

### Řešení druhé série začátečnické kategorie 38. ročníku KSP

#### 38-Z2-1 Kontrola vagónů

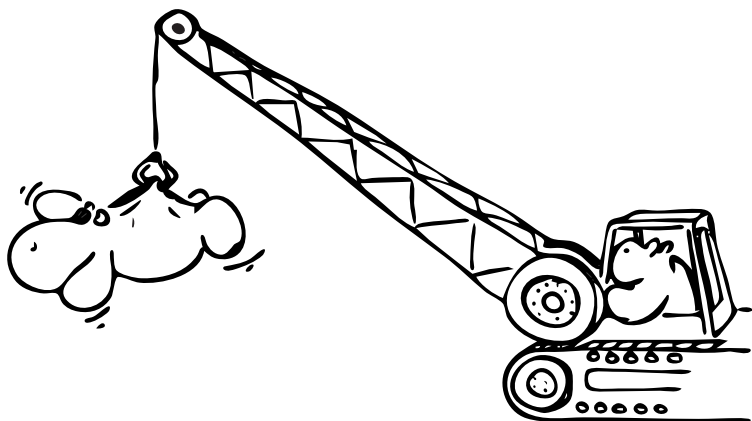
Zadání po nás chce na základě seznamu hlášení od počítadel určit vagóny, ve kterých zbyli na konci cesty nějací cestující a případně ohlásit chybu, pokud nějaké počítadlo tvrdí, že někdo vystoupil z prázdného vagónu. Počítadla říkají jen to, že někdo vystoupil nebo nastoupil konkrétními dveřmi, a tak se nabízí situaci odsimulovat.

Hlášení od počítadel obsahují čísla dveří, ale úloha se v odpovědi ptá na čísla vagónů, vyplatí se nám tedy čísla dveří převést na čísla vagónů a dál pracovat jen s nimi. Víme, že každý vagón má  $D$  dveří a že se dveře i vagóny číslují od jedné, a potřebujeme vědět, ke kolikátému vagónu patří dané dveře. K tomu nám stačí číslo dveří vydělit  $D$  a následně zaokrouhlit k nejbližšímu ne nižšímu celému číslu (spočítat jeho horní celou část).

Poté budeme procházet seznam hlášení na vstupu a pro každý vagón si budeme pamatovat počet cestujících. Pro každé hlášení převedeme číslo dveří na číslo vagónu a zvýšíme nebo snížíme počet cestujících v příslušném vagónu. Pak ještě zkontrolujeme, zda počet cestujících v nějakém vagónu neklesl pod nulu (a vypíšeme chybu a ukončíme program, pokud se tak stalo). Na konci si vyrobíme pole pro neprázdné vagóny, projdeme pole počtů cestujících ve vagónech a do pole neprázdných vagónů přidáme všechny, jejichž počet cestujících není nula. Pokud pole na konci není prázdné, setřídíme ho a vypíšeme na výstup, jinak vypíšeme PRAZDNY.

V nejhorším případě (nedošlo k žádné chybě počítadel) projdeme jednou celý vstup, to trvá  $\mathcal{O}(N)$ , a následně jednou celý seznam počtů cestujících – to trvá  $\mathcal{O}(V)$ . Pokud pole neprázdných vagónů není prázdné, musíme ho setřídít; pole může mít až  $V$  prvků a setřídít ho bude trvat  $\mathcal{O}(V \log V)$ . Výsledná složitost celého algoritmu je tedy  $\mathcal{O}(N + V \log V)$ .

Úlohu připravili: Honza Černohorský,  
Vojta Káně, David „Dejwut“ Pacák



#### 38-Z2-2 Dýně

Zadání se ptá na počet dýní, které vyrostou a na vstupu jsme dostali jejich počáteční počet, takže toto číslo můžeme rovnou přičíst k celkovému počtu dýní. Dále víme, že první a poslední dýně mají vedle sebe místo, kde je žádná dýně neblokuje, takže můžeme přidat dalších  $2 \cdot T$  dýní (po  $T$  na každou stranu).

Potom budeme chtít postupně projít dýni po dýni a zjistit, kolik dýní vyrostou mezi nimi. Nejdřív seřadíme dýně podle jejich pozic. Potom pro každou dvojici sousedních dýní zjistíme, kolik je mezi nimi volného místa. Pokud je mezi nimi  $2 \cdot T$  nebo více míst, tak tam za čas  $T$  vyrostou další dýně na každé straně, tedy  $2 \cdot T$  dýní. Pokud je ale mezi nimi méně místa, tak tam vyrostou jen tolik dýní, jaká byla vzdálenost mezi dýněmi na začátku.

Nakonec jen stačí sečíst všechny dýně a máme výsledek. Celý algoritmus proběhne v čase  $\mathcal{O}(N \log N)$  kvůli třídění dýní na začátku.

Úlohu připravili: Vojta Káně,  
Vojta Lančarič, Moř Rozínková

#### 38-Z2-3 Písmenková polévka

Zadání se nás ptá na nalezení všech jednoslovných *přesmyček* zadaného slova o  $P$  písmenech ve slovníku  $N$  slov. Stačí tedy pro každé slovo ze slovníku ověřit, zda jde o přesmyčku hledaného slova. Jak však ověříme, zda jsou dvě slova přesmyčky? Na to můžeme jít dvěma způsoby.

##### Přesmyčky jako stejné počty písmen

Jak nám již napovídá zadání úlohy, dvě přesmyčky používají právě stejná písmena. Musí tedy platit, že počty výskytů všech písmen anglické abecedy se v obou slovech rovnají. To vede na algoritmus, kde každé slovo převedeme na jeho *charakteristiku* – reprezentaci počtů výskytů písmen. Pak jen zkontrolujeme rovnost charakteristik obou slov.

Zbývá nám vyřešit, jak reprezentovat a porovnávat charakteristiku slova. Protože je malých písmen anglické abecedy 26, stačí si pořádit pole o 26 prvcích, kde  $i$ -tý prvek odpovídá  $i$ -tému písmenu abecedy a jeho hodnota je počet výskytů tohoto písmene ve slově. Pro jeho sestavení stačí projít slovo znak po znaku a na relevantních indexech přičíst jedničku; porovnání charakteristik pak provedeme prvek po prvu.

Jak bude takový algoritmus rychlý? Slovo převedeme na jeho charakteristiku v čase  $\mathcal{O}(P)$ , jejich porovnání zvládneme v  $\mathcal{O}(26) = \mathcal{O}(1)$ .

Jen podotkneme, že kdybychom místo anglické abecedy měli třeba slova složená z čínských piktogramů, můžou být velikosti polí neprakticky velké. Pak se může vyplatit použít datovou strukturu slovník.

<sup>1</sup> <https://www.czechency.org/slovník/ANAGRAM>

## Přesmyčky jako permutace

Podíváme-li se do českého slovníku,<sup>1</sup> zjistíme, že přesmyčky jsou vzájemné permutace písmen ve slově. Stačí tedy ověřit, zda jedno slovo je permutací jiného.

To vede na pokus prostě zkusit permutovat jedno slovo, dokud nedostaneme slovo druhé. Jenže permutací je  $P!$ , což je opravdu hodně rychle rostoucí funkce. Musíme tedy postupovat jinak.

Chtěli bychom slovo převést na jinou, *reprezentující* permutaci, která je stejná pro všechny přesmyčky jednoho slova a navíc ji umíme rychle najít. Pak stačí porovnat, zda jsou reprezentující permutace obou slov totožné. Jak si však takovou permutaci porovnat?

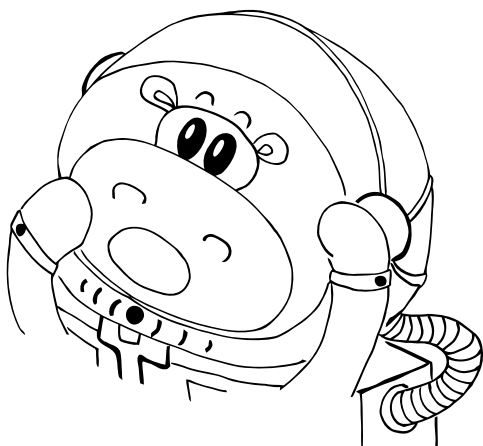
Zde nám pomůže *třídění*. To přeuspořádává prvky v seznamu tak, aby byly v rostoucím pořadí – jde tedy o permutaci, která je navíc jednoznačná. To je přesně to, co hledáme! Abychom tedy poznali, že jsou dvě slova přesmyčky, podíváme se na ně jako seznamy písmen, ty setřídíme a porovnáme výsledky.

Tentokrát nám ověření bude s běžným třídícím algoritmem trvat  $\mathcal{O}(P \log P)$  za setřídění písmen v obou slovech a  $\mathcal{O}(P)$  za jejich porovnání.

Všimněme si, že oba přístupy ve výsledku pracují velmi podobně – převedou slova na objekty, které se pro přesmyčky rovnají, a pak ověří jejich rovnost. Tuto podobnost můžeme ještě přiblížit, pokud budeme písmena třídit *počítáním*, neboli *count sortem*. Tím se dostaneme na lepší složitost  $\mathcal{O}(P)$ .

V obou případech se na konci dostaneme na algoritmus, který trvá čas  $\mathcal{O}(NP)$ . Toto je navíc nejlepší možná složitost, jelikož tolik času stejně potřebujeme na přečtení celého vstupu znak po znaku.

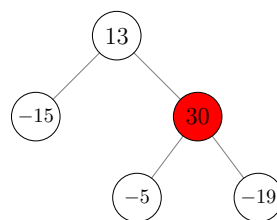
Úlohu připravili: Katia „Čiči“  
Kočíková, Vojta Lančarič



## 38-Z2-4 Těžká raketa

V zadání úlohy jsme dostali binární strom, který představuje raketu, kde každý vrchol našeho stromu má určitou váhu. Některé vrcholy (motory) mají zápornou váhu (tah), jiné (palivo) mají kladnou váhu. Naším cílem je odříznout některé části rakety (podstromy) tak, aby celková váha zbylé části byla záporná, což umožní raketě vzlétnout. My vyřešíme o něco obecnější úlohu a budeme chtít, aby celková váha byla nejmenší možná.

Představme si, že se rozhodujeme u konkrétního vrcholu, zda jeho podstrom odříznout, nebo ne. Předpokládejme, že pro tento podstrom už jsme naši úlohu vyřešili – tedy umíme ho ořezat tak, aby jeho váha byla co nejmenší možná. Pokud má celý tento podstrom (včetně všech jeho neodříznutých potomků) i po ořezání kladnou váhu, je pro nás zátěž. Jeho odříznutím se zbavíme kladné váhy, což nám pomůže. Pokud má naopak zápornou váhu, znamená to, že tento podstrom „táhne nahoru“ a snižuje celkovou váhu, a proto si ho chceme nechat.



Jak to ale celé zrealizujeme? Můžeme postupovat zdola nahoru, neboli od listů ke kořeni. Pro každý vrchol si spočítáme, jakou nejmenší váhu může mít jeho podstrom, pokud bychom řezali optimálně. K tomu potřebujeme znát tyto hodnoty pro jeho syny.

Algoritmus můžeme implementovat pomocí průchodu stromem do hloubky (DFS). Začneme v kořeni a rekurzivně sestoupíme až k listům. Při návratu z rekurze se rozhodneme, zda aktuální podstrom odříznout.

Konkrétně pro každý vrchol sečteme jeho vlastní váhu s vahami neodříznutých částí podstromů jeho synů. Pokud nám vyjde kladná hodnota, celý tento podstrom odřízneme (a pro nadřazený vrchol se bude tvářit, jako by měl váhu 0). Pokud vyjde záporná, podstrom si necháme a jeho váha přispěje k celkovému součtu.

Když takto dojdeme až ke kořeni, zjistíme celkovou váhu neodříznuté části rakety. Pokud je záporná, úloha má řešení.

Časová složitost bude lineární vzhledem k počtu vrcholů, tedy  $\mathcal{O}(N)$ , protože každý vrchol navštívíme jen jednou.

Úlohu připravil: Vladimír Sklenář