Pro k od 1 do K:
7. Pro p od 1 do P:
8. Pro každé nastavení n vytvářející prvek p:
9. $dp[k][p] \leftarrow \max(dp[k][p], dp[k-1][a_n] + u_n)$

Pro implementaci tohoto algoritmu potřebujeme umět rychle najít všechna nastavení, která vyrobí daný prvek. Nastavení bychom si mohli snadno předem roztřídit, ale snazší řešení je pro každé k jednou projít všechna nastavení a každé započítat do správné hodnoty dp:

1. Pro k od 0 do K:
2. Pro p od 1 do P:
3. $dp[k][p] \leftarrow -\infty$
4. $dp[0][1] \leftarrow 0$
5.
6. Pro k od 1 do K:
7. Pro n od 1 do N:
8. $dp[k][v_n] \leftarrow \max(dp[k][p], dp[k-1][a_n] + u_n)$

Tento algoritmus bude mít časovou složitost $\mathcal{O}(KP+KN)$, paměťovou $\mathcal{O}(KP+N)$.

Ještě musíme tento algoritmus upravit tak, aby kromě spočítání úžasnosti nejlepšího řešení toto řešení také našel. Stačí, když si u každé hodnoty v dp navíc zapamatujeme, pomocí kterého posledního nastavení jsme jí dosáhli. To nám umožní postupně posloupnost přeměn zrekonstruovat odzadu dopředu.

Program (C++):
`http://ksp.mff.cuni.cz/viz/37-2-1.cpp`

Úlohu připravili: Michal Kodad, Ben Swart

37-2-2 Pohoří plné jezer

Pojďme nejprve rozebrat, jak zaručeně určit, kolik maximálně vody se může nacházet na jednom vybraném políčku pohoří.

Víme, že se voda udrží na nějakém políčku pouze v případě, že nedokáže nějakou cestou přetéct přes okraj pohoří.

Uvažme potom libovolné políčko p. Nechť h_p je nejvyšší možná nadmořská výška hladiny vody nacházející se na tomto políčku a odpovídá součtu jeho výšky a výšky největšího sloupce vody, který na něm ještě může ležet.

Všimneme si, že nám libovolná cesta z p na okraj pohoří poskytuje horní odhad maximálního množství zadržené vody na políčku p. Je-li totiž výška nejvyššího políčka na cestě z p na okraj rovna m, nemůže h_p tuto výšku nikdy přesáhnout. To platí, jinak by v takovém případě existovala cesta, na které by se nenacházela žádná překážka vysoká, jako je

[1] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

nadmořská výška hladiny vody na políčku p, a nějaká voda by musela nutně odtéci.

Pokud bychom pro políčko p vybrali ze všech možných cest na okraj (těch bude konečně mnoho), takovou cestu, pro kterou platí, že je maximum z jejich políček nejnižší, získali bychom nejnižší a tedy i nejpřesnější horní odhad.

Dokažme, že tento odhad odpovídá skutečnému množství zadržené vody.

Předpokládejme pro spor, že je maximální množství zadržené vody na políčku p nižší než náš odhad. Pokud bychom potom na prázdné políčko p nalili množství vody odpovídající hornímu odhadu a maximální množství zadržené zadržené vody by bylo nižší, musela by nějaká voda nutně odtéci z políčka p pryč. Víme však, že voda může odtéci jedině po nějaké cestě. Na takové cestě by ovšem muselo být každé políčko menší, než nadmořská výška hladiny, jinak by byl někde proud vody zadržen. Nalezli jsme tedy cestu z vybraného políčka na okraj pohoří, jejíž maximum je menší než náš nejnižší horní odhad. To je ovšem spor s předpokladem, že jsme původně nalezli takovou cestu, jejíž nejvyšší políčko je nejnižší možné.

Máme tedy zaručený postup, jak nalézt pro libovolné políčko maximální množství vody, které může zadržet. Stačí nalézt pro dotyčné políčko takovou cestu na okraj, jejíž nejvyšší políčko je nejnižší možné.

Výška nejvyššího políčka této cesty pak bude odpovídat nadmořské výšce hladiny na políčku p a odečtením výšky samotného políčka získáme výšku vodního sloupu a tedy i jeho objem.

Pro větší přehlednost textu budeme cestu, jejíž nejvyšší políčko je nejmenší možné, odteď označovat jako nejnižší. Dále budeme hodnotu nejvyššího políčka libovolné cesty nazývat její výškou.

Reprezentace pomocí vhodného grafu

Doposud jsme pracovali s grafem, ve kterém jsou trochu neobvykle ohodnoceny vrcholy místo hran. Navíc pracujeme s okrajem pohoří, který byl dosud v našich úvahách zastoupen všemi políčky na obvodu pohoří.

Pokusme se tedy, ještě než se pustíme do samotné řešení, problém lehce přeformulovat do takové podoby, o které se nám bude lépe přemýšlet. To uděláme reprezentací pohoří pomocí vhodného grafu.

Tento graf bude obsahovat za každé políčko jeden vrchol a každé dva vrcholy budou propojeny hranou, pokud spolu odpovídající políčka sousedí stranou.

Navíc do tohoto grafu zavedeme nový vrchol, který bude představovat okraj pohoří a bude propojen se všemi vrcholy po obvodu pohoří.

Nakonec ohodnotíme každou hranu maximem z hodnot vrcholů, které propojuje. V případě ohodnocení hran vedoucích z vrcholu okraje si můžeme představit, že je hodnota vrcholu okraje rovna nule.

Nahlédneme, že je maximum z hodnot políček na libovolné cestě a maximum z ohodnocení hran na téže cestě v novém grafu totožné.

Problém nalezení nejnižší cesty z políčka p do okraje je tedy ve staré i nové reprezentaci ekvivalentní, jen v nové reprezentaci plyne ohodnocení z jednotlivých hran cesty a ne vrcholů.

Řešení založené na Dijkstrově algoritmu

Nejprve náš problém převedeme do výše uvedené grafové reprezentace. Naším cílem nyní bude pro každý vrchol políčka zjistit výšku nejnižší z něj do okraje vedoucí cesty. K tomu použijeme upravený Dijkstrův algoritmus, který místo délky nejkratších cest z vybraného vrcholu do všech ostatních bude hledat výšku nejnižších cest.

Ten spustíme z vrcholu okraje. Platí totiž, že je kvůli obousměrnosti hran každá nejnižší cesta z vrcholu v do vrcholu okraje zároveň nejnižší cesta vedoucí z okraje do vrcholu v, a naopak.

Pro každý vrchol si budeme v průběhu algoritmu pamatovat, jaká je výška nejnižší aktuálně známé cesty vedoucího z vrcholu okraje. Na počátku bude mít vrchol okraje hodnotu záporného nekonečna a ostatní vrcholy budou ohodnoceny nekonečnem.

Zároveň budeme pro každý vrchol rozlišovat tři stavy. První bude stav odpovídá tomu, že nebyl dosud nalezen. Jakmile vrchol nalezneme, označíme jej jako otevřený a naopak v okamžiku, kdy si budeme jisti s jeho ohodnocením, vrchol označíme jako uzavřený.

Jediný vrchol, který bude na začátku otevřený, bude vrchol okraje. Zbývající vrcholy označíme jako nenalezené.

V každém kroku algoritmu vždy vybereme otevřený vrchol u, do kterého vede cesta s aktuálně nejmenší výškou. Tento vrchol prohlásíme za uzavřený. Následně projdeme jeho otevřené a nenavštívené sousedy.

Všechny nenavštívené sousedy vrcholu u otevřeme. Následně pro každého jeho souseda zjistíme nejnižší známou výšku cesty, který nejprve vede z okraje do vrcholu u a následně vstoupí souseda přes hranu, jež je propojuje. To uděláme tak, že vezmeme maximum z nejnižší známé cesty vedoucí do vrcholu u a ohodnocení hrany, která jej spojuje s aktuálním sousedem. Pokud je tato hodnota nižší, než nejlepší známá cesta vedoucí do tohoto souseda, aktualizujeme ji.

Pojďme dokázat, že na konci běhu algoritmu budeme pro každý vrchol znát výšku nejnižší cesty vedoucí do vrcholu okraje.

Nejprve nahlédneme, že ohodnocení vrcholů, v tom pořadí, v jakém je uzavíráme, nikdy neklesne. To platí, neboť ve chvíli, kdy nějaký vrchol uzavřeme, je jeho ohodnocení nejmenší ze všech ostatních otevřených vrcholů a každé nové ohodnocení, kterým tento vrchol nebo libovolný z ostatních otevřených vrcholů ohodnotí některého ze svých sousedů, bude vyšší nebo stejné.

Dále si všimneme, že každé ohodnocení odpovídá výšce nějakého skutečně existujícího sledu. Rozmysleme si, že lze každý sled zjednodušit na cestu, aniž by došlo k zhoršení hodnocení. Nemůže tedy nastat situace, kdy by algoritmus ohodnotil nějaký vrchol nižší hodnotou, než je výška jeho nejnižší cesty.

Stále ovšem mohl být nějaký vrchol ohodnocen vyšší hodnotou, než je ta správná, případně nemusel být navštíven vůbec. Předpokládejme nyní pro spor, že existuje nějaký vrchol, jehož nalezená hodnota je vyšší než výška nalezené nejnižší cesty, případně nekonečno, protože nebyl dotyčný vrchol vůbec navštíven.

Uvažme libovolnou nejnižší cestu vedoucí z tohoto vrcholu na okraj. Vrcholu na této cestě, jehož ohodnocení odpovídá skutečné výšce nejnižší cesty, říkejme dobrý, jinak dotyčný vrchol nazvěme špatným.

Nejnižší cesta zaručeně obsahuje dobrý vrchol, to je vrchol okraje a špatný vrchol, to je náš vrchol se špatným hodnocením.

Zaměřme se potom na k okraji nejbližší špatný vrchol na této cestě. Ten zaručeně sousedí s dobrým vrcholem, který se nachází blíž k okraji. Protože byl tento dobrý vrchol navštíven, nemohlo se stát, že by nebyl špatný vrchol navštíven. Špatný vrchol tak byl alespoň jednou ohodnocen. Špatné ohodnocení by potom mohl získat jedině tehdy, kdyby byl uzavřen dřív než sousední dobrý vrchol, jinak by byl správně ohodnocen dobrým vrcholem. Na nejnižší cestě je skutečné ohodnocení neklesající a ohodnocení špatného vrcholu je ostře větší než to skutečné, takže byl špatný vrchol uzavřen s větší hodnotou a tedy i později než dobrý vrchol.

Potom by ale dobrý vrchol tento vrchol ohodnotil správnou výškou a musel by být též dobrým vrcholem. Došli jsme k hledanému sporu.

Algoritmus opravdu nalezne pro každý vrchol hledanou výšku do něj vedoucí cesty.

Pojďme zrekapitulovat celé řešení.

Nejprve v $\mathcal{O}(RS)$ postavíme grafovou reprezentaci pohoří, následně spustíme upravený Dijkstrův algoritmus a získáme pro každé políčko maximální možný objem vodního sloupu. Nakonec v $\mathcal{O}(RS)$ objemy sečteme.

Pokud Dijkstrův algoritmus implementujeme pomocí binární haldy, dosáhneme časové složitosti $\mathcal{O}(RS \log RS)$ a paměťové $\mathcal{O}(RS)$.

Dodejme, že není nutné v implementaci stavět celý graf. Taková implementace je třeba v našem vzorovém zdrojovém kódu.

Řešení založené na minimální kostře

Na tuto úlohu existuje ještě jeden pohled, tentokrát založený na minimální kostře.

Platí totiž, že stačí vzít libovolnou minimální kostru našeho grafu a cesta mezi každým vrcholem a vrcholem okraje bude vždy nejnižší.

Toto tvrzení dokážeme sporem. Mějme libovolnou minimální kostru a předpokládejme, že existuje vrchol, pro který platí, že cesta v minimální kostře mezi ním a vrcholem okraje není nejnižší.

Vezměme nyní libovolnou nejnižší cestu do tohoto vrcholu a vyberme z ní ty hrany, které neleží v minimální kostře. Každou z těchto hran zkusíme vložit do minimální kostry. Tím nám vždy vznikne v grafu cyklus. Pokud je v tomto cyklu tato hrana největší, zahodíme ji a zkusíme do grafu vložit další. V případě, že v cyklu existuje nějaká ostře větší hrana než námi přidaná, můžeme ji odstranit, zbavit se cyklu a získat kostru s nižší velikostí. To by byl ovšem spor s předpokladem, že byla původní kostra minimální.

Tato situace tedy nesmí ani jednou nastat. Pokud by ovšem byla každá z vyzkoušených hran největší hranou ve svém cyklu, musela by cesta po hranách v minimální kostře být nejvýše tak vysoká, jako ta nejnižší, což je zase spor s předpokladem, že minimální kostra nejnižší cestu neobsahuje.

V každém případě jsme došli ke sporu, takže je tvrzení dokázáno.

V našem řešení tak nejprve nalezeneme v grafové reprezentaci problému nějakým algoritmem minimální kostru.

Ve vzniklé kostře pak již je třeba jen spočítat výšku cesty mezi každým vrcholem a vrcholem okraje a získat tak objem vodního sloupu nad každým políčkem. To lze udělat například pomocí průchodu do šířky začínajícím ve vrcholu okraje.

Časová a paměťová složitost algoritmu závisí na algoritmu použitém k hledání minimální kostry, neboť všechny ostatní kroky jsou lineární v čase i paměti s velikostí pohoří.

Pokud použijeme k hledání minimální kostry algoritmus z kuchařky, získáme řešení běžící v čase $\mathcal{O}(RS \log RS)$ s paměťovou složitostí $\mathcal{O}(RS)$.

Dodejme pro zajímavost, že spolu oba uvedené způsoby řešení úzce souvisí. Řešení založené na Dijkstrův algoritmu totiž odpovídá hledání minimální kostry pomocí Jarníkovo algoritmu, ve kterém rovnou počítáme výšku nejnižších cest.

Lineární řešení

Úlohu lze řešit ještě trochu rychleji. Použijeme k tomu předchozí řešení založené na minimální kostře.

V něm nás nejvíce zdržovalo samotné hledání minimální kostry, neboť ostatní části algoritmu byly lineární s velikostí pohoří.

Minimální kostru lze sice nalézt pomocí složitějších algoritmů v obecných grafech rychleji, než to umí náš algoritmus z kuchařky, my místo toho nicméně využijeme poznatku, že má náš graf poměrně specifický tvar.

Odpovídá totiž čtvercové mřížce s jedním vrcholem navíc, který je propojen s každým vrcholem na obvodu této mřížky. Takový graf lze nakreslit bez křížení hran a je tedy rovinný.

K nalezení minimální kostry v našem grafu tentokrát použijeme Borůvkův algoritmus. Popíšeme jen jeho průběh, případný důkaz správnosti lze nalézt například v Medvědovo Průvodci labyrintem algoritmů.

Borůvkův algoritmus ve svém průběhu udržuje les, do kterého postupně přidává vhodné hrany z grafu tak dlouho, dokud se z tohoto lesa nestane minimální kostra.

Toto činí v jednotlivých iteracích. Na počátku první iterace udržovaný les odpovídá izolovaným n vrcholům grafu. V každé iteraci potom algoritmus vybere pro každý strom v aktuálním lese nejlehčí hranu, která dotyčný strom propojuje s nějakým jiným stromem lesa. Všechny takové hrany do lesa algoritmus následně přidá.

Pokud jsou hodnoty hran unikátní, nevznikne nám po žádné iteraci přidáním hran cyklus a počet stromů tohoto lesa se pokaždé alespoň dvakrát zmenší, dokud po $\mathcal{O}(\log n)$ iteracích nevznikne minimální kostra.

My ovšem nemáme unikátnost hran vůbec zaručenou. Můžeme si pomoci tak, že jednotlivé hrany ohodnotíme uspořádanou trojicí, jejíž první hodnota bude původní ohodnocení hrany a druhé dvě budou odpovídat vrcholům, mezi kterými hrana vede. Takové ohodnocení hran již bude zaručeně unikátní. Hodnoty hran budeme porovnávat lexikograficky.

Borůvkův algoritmus dále upravíme, aby každý strom lesa udržoval kontrahovaný do jednoho vrcholu. Iterace v takto upraveném grafu bude vypadat tak, že si každý vrchol vybere nejlehčí incidentní hranu, tyto hrany algoritmus zkontrahuje a zapamatuje si, že patří do minimální kostry.

Popišme, jak provést rychle potřebné kontrahování.

Nejprve prohledáme les vzniklý po první části iterace a přiřadíme každému vrcholu číslo komponenty, v níž se nachází.

Následně přečíslujeme hrany podle čísel komponent.

Hrany spojující vrcholy v téže komponentě poté přečíslujeme zpět (informace o vrcholech, které hrany na začátku propojovaly jsou díky naší úpravě již obsaženy v jejich ohodnocení) a přidáme je do seznamu hran minimální kostry. Ve zbytku algoritmu s nimi již nebudeme pracovat.

Zbyly nám jen hrany spojující jednotlivé stromy, nicméně následkem přečíslování mohou být některé z nich násobné. Zbavit se jich můžeme lexikografickým přihrádkovým tříděním. Pomocí něj dostaneme násobné hrany k sobě. Nyní stačí projít posloupnost hran a z každého úseku násobných hran ponechat pouze tu s nejmenší hodnotou.

Pojďme určit, jakou bude mít takto upravený Borůvkův algoritmus na rovinných grafech časovou složitost.

Zkusme nejprve odhadnout, s kolika nejvýše hranami náš algoritmus bude v jednotlivých iteracích pracovat.

K tomu využijeme dvou užitečných vlastností rovinných grafů. Jednak platí, že rovinný graf zůstane po každém kontrahování hran nadále rovinný. Pro každý rovinný graf o n vrcholech navíc platí, že obsahuje méně než $3n$ hran.

Z předchozích poznatků plyne, že v každé iteraci pracujeme s lineárním počtem hran vzhledem k počtu vrcholů. Přihrádkové třídění tak bude běžet v lineárním čase vzhledem k počtu vrcholů a tedy i hran. Stejně rychlé budou i zbylé části iterace.

Celkovou časovou složitost získáme sečtením počtu operací přes jednotlivé iterace. Jejich počet v každé iteraci zhora odhadneme pomocí horního odhad počtu hran, to je trojnásobek počtu vrcholů, násobený nějakou konstantou. Víme, že se počet vrcholů v každé iteraci alespoň dvakrát zmenší. Tím dostáváme pro graf o n vrcholech geometrickou posloupnost, která se sečte na $\mathcal{O}(n)$.

Protože náš graf obsahuje celkem $\mathcal{O}(RS)$ vrcholů, bude výsledná časová složitost $\mathcal{O}(RS)$, což je asymptoticky optimální.

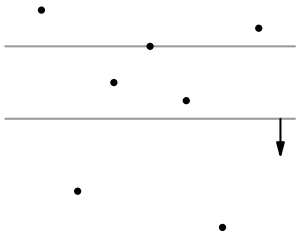
Program (Python):
`http://ksp.mff.cuni.cz/viz/37-2-2.py`

Úlohu připravili: Daniel Culliver, Michal Kodad, David Kolář

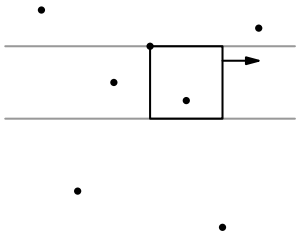
37-2-3 Chov ovcí

Pojďme najít osově zarovnaný čtverec $A \times A$ s právě K body uvnitř. Na to použijme známou geometrickou techniku, totiž zametání roviny.[2] Všimněme si totiž, že pokud najdeme libovolný čtverec odpovídající naším požadavkům, tak ho můžeme posouvat doleva, dokud by posunutí nepřidalo nebo neodebralo nějaký bod. To samé platí pro posunutí nahoru. Tedy alespoň jedna z vertikálních hran bude (nebo těsně nebude) mít na sobě bod. A to samé platí pro jednu z horizontálních.

A jak na to? Budeme si udržovat pás výšky A, který bude obsahovat všechny body se souřadnicemi mezi y_p a $y_p + A$. Tímto pásem budeme zametat rovinu a pro každou pozici pás zkoušet hledat čtverec, který má horizontální hrany na pásu. V předchozím odstavci jsme zdůvodnili, že nám stačí uvažovat pouze pozice čtverce, kde na jedné hraně těsně bude, nebo nebude bod. Tedy zajímavé situace nastávají pouze, když nám bod do pásu přibude nebo vypadne.



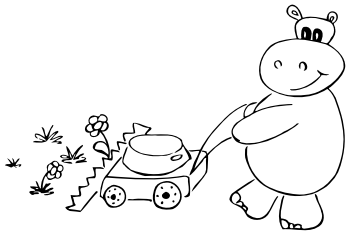
A tedy pro každou takovouto pozici pásu stačí vyzkoušet všechny možné pozice čtverce. Pokud si aktuální body v pásu budeme udržovat v seřazené podle x (sekundárně podle y) v binárním vyhledávacím stromě, stačí najít bod s souřadnicí x splňující: Čtverec s levým horním rohem (y_p, x) obsahuje K bodů. (Zase použijeme argument s posunutím.)



A jak najdeme daný čtverec? Projdeme všechny body v pásu. – Pokud má čtverec začínat na souřadnici x_i (kde i je pořadí bodu v bvs), pak aby měl K bodů, musí:

- x_i je první bod s takovouto souřadnicí: $x_{i-1} < x_i$ (Jinak bychom část bodů s touto souřadnicí vzali a jinou ne.)
- Obsahovat následujících k bodů: $x_{i+k-1} \leq x_i + A$ (Všechny na předchozích indexech mají větší x.)
- Zbylé body neobsahovat: $x_{i+k} > x_i + A$ (Všechny následující body jsou větší.)

Časovou složitost určíme následovně: Pozic pásu je $\mathcal{O}(N)$, pro každou přidáváme bod do binárního vyhledávacího stromu ($\mathcal{O}(\log N)$) a zkoušíme všechny pozice čtverců ($\mathcal{O}(N)$). Celkem $\mathcal{O}(N(\log N + N)) = \mathcal{O}(N^2)$.



Zrychlujeme

Na poslední zrychlení si všimněme, že pokud vkládáme nebo odstraňujeme bod do našeho pásu, tak toho nově nalezený čtverec musí využívat. Jinak bychom ho nalezli už dřív. Proto stačí v našem binárním vyhledávacím stromě K pozic doleva a K doprava od našeho přidaného (odstraněného) bodu. Navíc ale potřebujeme do BVS přidat velikosti podstromů, abychom mohli v něm rychle najít bod na daném indexu.

Nicméně si musíme dát pozor na případ, kdy více bodů má stejnou y-novou souřadnici. Pro předchozí řešení jsme mohli všechny body přidat nebo odebrat naráz a poté projít celý strom. Něco podobného uděláme i zde, jen musíme být

[2] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`

opatrnější. Nejdřív si všechna přidání se stejnou souřadnicí seskupíme. Totéž pro všechna odebrání.

Při přidávání skupiny bodů nejdřív všechny body přidáme, a potom si najdeme jejich pozice ve binárním vyhledávacím stromě. Pokud body odebíráme, budeme si vyhledávat pozice, na kterých by ve výsledném stromě byly, kdybychom je tam nechali. (Tedy index nejbližšího většího prvku.) Poté akorát zkontrolujeme okolí nalezené každé pozice.

1. události ← {Přidej bod i, v čase y_i} ∪ {Odeber bod i, v čase $y_i + A$}
2. seřaď(události) ◁ $\mathcal{O}(N \log N)$
3. seskup(události) ◁ $\mathcal{O}(N)$
4. body ← BVS()
5. Pro každou událost $u \in$ události: ◁ $\mathcal{O}(U)$
6. Pokud u je přidání:
7. Pro $b \in$ u.body: ◁ $\mathcal{O}(U_k)$
8. body.přidej(b) ◁ $\mathcal{O}(\log N)$
9. Pro $b \in$ u.body: ◁ $\mathcal{O}(U_k)$
10. $i_b \leftarrow$ body.find(b) ◁ $\mathcal{O}(\log N)$
11. Jinak u je odebrání:
12. Pro $b \in$ u.body: ◁ $\mathcal{O}(U_k)$
13. body.odeber(u.bod) ◁ $\mathcal{O}(\log N)$
14. Pro $b \in$ u.body: ◁ $\mathcal{O}(U_k)$
15. $i_b \leftarrow$ body.nejbližší_větší(u.bod) ◁ $\mathcal{O}(\log N)$
16.
17. Pro $b \in$ u.body: ◁ $\mathcal{O}(U_k)$
18. $tb \leftarrow$ body.podposloupnost$(i_b - k, i_b + k + 1)$ ◁ tb – *testované body,* $\mathcal{O}(\log N + K)$
19. Pro všechna $1 \le j \le 2k + 1$: ◁ $\mathcal{O}(K)$
20. Pokud $tb[j-1] < tb[j]$ a $tb[j+k-1] \le tb[j] + A < tb[j+k]$:
21. Našli jsme čtverec.

Kde U je počet událostí a U_k počet bodů v k-té události. Pro každý bod děláme $\mathcal{O}(\log N + K)$ operací, takže naše nové časová složitost bude $\mathcal{O}(N(\log N + K)) = \mathcal{O}(NK + N \log N)$.

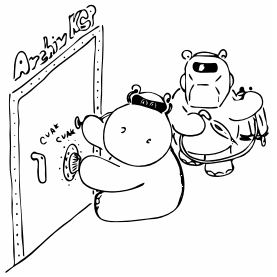
Úlohu připravil: Dan Skýpala

37-2-4 HTTP bludiště

1. úkol: Autentikace

V prvním úkolu na nás server vybafne `401 Unauthorized`. Ale naštěstí v poli hlavičky `X-Hint` napoví, jaké je správné heslo. Tak si vymyslíme nějaké uživatelské jméno (třeba `hroch`), použijeme autentikační metodu Basic a zkusíme to znovu.

Tak jednoduché to ovšem není: server nás vyžene s `403 Forbidden` a vysvětlením, že právo stáhnout si heslo má jedině `root` (správce serveru). Zkusíme tedy změnit uživatelské jméno na `root` a server spokojeně pošle soubor typu `text/plain` s klíčem.



2. úkol: Formulář

Server nás nejdřív přesměruje (`307 Temporary Redirect`) na `.../form.cgi?task=37-2-4&what=key` a následně nám vysvětlí, že používáme špatnou metodu (`405 Method Not Allowed`). Naštěstí, jak standard káže, připojí `Allow:`, kde prozradí, že jediná podporovaná metoda je `POST`, a vysvětlí že očekává `POST` webového formuláře.

Podle kuchařky máme tedy odeslat obsah formuláře jako `Content-Type: application/x-www-form-urlencoded` (jinak dostaneme `415 Unsupported Media Type`). Potom nás server navede odpovědí `422 Unprocessable Content`, že vyžaduje argumenty `task` a `what`. Zkopírujeme je tedy z původního URL a server odpoví přesměrováním `303 See Other` na textový soubor s klíčem.

Mimochodem, kdybyste místo `what=key` poslali `what=hint`, doslechli byste se, že nápovědy nevydáváme :)

3. úkol: PUT

Ve třetím úkolu se opět dozvíme, že metoda `GET` není podporována. Tentokrát máme použít `PUT` a nahrát tak své jméno. Tělo musí mít textový `Content-Type`, jinak ho server odmítne s `415 Unsupported Media Type`. Musíme uvést délku těla (jinak `411 Length Required`) a ta nesmí být větší než 1000 bytů (nechceme-li dostat `413 Request Entity Too Large`). V odpovědi na `PUT` obdržíme heslo.

4. úkol: Varianty

Zde na první pokus server odpoví `300 Multiple Choices` a nabídne nám výběr z několika variant: `minotauros.gif`, `minotauros.xml` a `minotauros.png`.

Požádáme-li o GIF, server ho odmítne vydat s `451 Unavailable for Legal Reasons` a odkazem na známou kontroverzi kolem softwarových patentů na kompresi LZW. Odmítnutí je nicméně doprovázeno omluvou a nabídkou kompenzace ve výši jednoho dolaru. A vskutku: v hlavičce odpovědi objevíme `X-Coin: $1`. Heslo ovšem nikde.

Tak zkusíme XML a dostaneme dokument ve formátu XML, v němž je očividný element `<klic>` s očekávaným obsahem.

Intermezzo: Odznáček

Co kdybychom zkusili ještě PNG? To by nás s omluvou, že dokument byl dočasně přesunut, přesměrovalo (`307 Temporary Redirect`) na `.../img/1/minotauros.png`. Tam bychom dostali přesměrování na `.../img/2/...` atd. Kdo si to vyzkoušel, zjistil, že server umí libovolně velká čísla, takže posloupnost přesměrování je nejspíš nekonečná.

Pokud máme hravou náladu, zkusíme se podívat na konec nekonečné posloupnosti: místo čísla napíšeme `nekonecno`, `nekonečno`, `infinity` nebo ∞ (v UTF-8). To překvapivě funguje a obdržíme `303 Found something at infinity` s `Location:` odkazující na URL typu `mailto:` – požadavek na odeslání mailu organizátorům, jehož předmětem si říkáme o odznáček.

5. úkol: Pigzip

V tomto úkolu nám server hned pošle klíč. Jenže ouha, klíč je ve formátu `application/x-pigzip`, zkomprimovaný do řetězce `42`. `X-See-Also` nám prozradí, že Pigzip je narážka na dávný linuxový comics Hackles,[3] kde prasátko Preston vyvine nový kompresní algoritmus, schopný cokoliv zkomprimovat do 3 bajtů. Jen má zatím potíže s dekompresí ...

[3] `https://web.archive.org/web/20060924193925/http://www.hackles.org/cgi-bin/archives.pl?request=310`

Ani my nedokážeme Pigzip dekomprimovat, tak si všimneme, že v hlavičce odpovědi je ještě `Vary: Accept` a `Accept-Ranges: bytes`. To druhé nás ještě nezajímá, ale první naznačuje, že server nejspíš podporuje domlouvání se na typu dat.

Nevíme ovšem, o jaký typ si říci. `Accept: text/plain` ani nic podobného server není ochoten splnit. Co tedy chceme? Inu, cokoliv kromě Pigzipu. To se dá říci jako `Accept: */*;q=1.0, application/x-pigzip;q=0.9`.

Server málem odpoví s `Content-Type: application/x-ksp-key`, ale v poslední chvíli se zarazí, že je odpověď moc dlouhá (`413 Request Entity Too Large`) a dodá, že nekonečné soubory neposílá. Není divu, že na to potřebovali Pigzip ...

Vzpomeneme si tedy na `Accept-Ranges` a zkusíme poslat požádavek na nějaký interval, třeba `Range: bytes=0-1000`. Dostaneme `206 Partial Content` s tělem, v němž se opakují řádky `klic=...`. Vida.

Dodejme, že kdybychom si řekli o interval delší než 1 KB, dostali bychom také status `413`.

6. úkol: Ach ty keše

V tomto úkolu nám server ochotně pošle požadovaný soubor. Jenže v něm je napsáno, že klíč zatím nebyl nastaven a máme to zkusit znovu po vydání 2. série. `Last-Modified` nám prozradí, že tento soubor opravdu vznikl před vydáním série, a podle přítomnosti `Age` poznáme, že mezi námi a serverem stojí keš, která si soubor zapamatovala. A pamatovat si ho nejspíš bude dost dlouho, protože uvedené `Expires` je v daleké budoucnosti.

Naštěstí můžeme keši pomocí `Cache-Control: no-cache` nebo `max-age` nařídit, aby sehnala čerstvý obsah. V něm už klíč cinká.

7. úkol: Vhoďte minci

V posledním úkolu nám server odmítá vydat klíč, dokud nezaplatíme: `402 Payment Required`. A prý máme vložit minci. Tak si vzpomeneme, že ve 4. úkolu jsme jednu minci získali. Pošleme tedy s požadavkem `X-Coin: $1` a hle – je tu klíč.

Závěrem

Vzorový program si vystačí s pythoní knihovnou `requests`, která všechny potřebné části HTTP buď přímo umí, nebo si je jde objednat ručním nastavením hlavičky.

Program (Python):
`http://ksp.mff.cuni.cz/viz/37-2-4.py`

Úloha je inspirovaná Medvědovou dávnou úlohou v soutěži Po drátě. Děkujeme Jefovi Poskanzerovi za minimalistický webový server thttpd, z nějž naše implementace vychází.

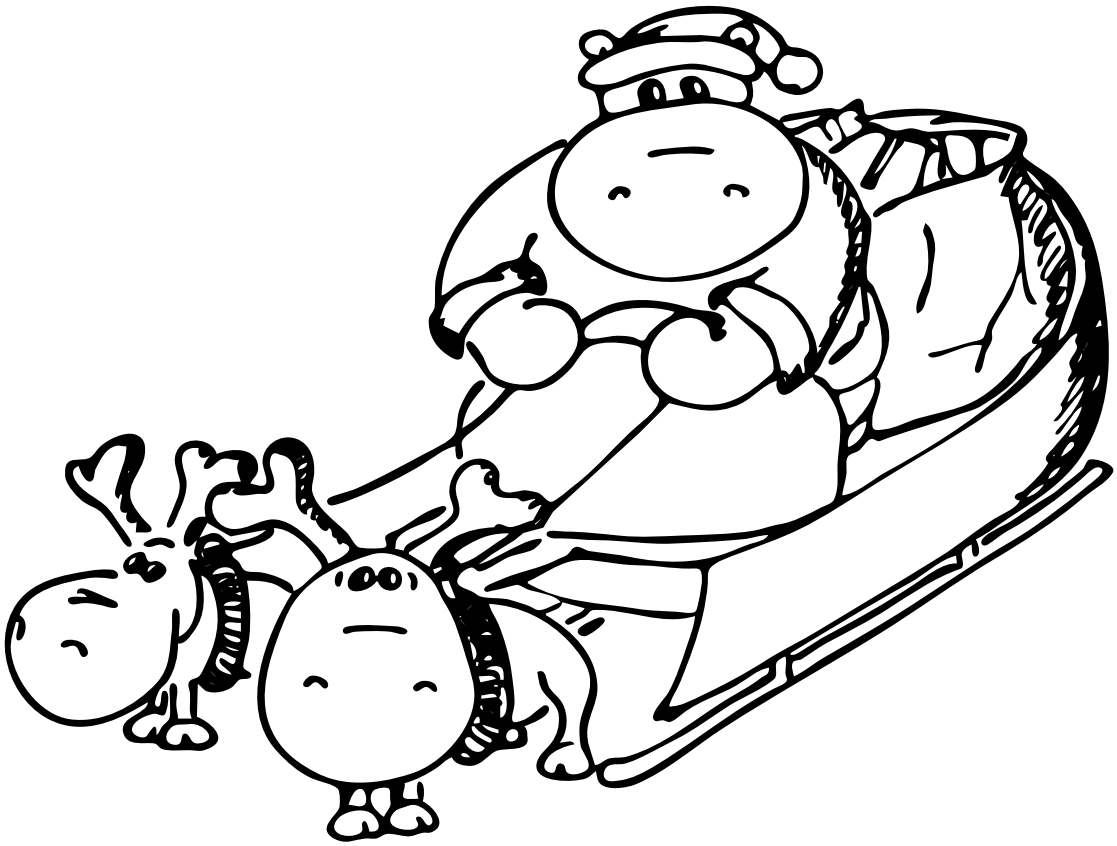
Úlohu připravili: Honza Černohorský, Martin „Medvěd“ Mareš

37-2-X1 Intervalové většiny

Protože se nikomu nepodařilo tuto úlohu vyřešit, rozhodli jsme se, že prodloužíme její deadline do konce 3. série. Navíc vydáváme nápovědu, která vám snad pomůže k řešení.

37-2-S Lexery útočí

Seriál lze odevzdávát za snížený počet bodů až do konce školního roku. Řešitelům, kteří už první sérii odevzdali, rozešleme vzorové řešení napřímo. Pokud chceš tuto sérii přeskočit a začít řešit další díl, můžeš si o vzorové řešení napsat na ksp@mff.cuni.cz.



KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.
Realizace projektu byla podpořena Ministerstvem školství, mládeže a tělovýchovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Organizátoři a kontakty: *https://ksp.mff.cuni.cz/kontakty/*