

Vzorová řešení první série třicátého devátého ročníku KSP

39-1-1 Počet podposloupností KSP

Nejjednodušší je pustit tři vnořené for-cykly, které pro každou trojici pozic $i < j < k$ kontrolují, zda se na ní nevyskytuje K, S a P. Takové řešení běží v čase $\Theta(n^3)$, kde n je délka zadání, a šlo za něj získat až 4 body.

Můžeme si ale všimnout, že zbytečně počítáme tu samou věc opakovaně. Zafixujeme-li pozici K a S, už je jasné, kolik výskytů KSP jako podposloupnost zvládneme vytvořit: tolik, kolikrát se P vyskytuje napravo od našeho S. Podobně, pro fixní pozici S a P, dokážeme vytvořit tolik výskytů KSP, kolikrát se K vyskytuje nalevo od našeho S.

Vytvoříme si tedy dvě pole, K a P . $K[i]$ bude počítat, kolik písmen K se nachází nalevo od i (včetně), kdežto $P[i]$ bude počítat, kolik písmen P se nachází napravo od i včetně. Obě pole umíme spočítat v lineárním čase – pro výpočet K stačí procházet pole zleva a za každý výskyt K zvýšit hodnotu na aktuálním políčku o 1 oproti té předchozí. Pole P spočítáme obdobně. Toto je speciální případ techniky zvané *prefixové* (resp. *suffixové*) součty, o které si můžete přečíst v základní kuchařce.¹

Díky předpočítaným polím můžeme rychle spočítat celou odpověď. Podposloupností KSP, jejichž písmenko S leží na pozici i , je buď 0 (pokud na pozici i není S), nebo přesně $K[i-1] \cdot P[i+1]$. To proto, že ke každé volbě K nalevo si můžeme nezávisle vybrat nějaké P napravo. Výsledný algoritmus jen posčítá tyto hodnoty přes všechna i . Jeho časová i paměťová složitost je pak $\mathcal{O}(n)$.



Dynamické programování

Na tuto úlohu existuje ještě jedno řešení. To je sice mírně složitější, ale zase má tu výhodu, že funguje i tehdy, pokud bychom počítali výskyty slov obecné délky. Pojďme s ním tedy vyřešit úlohu, kde místo KSP počítáme všechny výskyty nějakého slova J délky t . Označme také zadání jako řetězec Z délky n . Obecně, máme-li řetězec R , pak $R[i]$ značí jeho podřetězec skládající se z prvních i znaků.

Pořídíme si dvojrozměrné pole D velikosti $n \times t$. Na políčku $D[i][j]$ si budeme počítat, kolika způsoby můžeme v $Z[:i]$ najít $J[:j]$ jako podposloupnost. Všechny výskyty $J[:j]$ v $Z[:i]$ si pro účely řešení rozdělíme na dva typy:

- *Nové*: ty, které zahrnují $Z[i]$
- *Předchozí*: ty, které nezahrnují $Z[i]$

To nás navádí na myšlenku, jak si problém průběžně dělit a převádět na jednodušší. Zafixujme si nyní j , tedy budeme v Z na všech pozicích i počítat $D[i][j]$. Nejdůležitější myšlenka je, že to počítáme postupně s rostoucím i , tedy v každém okamžiku budeme mít $D[i][j]$ pro menší i již spočítané.

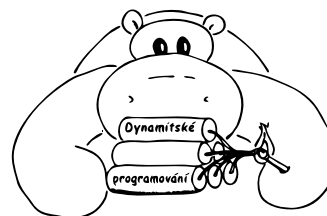
Prvně si všimněme, že všechny *předchozí* výskyty $J[:j]$ již máme v $D[i-1][j]$. Druhé pozorování bude, že *nové* výskyty $J[:j]$ budou končit v textu na pozici i – tedy vůbec nějaké vzniknou, pouze pokud $Z[i] = J[j]$. Potom *nových* výskytů $J[:j]$ je právě tolik, kolik je *předchozích* výskytů $J[:j-1]$, tedy $D[i-1][j-1]$.

Z toho také vidíme, že už potřebujeme mít spočítané výskyty $J[:j-1]$. Vychází nám z toho algoritmus, který je vlastně složen jen z dvou for-cyklů v sobě – iterujeme přes všechna j , kde pro každé j iterujeme přes všechna i a počítáme $D[i][j]$. Pro každou dvojici (i, j) provedeme konstantní práci, dvojic je $n \cdot t$, celkem má tedy algoritmus složitost v $\Theta(nt)$. Speciálně – pro slovo KSP máme tedy složitost v $\Theta(n)$, jako dával první algoritmus.

Pokud dynamické programování ještě neznáte, doporučujeme se začíst do naší kuchařky.²

Program (Python):

<https://ksp.mff.cuni.cz/viz/39-1-1.py>



Další zlepšení, které se nabízí, je paměť. Tabulka D konstruovaná pro dynamické programování má velikost nt , což je pro naše účely lineární. Dá se to ale lépe. Uvědomme si, že pro konstrukci řádku j potřebujeme pouze řádek $j-1$ a všechny předchozí můžeme zahodit. Tím zadarmo dostaneme paměť na $\Theta(n)$, do čehož se vejde i pamatovaný vstupní text.

Nakonec si všimněme, že stejně jako jsme konstruovali tabulku D po řádcích j , můžeme ji konstruovat po sloupcích i . Stejně tak nám bude stačit pamatovat si poslední sloupec. Tím paměť algoritmu dostaneme na $\Theta(t)$, což je pro účely úlohy konstantní! Hurá! Protože znak vstupu na pozici i nás zajímá pouze při konstrukci sloupce i , můžeme číst vstup postupně a ani si ho nepamatovat. Takovému algoritmu se říká, že je *online* – čte vstup jednou a nepamatuje si ho.

Program (Haskell):

<https://ksp.mff.cuni.cz/viz/39-1-1.hs>

Úlohu připravili: Ríša Hladík, Matúš Púll

¹ <https://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy>

² <https://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani>

³ <https://ksp.mff.cuni.cz/viz/kucharky/grafy>

39-1-2 Oboustranně nejbližší hnízdo

Úlohu můžeme přímočaře převést do řeči teorie grafů:³ máme neohodnocený orientovaný graf G a zajímá nás, do jakého vrcholu v máme jít, aby byl součet délek nejkratší cesty z 0 do v a z v do 0 co nejmenší. Máme-li víc možností, máme najít v s co nejmenším indexem.

Jinými slovy, pokud $d_G(u, v)$ označuje délku nejkratší cesty mezi u a v , pak hledáme takový vrchol $v \neq 0$, který minimalizuje hodnotu výrazu $d_G(0, v) + d_G(v, 0)$, a ze všech takových v navíc chceme to s nejmenším indexem.

Na hledání vzdáleností můžeme použít prohledávání do šířky (BFS). Pustíme-li ho z vrcholu 0, spočítáme do každého vrcholu v jeho vzdálenost $d_G(0, v)$. Kdyby byl graf neorientovaný, máme hotovo, neboť $d_G(v, 0) = d_G(0, v)$. Náš graf ale je orientovaný a obecně mohou být vzdálenosti z v a do v velmi různé.

Jedno řešení je prostě každé $d_G(v, 0)$ spočítat zvlášť. To funguje, ale časová složitost je až $\Theta(N(N + M))$, jelikož použijeme prohledávání do šířky z každého vrcholu.

Místo toho si můžeme pomoci jednoduchým trikem: Definujeme graf G^T vzniklý z G tak, že každé hraně obrátíme její orientaci. Pak platí, že $d_G(a, b) = d_{G^T}(b, a)$ – nejkratší cesta z a do b je to samé, jako nejkratší cesta z b do a v grafu, kde můžeme jen couvat. Můžeme teď v G^T pustit prohledávání do šířky z 0. To nám pro každý vrchol v spočte hodnotu $d_{G^T}(0, v) = d_G(v, 0)$, přesně, jak jsme chtěli.

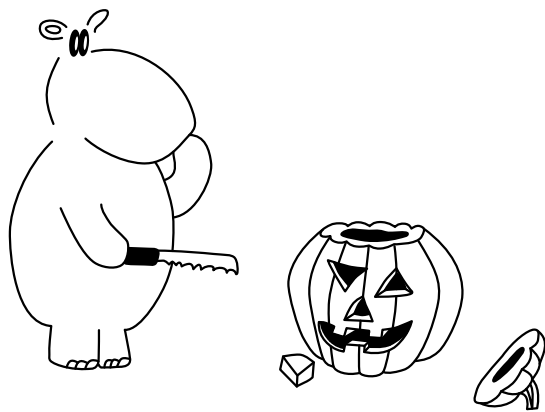
Výsledný algoritmus tedy jen dvakrát pustí prohledávání do šířky z 0 – jednou při něm chodí po směru hran, jednou v protisměru. Tím spočteme vzdálenosti z 0 i do 0 a stačí nám pak najít a vypsat kýžený nejlepší vrchol.

Na závěr poznamenejme, že kdybychom neměli sekundární požadavek na v s co nejmenším indexem, stačilo by úlohu vyřešit nalezením nejkratšího cyklu procházejícího vrcholem 0. K tomu stačí pustit BFS a dívat se, kdy poprvé znovu narazíme na hranu vedoucí do 0. Ve skutečnosti se i tento algoritmus dá upravit, aby počítal řešení naší úlohy, ale detaily zde vynecháme. Můžete je nalézt ve vzorovém řešení.

Program (Python):

<https://ksp.mff.cuni.cz/viz/39-1-2.py>

Úlohu připravili: Ríša Hladík, Matúš Púll



39-1-3 Vysílač

Označme si jako (x_i, y_i) souřadnice i -tého domu a (x_V, y_V) souřadnice vysílače. Úloha po nás chce najít takové souřadnice vysílače, že následující výraz (říkejme mu třeba *ztráta*) je minimalizován:

$$(x_1 - x_V)^2 + (y_1 - y_V)^2 + (x_2 - x_V)^2 + (y_2 - y_V)^2 + \dots + (x_n - x_V)^2 + (y_n - y_V)^2.$$

Nejprve si můžeme všimnout, že ztrátu můžeme přeuspořádat a přirozeně ji rozložit na dvě nezávislé funkce: ztrátu kvůli x a ztrátu kvůli y . Konkrétněji, zadefinujeme-li:

$$f(x) = (x_1 - x)^2 + (x_2 - x)^2 + \dots + (x_n - x)^2, \\ g(y) = (y_1 - y)^2 + (y_2 - y)^2 + \dots + (y_n - y)^2,$$

pak platí, že máme-li vysílač na pozici (x_V, y_V) , je jeho ztráta $f(x_V) + g(y_V)$.

Můžeme tedy vyřešit dva nezávislé problémy: hledání x_V , které minimalizuje f , a hledání y_V , které minimalizuje g . Ve zbytku řešení tedy budeme pracovat pouze s f , a g se pak vyřeší obdobně.

Teď už si jen stačí všimnout, že f je kvadratická funkce s jednou neznámou. Wikipedie⁴ nám dává jednoduchý vzoreček na spočítání jejího minima: funkce $f(x) = ax^2 + bx + c$ nabývá minima pro $x = -b/2a$.⁵ Zbývá vyjádřit f v tomto tvaru: každá závorka se rozepíše jako $x^2 - 2x_ix + x_i^2$. Nenecháme se zmást tím, že x_i (konstanta) vypadá hodně podobně jako x (proměnná) a dostaneme:

$$f(x) = nx^2 - 2 \cdot (x_1 + \dots + x_n)x + (x_1^2 + \dots + x_n^2).$$

Máme tedy $a = n$, $b = -2(x_1 + \dots + x_n)$. Dosazením do vzorečku dostáváme, že optimální x_V je $(x_1 + \dots + x_n)/n$.

Jinými slovy, optimální místo pro vysílač je v průměru všech bodů. (To je ale náhoda, že?)

Náš algoritmus tedy pouze načte body a spočte jejich průměr. Stačí mu projít celý vstup jednou bod po bodu, má tedy časovou složitost $\mathcal{O}(N)$ a paměťovou konstantní.

Na závěr se můžete zamyslet, co se změní, pokud pro ztrátu použijeme jinou funkci, třeba $|x_1 - x_V| + |y_1 - y_V| + \dots$. Dá se výsledek zase nějak jednoduše vztáhnout na původní množinu bodů?

Úlohu připravili: Ríša Hladík, Matúš Púll

39-1-4 Sněhuláci

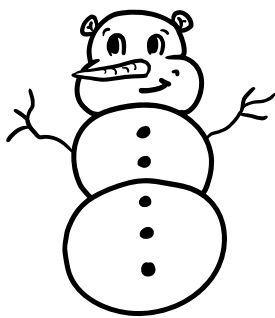
Dle zadání máme sestavit co nejméně sněhuláků – množin koulí S_i takových, že každá obsahuje nejvýše tři koule, jejichž velikosti jsou navíc navzájem různé. Řekněme, že nejmenší dosažitelný počet sněhuláků je M .

Protože velikosti koulí a jejich četnosti budou hrát velkou roli, označme f_r počet koulí, které mají velikost r . Pro tyto četnosti platí $0 \leq f_r \leq N$ a $\sum_r f_r = N$.

Nejprve uvažme případ, když jsou všechny koule navzájem různě velké (tedy všechna $f_r \leq 1$). Pak je můžeme libovolně rozdělit do $\lceil N/3 \rceil$ sněhuláků. Všimněme si, že méně sněhuláků být nemůže, jinak by z principu holubníku musel být alespoň jeden sněhulák složený ze čtyř nebo více koulí.

⁴ https://en.wikipedia.org/wiki/Quadratic_function#Maximum_and_minimum_points

⁵ Se znalostí derivací to jde snadno nahlédnout: spojitá funkce může nabývat minima jen v bodech s nulovou derivací. Musí tedy platit $f'(x) = 2ax + b = 0$. Jde to nahlédnout i bez derivací, doplněním na čtverec, ale do toho se pouštět nebudeme.



Na druhou stranu víme, že sněhulák nemůže obsahovat kouli stejné velikosti dvakrát. To nám dává pro každou velikost r odhad $M \geq f_r$. Tím získáváme dolní odhad na počet sněhuláků $M \geq \bar{M} = \max(\lceil N/3 \rceil, \max_r f_r)$. Ukážeme, že tento odhad je *těsný*, tedy že vždy umíme koule rozdělit do $\bar{M}$ sněhuláků.

Seřadíme koule k_i podle jejich velikostí. Náš algoritmus postaví $\bar{M}$ sněhuláků, toho i -tého postaví z koulí k_i , $k_{i+\bar{M}}$ a $k_{i+2\bar{M}}$, přičemž pokud nějaké z těchto koulí neexistují (indexujeme mimo rozsah $1, \dots, N$), bude sněhulák akorát složen z méně než 3 koulí.

Povšimněme si, že takto každá koule dostane sněhuláka: pro $i < 3\bar{M}$ koule k_{i+1} padne sněhulákovi ($S_{i \bmod \bar{M}+1}$), navíc $3\bar{M} \leq 3\lceil N/3 \rceil \leq N$, takže každá koule bude použita. Máme tedy přiřazení všech N koulí $\bar{M}$ sněhulákům, každý sněhulák má alespoň jednu a nejvýše tři koule.

Zbývá rozmyslet, že každý sněhulák se skládá ze tří různě velkých koulí, tedy že vždy $k_i < k_{i+\bar{M}} < k_{i+2\bar{M}}$. To je snadné: pokud by pro nějaké i platilo $k_i = k_{i+\bar{M}}$, pak díky seřazenosti koulí platí i $k_i = k_{i+1} = k_{i+2} = \dots = k_{i+\bar{M}}$. To znamená, že koulí této velikosti je více než $\bar{M}$, což je spor s definicí $\bar{M}$. Identicky dokážeme, že $k_{i+\bar{M}} < k_{i+2\bar{M}}$.

Zbývá si rozmyslet implementační detaily algoritmu: Nejprve všechny koule seřadíme podle velikosti. Pak každá velikost tvoří souvislý úsek, takže spočítat $\max_r f_r$ jde jednoduchým průchodem celého pole. Z toho dostaneme $\bar{M} = M$, a pak dalším průchodem přiřadíme $S_i = \{k_i, k_{i+M}, k_{i+2M}\}$. Jako bonus každý sněhulák bude mít koule seřazené od nejmenší po největší.

Kromě třídění potřebujeme čas $O(N)$. Jelikož však víme, že velikosti jsou v rozsahu $1, \dots, 10N$, můžeme je třídit počítáním, ze kterého i zdarma dostaneme f_r . Celkovému algoritmu tak stačí $O(N)$ času.

Jako menší poznámku uveďme, že tento algoritmus funguje pro libovolně velké sněhuláky, jen v algoritmu a v důkazu nahradíme 3 libovolnou jinou hodnotou $C \leq N$ a S_i přiřadíme koule $k_i, k_{i+M}, \dots, k_{i+(C-1)M}$.

Úlohu připravili: Katia „Čiči“ Kočická, Matuš Púll

39-1-S Hello, Rust!

Abyste měli na řešení seriálu dostatek času, lze ho odevzdávat až do 15. listopadu. Přejeme příjemný růst!

