Korespondenční Seminář z Programování

ZAČÁTEČNICKÁ KATEGORIE

37. ročník | **KSP-Z** | **Únor 2025**

Řešení třetí série začátečnické kategorie 37. ročníku KSP

37-Z3-1 Wi-Fi ve vlaku

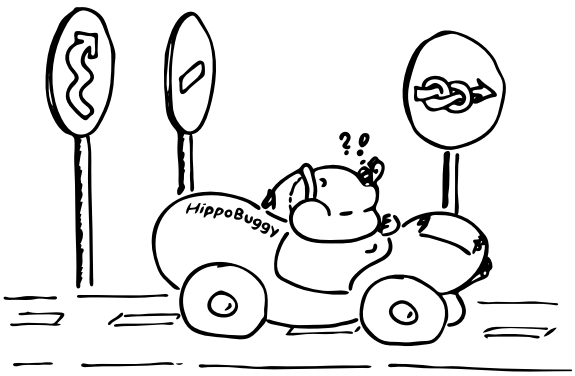
Naším cílem bude zjistit počet dobrých a špatných jednotek v každé řadě, potom všechny řady projít a spočítat, v kolika z nich je alespoň jedna dobrá jednotka a žádná špatná. Proto si vytvoříme dvě pole o délce R, jedno pole pro každý typ jednotky. Do těchto polí budeme zaznamenávat počty funkčních, respektive nefunkčních, jednotek, které mají dosah do odpovídající řady.

Poté budeme načítat jednotlivé jednotky. Nejdříve od pozice jednotky odečteme jedničku – naše pole číslují řady od nuly, zatímco zadání od jedna. Následně se podíváme, na které řady tato jednotka dosáhne, a v příslušném poli (podle funkčnosti) pak na odpovídajících indexech zvýšíme počet jednotek s dosahem o 1. To odpovídá intervalu $[r-d, r+d]$, kde r je řada, ve které se jednotka nachází, a d je její dosah. Zde ale narazíme na problém, že jednotka může mít dosah i mimo vlak (pokud je rozdíl nebo součet čísel r a d menší než 0 či vetší roven R). Proto se omezíme na interval $[\max(0, r-d), \min(R-1, r+d)]$.

Na závěr nám stačí projít všechny řady a pro každou se správným počtem dobrých a špatných jednotek zvýšit výsledek o 1.

A jaká je časová složitost tohoto řešení? Vytvoření a závěrečné projití řad trvá lineárně vzhledem k jejich počtu $\mathcal{O}(R)$. Za každou z J jednotek ale můžeme ke každé řadě přičíst jedničku. Takže celková složitost bude $\mathcal{O}(JR)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/37-Z3-1-jr.py`



Zrychlujeme

Tato část popisuje rychlejší řešení, které ale na plný počet bodů nebylo zapotřebí.

Náš algoritmus je pomalý při zpracovávání jednotek, tak ho pojďme zrychlit. Všimněme si, že zatímco počet řad, na který jednotka dosáhne může být velký, tak když půjdeme zleva doprava, zaznamenáme jen dvě změny – jednou se na jednotku připojíme a jednou se z ní odpojíme. Co kdybychom si místo celkového počtu udržovali jen změny a potom z nich dopočítali počty jednotek?

Opět si vyrobme dvě pole, jedno pro dobré jednotky a jedno pro špatné. Ale nyní je udělejme o délce $R+1$ a budou zaznamenávat změny.

Načteme všechny jednotky a pro každou si zaznačme dvě změny. V řadě $\max(0, r-d)$ se poprvé připojíme, zvyšme tedy danou pozici o 1. Na pozici $\min(R-1, r+d)+1$ se od dané jednotky odpojíme, tak snižme danou pozici o 1.

Nyní projděme všechny řady. Na začátku je počet dobrých i špatných jednotek 0, tedy s ním začneme. Pro každou pozici přičtěme k aktuálnímu počtu jednotek i jejich změny (zvlášť pro dobré i špatné jednotky). Potom, pokud počet dobrých jednotek je kladný a špatných nulový, připočtěme k výsledku 1.

Nyní jsme zlepšili zpracovávání jednotek – pro každou jednotku děláme jen dvě spočítání pozic a přičtení. Zkomplikovali jsme procházení řad, ale taky jen konstantním množství aritmetiky. Celkem tedy máme $\mathcal{O}(J+R)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/37-Z3-1-j+r.py`

Pokud vám naše řešení přišla snadná, můžete zkusit vymyslet řešení v čase $O(J)$.

Úlohu připravili: Matúš Púll, Vladimír Sklenár, Dan Skýpala

37-Z3-2 Řízky

Naším úkolem je vypsat 5 nejlepších babiččiných řízků. Můžeme k tomu přistupovat různými způsoby. Jedním přístupem by bylo si řízky setřídit. Když bychom měli řízky seřazené, je pro nás jednoduché vybrat si nejlepších 5 – podíváme se na prvních 5, pokud jsou seřazené sestupně, nebo posledních 5, pokud vzestupně.

Teď ještě vyřešit, jak řízky setřídit, když to není pouze posloupnost čísel. Většina programovacích jazyků vás nechá napsat vlastní komparátor – funkci, která porovnává dva prvky, který potom můžeme předhodit třídění. To pro nás stačí – řízky jsou porovnatelné tak, že porovnáme jejich ohodnocení.

Tím pádem umíme řízky setřídit – to všeobecně umíme v čase $\mathcal{O}(\check{\mathrm{R}} \log \check{\mathrm{R}})$, poté v konstantním čase vezmeme 5 nejlepších řízků, takže celý algoritmus poběží taky v čase $\mathcal{O}(\check{\mathrm{R}} \log \check{\mathrm{R}})$. Pamatujeme si celé pole řízků, což zabírá $\mathcal{O}(\check{\mathrm{R}})$ prostoru.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/37-Z3-2.py`

A nešlo by to lépe?

Ano, šlo! Můžeme si postupně procházet pole řízků a udržovat pět těch dosud nejlepších. Při zpracování každého řízku máme 3 možnosti:

- Ještě nemáme 5 řízků: přidáme nový řízek do naší pětice.
- Už máme 5 řízků a nový řízek není lepší než nejhorší z naší pětice: nic neděláme, protože víme že nový řízek

nikdy nebude mezi 5 nejlepšími řízky, když už teď existuje 5 řízků, které jsou lepší.
- Už máme 5 řízků a nový řízek je lepší než nejhorší z naší pětice: nejhorší řízek v seznamu nahradíme novým.

Takto nám na konci zbyde nejlepší pětice ze všech řízků, což je cílený stav. Tu ještě nakonec seřadíme, jak po nás chce zadání.

Jak tento průchod realizovat? Můžeme si například udržovat pole o velikosti 5 s dosavadní nejlepší pěticí. V každém kroku pak ručně najdeme pozici nejhoršího řízku z pětice a případně provedeme nahrazení za nový řízek.

A jak je tenhle přístup dobrý? Řízků je Ř a s při zpracovávání každého z nich procházíme pole o délce 5 v čase $\mathcal{O}(5)$, což je konstantní. Tím pádem máme časovou složitost $\mathcal{O}(\check{\mathrm{R}})$. Paměťová složitost je $\mathcal{O}(1)$, protože si průběžně pamatujeme pouze 5 nejlepších řízků.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/37-Z3-2-linear.py`

Když 5 řízků nestačí

Předchozí řešení je naprosto dostačující. Co by se ale změnilo, kdyby se úloha místo na 5 nejlepších řízků ptala na K řízků? Pak by mělo časovou složitost $\mathcal{O}(K\check{\mathrm{R}})$, což zdaleka není optimální. Pro K řádově větší než $\log \check{\mathrm{R}}$ je to dokonce horší než naivní řešení s tříděním.

Pomůžeme si standardním zlepšovákem: použijeme stejný algoritmus, jen si v něm budeme efektivněji ukládat data. Všimneme si totiž, že jsme nikde nepoužili, že kolekce K nejlepších řízků je uložená v poli. Stačilo nám, že s ní umíme provádět následující tři operace:

- Vlož prvek.
- Najdi a vrať největší prvek.
- Změň hodnotu daného prvku.

Znáte-li haldy, pak vám je tento výčet nejspíš povědomý. A pokud je neznáte, doporučujeme kouknout do naší kuchařky.[1] Podstatné pro nás je, že halda je datová struktura, která umí všechny tyto operace provádět v čase $\mathcal{O}(\log K)$, což je výrazné zlepšení oproti poli, které sice umí vkládat a měnit v konstantním čase, ale hledání největšího prvku mu trvá lineárně s jeho velikostí.

Pokud tedy v našem řešení nahradíme pole K nejlepších řízků za haldu K nejlepších řízků, zlepšíme časovou složitost na $\mathcal{O}(\check{\mathrm{R}} \log K)$. Paměťová složitost je $\mathcal{O}(K)$.

Na závěr poznamenáme, že existují ještě jiné algoritmy, kterými se umíme dostat až na časovou složitost $\mathcal{O}(\check{\mathrm{R}} + K \log K)$, ale o těch si povíme zase někdy příště.

Úlohu připravili: Daniel Culliver, Matúš Púll

37-Z3-3 Osmisměrka

Osmisměrka je vlastně dvourozměrné pole, kde se snažíme najít každé slovo. Abychom mohli osmisměrku řešit, musíme si uvědomit několik věcí:

- Slova se můžou opakovat vícekrát
- Slova se můžou překrývat
- Slova se vyskytují v řádcích, sloupcích a diagonálách $\Longrightarrow$ spolu 8 směrů
- Výsledek je tajenka s písmeny, kterými neprochází žádné slovo

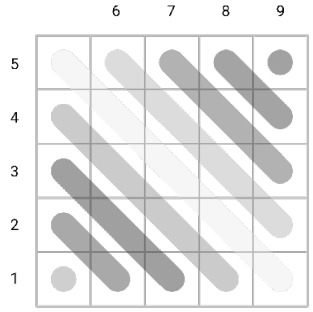
Zamysleme se nad tím, jak bychom si mohli pamatovat, zda slovo prochází nějakou souřadnicí. Můžeme si vytvořit 2D pole o velikosti $N \times N$, kde N je délka strany osmisměrky. Každá buňka tohoto pole bude obsahovat hodnotu `True` nebo `False`, která bude indikovat, zda jsme na daném indexu našli nějaká písmena ze slova. Toto pole nám umožní snadno zjistit, která písmena osmisměrky nejsou součástí žádného slova, a tvoří tak tajenku. Když jsme teď vyřešili problém s ukládáním prohledaného prostoru, zkusme vymyslet, jak prohledávat osmisměrku.

Základní řešení

Úlohu si můžeme rozdělit na tři části: hledání slova v řádcích, sloupcích a diagonálách. Nejdříve vyřešíme tu jednodušší část, tedy jak najít slovo v řádcích a sloupcích. Slovo se může vyskytovat v řádku zleva doprava nebo zprava doleva, a ve sloupci shora dolů nebo zdola nahoru, takže si prakticky vytvoříme obrácenou verzi pro každé slovo.

Můžeme si vytvořit samostatné dvourozměrné pole pro hledání v řádcích a sloupcích. Pole pro hledání ve sloupcích bude jen otočením pole pro hledání v řádcích o 90 stupňů. Pak pro každé slovo a jeho obrácenou verzi projdeme naše políčka v řádcích a sloupcích a pokud najdeme slovo, zapíšeme si, na kterých konkrétních souřadnicích se nachází jeho písmena.

Teď přichází ta složitější část, diagonály. Diagonál je přesně $2N - 1$, kde N je délka strany osmisměrky. Tohle řešení funguje, ale projít tolik diagonál a spočítat, kde se nachází, je dost náročné, a často se u toho dají udělat chyby (orgové se do toho chytli taky), tak se na to zkusme podívat jinak.



Jednodušší implementace

Diagonály jsou vlastně dlouhé řetězce, jen se neposouváme po jednom indexu, ale po dvou. Tedy když chceme jít doprava dolů, musíme se posunout o $+1$ v řádku a $+1$ ve sloupci, když chceme jít doprava nahoru, musíme se posunout o $+1$ v řádku a -1 ve sloupci atd. Stejný princip funguje i pro čtení po řádcích a sloupcích: čtení shora dolů odpovídá posunu o $+1$ v řádku a 0 ve sloupci.

Takto si umíme zapsat každý z osmi směrů jako dvě čísla a pak se po osmisměrce pohybovat jednoduchým přičítáním. Vytvoříme si tedy pole směrů a každý z osmi směrů do něj uložíme jako onu dvojici čísel.

Teď si implementujeme funkci, která umí zjistit, jestli v osmisměrce na zadané pozici začíná v zadaném směru zadané slovo. Ta funguje jednoduše: postupně iterujeme písmeny ve slově a souběžně s tím se v osmisměrce posouváme zadaným směrem a kontrolujeme, že jsou odpovídající znaky slova a osmisměrky shodné. Pokud narazíme na neshodu, můžeme funkci ukončit dříve. Pokud jsme naopak našli celý výskyt slova v osmisměrce, provedeme ještě druhý průchod,

[1] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`

při kterém nastavíme odpovídající políčka v našem dvourozměrném poli výskytů na `True`. Nyní už jen zbývá tuto funkci zavolat pro každou pozici, každé slovo a každý směr.

Na konci už jen vypíšeme všechna písmena, kterými neprochází žádné slovo podle zadání. Časová složitost je $O(N^2 \cdot M)$, kde N je délka strany osmisměrky a M je počet slov.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/37-Z3-3.py`

Bonus: Efektivnější řešení

Řešení, které jsme dosud popsali, by bylo pomalé i v jednosměrce $1 \times N$ – běželo by v čase $\Theta(M \cdot N)$. Hledání v jednosměrce je přitom ekvivalentní hledání všech výskytů slov v textu. To je známý problém řešitelný v čase $\mathcal{O}(M + N)$, například pomocí algoritmu Aho-Corasickové, o kterém se více dočtete v naší kuchařce.[2] To nám dává naději, že i náš osmisměrný algoritmus půjde zrychlit. Ve zbytku řešení nastíníme, jak na to.

Místo abychom se snažili pochopit algoritmus pro jednosměrky a nějak ho upravit pro osmisměrky, představíme si ho jako black box, kterému dáme seznam slov a nasypeme do něj vstupní text a on pro každý znak textu vrátí, zda jím prochází nějaké ze zadaných slov.[3]

Pomocí tohoto black boxu umíme vyřešit naši osmisměrku. Pro jednoduchost nejprve popíšeme, jak najít všechna slova psaná zleva doprava. Z osmisměrky vyrobíme slepením všech řádků jeden dlouhý text. Mezi jednotlivé řádky vložíme oddělovač, abychom se vyhli situaci, kdy nahlásíme slovo, které ve skutečnosti začíná na konci jednoho řádku a končí na začátku dalšího. Na takto zplacatělou osmisměrku pak můžeme pustit řešení jednosměrky. Nakonec už jen výskyty nahlášené jednosměrkovým algoritmem zaneseme do naší 2D tabulky a máme hotovo. Při tom musíme umět překládat mezi pozicemi v 1D textu a 2D tabulce, ale to umíme s trochou dělení a modulení.

Podobně vyřešíme i sedm zbylých směrů – přečteme všechny řádky/sloupce/diagonály v daném směru a poskládáme je za sebe do jednoho dlouhého textu se separátory mezi jednotlivými řádky/sloupci/diagonálami, pak pustíme jednosměrkový algoritmus a na závěr výskyty zaneseme do 2D tabulky.

V tomto řešení osmkrát pouštíme jednosměrkové řešení na text o délce $\mathcal{O}(N^2)$ písmen (o něco více než N^2, jelikož v něm jsou navíc oddělovače). Celková časová složitost je $\mathcal{O}(N^2+M)$, což – navzdory druhé mocnině u N – je lineární s velikostí vstupu.

Jak jsme zmínili, tajenky byly napsány ručně a budou dostupné po ukončení soutěže. Také vydáme náš generátor jako GitHub link s popisem a jsme ochotní odpovídat na dotazy, jak (ne)funguje. Je tam i menší záznam problémů, na které jsme narazili při testování.

Bonus algoritmus pro generování osmisměrek a kontrolu tajenek GitHub.[4] A taky tajenky pro testování.

Úlohu připravili: Daniel Culliver, Ríša Hladík, Vladimír Sklenár

37-Z3-4 Věšení ozdobiček

Podle zadání chceme mezi háčky pověsit Z ozdob tak, aby nejmenší vzdálenost mezi jakýmikoli dvěma z nich byla co největší. Uvědomme si, že potřebujeme najít největší hodnotu d, pro kterou umíme rozmístit všechny ozdoby na háčcích tak, že vzdálenost mezi každými dvěma ozdobami je alespoň d. Jinými slovy, hledáme „největší minimální vzdálenost“ mezi ozdobami.

Jak ověřit, jestli je dané d validní?

Zkusme nejdříve zjistit, jak bychom mohli ověřit, zda je dané d validní. Pak se zamyslíme, jak ho budeme hledat. Nejprve umístíme první ozdobu na první háček. Postupně projdeme další háčky a pokaždé, když najdeme háček vzdálený alespoň d od posledního, na který jsme ozdobu pověsili, tak pověsíme na něj další ozdobu. Pokud takto pověsíme všech Z ozdob, d je validní. Pokud ozdoby dojdou dříve, než pověsíme všech Z ozdob, tak d je příliš velké, a tedy není platné, protože jsme překročili limit $L = \sum h_i$ (součet vzdáleností přes všechny háčky).

Jak najít největší validní d?

Nyní se zamysleme, jak najít největší validní d. Můžeme si všimnout, že pokud d je validní, pak všechny $d' < d$ jsou také validní. Naopak pokud d není validní, pak víme, že každé větší d'' také není validní $d'' > d$.

Tohle se dá využít při použití binárního vyhledávání.[5] Vytvoříme si dvě proměnné pro hodnotu 0 a L, které budou reprezentovat interval možných hodnot d. Potom v každém kroku spočítáme střed intervalu a zjistíme, zda je validní. Pokud ano, posuneme dolní mez, jinak horní. Tímto postupem budeme interval zmenšovat, dokud nebude obsahovat jen jednu hodnotu, která bude největším validním d.

Časová a paměťová složitost

Na naše zjištění, jestli je d validní, potřebujeme projít všechny háčky, kterých je dohromady H. To umíme udělat v čase $\mathcal{O}(H)$. Binární vyhledávání provádíme v $\log L$ krocích, takže celková složitost bude $\mathcal{O}(H \log L)$. Pamatujeme si jen vzdálenosti ozdob a pár konstant, takže paměťová složitost je $\mathcal{O}(H)$.

Úlohu připravili: Kačka Doubková, Vladimír Sklenár

[2] `http://ksp.mff.cuni.cz/viz/kucharky/hledani-v-textu`
[3] Standardní algoritmus Aho-Corasickové pouze vypisuje pozice kde nějaké slovo začíná, ale převést ho na náš formát je implementační detail.
[4] `https://github.com/Faun78/word_search_puzzle_gen`
[5] `http://ksp.mff.cuni.cz/viz/kucharky/binarni-vyhledavani`

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/*