



Vzorové riešenia 1. kola zimnej časti

Eman

1. Kupujeme RAMky

(max. 12 b za popis, 8 b za program)

Riešenie s násobením/delením medzivýsledku za 4 body

Budeme prechádzať postupnosť skratiek prípon zľava doprava. Spočiatku vynásobíme/vydelíme hodnoty prvých 2 prípon. Ich výsledok si zapamätáme a následne ho vynásobíme/vydelíme ďalšou hodnotou prípony. Takto pokračujeme až na koniec postupnosti.

To, či násobíme, alebo delíme, možno nájsť v zadaní. Ak nasledujúca prípona nie je menšia, násobíme, inak delíme.

Na vstupe dostávame len skratky prípon, my potrebujeme tieto skratky prípon premeniť na ich hodnotu. Na to si môžeme vytvoriť slovník, kde kľúč bude skratka prípony a hodnota bude hodnota prípony. Teda, ak potrebujeme poznať hodnotu ľubovoľnej skratky, stačí sa v slovníku pozrieť, aká hodnota je skratke priradená.

Postupnosť skratiek prechádzame raz, teda časová zložitosť je $O(n)$, kde n je počet skratiek na vstupe. Pamäťová zložitosť je tiež $O(n)$, pamätáme si všetky skratky zo vstupu.

Nevýhody tohto riešenia

Prístup s násobedným/delením medzivýsledku prináša nasledujúce komplikácie:

1. V niektorých prípadoch bude algoritmus pracovať s float hodnotami, teda desatinnými číslami. Tie sú uložené v binárnej sústave a existujú také, ktoré sa nedajú zapísať konečným počtom bitov, teda majú nekonečný bitový rozvoj (podobne ako môžu mať desatinné čísla nekonečný desatinný rozvoj). Zároveň máme len konečný počet bitov, ktoré môžeme použiť na reprezentáciu daného desatinného čísla. Teda nekonečný bitový rozvoj nevieme zapísať konečným počtom bitov, z čoho vyplýva, že číslo musíme zaokrúhliť, a teda bude nepresné.
Pri kumulácii takýchto nepresností a následnými operáciami s nepresnými číslami, násobeniami/deleniami, sa bude zaokrúhľovacia chyba zväčšovať, čo nám na konci môže dať nesprávny výsledok.
Všeobecne je dobré sa vyhnúť práci s float hodnotami, ak sa to dá.
2. Môže nastať prípad, keď násobíme veľa veľkých čísel za sebou, to napríklad, keď vstup bude tvorený iba z prípon mega. Potom výsledok nebudeme vedieť reprezentovať, keďže bude obrovský. V Pythone sa zaokrúhli na nekonečno a v C++ nastane tzv. overflow, teda pretečenie.
3. Môže nastať prípad, keď násobíme veľa malých desatinných čísel, to napríklad, keď vstup bude tvorený iba z prípon mikro. Tentokrát bude výsledok hrozne malé číslo, opäť nebude reprezentovateľné. V Python by sa číslo zaokrúhlilo na 0 a v C++ by nastal tzv. underflow, teda podtečenie.

V prvej sade tieto problémy ešte nemusíme riešiť, keďže $n \leq 10$ a zároveň platí, že medzivýsledok je vždy medzi 1 a 10^5 . Teda nerobíme veľa operácií, zaokrúhľovacie chyby sa neprejavujú a výsledok nebude nikdy obrovský, ani príliš malé desatinné číslo. Existuje aj riešenie za 4 body, ktoré vôbec nepoužíva floats, viď odstavec s názvom Integer-only riešenie za 4 body.

Integer-only riešenie za 4 body

V prvom riešení sa nám mohlo stať, že sme násobili alebo delili desatinným číslom, z čoho nám vznikol float. Napr. v Pythone, ak vynásobíme $1000 * 0.1$, dostaneme číslo 100.0, čo je typu float.

Existuje trik, akým sa vieme úplne vyhnúť počítaniu s desatinnými číslami, teda aj ich zaokrúhľovacím chybám.

Začneme s tým, že chceme, aby všetky hodnoty skratiek boli celé čísla. To môžeme spraviť tak, že každú z nich vynásobíme číslom 1 000 000. Teda napríklad prípona mikro, ktorá má hodnotu 0.000 001, jej nová hodnota bude 1.

Ďalej budeme pokračovať ako v predošlom riešení, budeme si držať nejaký medzivýsledok, ktorý budeme násobiť/deliť novými hodnotami skratiek. Toto by nám samozrejme dávalo nesprávne výsledky, keďže každá z hodnôt je 1 000 000-krát väčšia.

Stačí však po každom násobení vydeliť medzivýsledok hodnotou 1 000 000. Takto bude výsledok správny, keďže najprv sme hodnotu skratky vynásobili 1 000 000, ale nakoniec sme medzivýsledok vydělili tou istou hodnotou. Zároveň, sme pri tom nepoužili floaty. Dôležité je medzivýsledok najprv vynásobiť hodnotou skratky, až potom vydeliť 1 000 000. Ak by sme to spravili naopak, hrozí, že medzivýsledok sa zaokrúhli na nulu.

Podobne pre delenie, tu budeme musieť pred každým delením vynásobiť medzivýsledok hodnotou 1 000 000. Znova, medzivýsledok bude správny, opäť je však dôležité poradie operácií, v tomto prípade najprv násobíme.

Toto riešenie funguje iba v prvej sade, keďže v 2. sade môže byť medzivýsledok ľubovoľne veľký. Oproti riešeniu s floatmi však dokáže fungovať aj na príklady s medzivýsledkami až po približne 10^{18} , pokiaľ použijeme 64-bitové čísla (na rozdiel od floatov, ktoré majú iba 23 bitov presnosti). Celé čísla v Pythone majú presnosť efektívne nekonečnú, čím sú však väčšie, tým je práca s nimi pomalšia... A najmä, tento prístup nám nedovoľuje reprezentovať výsledky menšie než 1.

Lepšie riešenie s počítaním exponentu za 8 bodov

V tomto riešení opäť prechádzame postupnosť skratiek zľava doprava, rozdiel je v tom ako si pamätáme medzivýsledok. Čo si môžeme všimnúť je, že hodnoty všetkých skratiek sú mocniny čísla 10. Teda každé z nich vieme zapísať ako 10^x , kde x je nejaké celé číslo. Napr. 100 sa dá zapísať ako 10^2 alebo 0.01 ako 10^{-2} .

Zároveň platí vzťah pre násobenie $10^a * 10^b = 10^{a+b}$, podobne pre delenie $10^a / 10^b = 10^{a-b}$, kde a, b sú celé čísla. Z toho vyplýva, že aj po vynásobení a vydelení hodnôt skratiek sa bude exponent dať zapísať ako 10^x , kde x je celé číslo.

Miesto násobenia a delenia hodnôt si teda budeme pamätať, akú má hodnotu exponent, teda náš medzivýsledok bude hodnota exponentu. Spočiatku to bude exponent prvej skratky. Postupne k nemu pričítavame exponenty ďalších skratiek, to v prípade, ak by sme mali násobiť medzivýsledok hodnotou ďalšej skratky, inak od neho odčítavame exponent, teda keď by sa malo deliť.

Na konci máme exponent, podľa ktorého poznáme aj aký ma byť správny výstup.

Tu sa časová ani pamäťová zložitosť nezmení, keďže stále prechádzame celú postupnosť. Môžeme si rozmyslieť aj to, že sa lepšia časová zložitosť v tejto úlohe ani nedá dosiahnuť, keďže na každú skratku sa určite budeme musieť pozrieť aspoň raz, aby sme úlohu korektne vyriešili.

Pamäťovú zložitosť vieme optimalizovať, viď odstavce s názvom Optimalizácia pamätevej zložitosti.

Výpis výstupu čísla s exponentom

Na konci dostaneme *exponent* čísla, ktoré má byť výstupom úlohy. Teda ak dostaneme *exponent* rovný 3, tak výstup má byť 10^3 . Tento *exponent* by mohol byť veľmi veľký, preto nechceme vypísať hodnotu vo forme `print(10**exponent)`, keďže by sa najprv vyhodnotil výraz `10**exponent`, ten by však mohol byť taký veľký, že sa zaokrúhli na nekonečno a to sa vypíše. Alebo naopak taký malý, že sa zaokrúhli na 0 a to sa vypíše.

Miesto toho vieme vypísať výstup nasledovne.

Ak je exponent kladný, potom číslo začína s 1 a pokračuje s počtom núl, aký veľký je exponent. Teda nevyhodnocujeme výraz, len vypíšeme číslicu 1 a za to počet núl, aký veľký je exponent.

Podobne, ak je exponent záporný, číslo je desatinné, začína s nulou a desatinnou čiarkou, pokračuje s počtom núl, aký veľký je exponent - 1 a je zakončený číslicou 1. Opäť miesto vyhodnotenia výrazu vypisujeme ako "vyzerá" to desatinné číslo.

Optimalizácia pamätevej zložitosti

Môžeme si všimnúť aj to, že postupnosť skratiek nemusíme mať celú uloženú v pamäti. To preto, lebo porovnávame vždy 2 skratky vedľa seba, stačí si pamätať v každom momente len tie 2. Pamäťovú zložitosť teda vieme optimalizovať na konštantnú.

Slovník má tiež konštantnú pamäť, keďže obsahuje konštantný počet prípon zo zadania.

Zároveň v tomto prípade musíme prispôbiť aj vypisovanie výstupu. V predošlej ukážke ho vypisujeme s využitím lineárne veľkej pamätevej zložitosti. Pri zavolaní napr. `print("1" + exp * "0")` sa najprv vyhodnotí výraz vo vnútri funkcie. Tento výraz môže byť však lineárne veľký vzhľadom k n , teda $O(n)$ pamäti.

Vypisovanie výstupu vieme tiež optimalizovať na konštantnú pamätevej zložitosti. Podobne ako sme načítavali vstup po jednom znaku, tu budeme vypisovať výstup po jednom znaku.

2. Upratovanie

(max. 12 b za popis, 8 b za program)

Bruteforce

Najjednoduchšie je nasimulovať si jednotlivé zmeny. Pre každú zmenu nájdeme všetky výskyty daného písmena a postupne ich nahradíme daným reťazcom. Takto pre každú zmenu musíme prejsť celý reťazec v čase $O(n)$, čím dostaneme časovú zložitosť $O(n \cdot m)$. Netreba však zabúdať na to, že dĺžka reťazca sa pri nahrádzaní postupne zväčšuje. Skutočná časová zložitosť by tak rástla a program by bol pre väčšie vstupy extrémne pomalý.

Zaujímavá myšlienka

Môžeme si uvedomiť, že presný tvar výsledného reťazca nás vôbec nezaujíma. Keďže sa pýtame len na konečný počet nejakého písmena, stačí si po celý čas pamätať len to, koľkokrát sa jednotlivé písmená v reťazci momentálne nachádzajú. Ak máme napríklad zmeniť písmeno **a** za reťazec **bcd**, stačí sa pozrieť, koľko **a**-čok momentálne máme. Ak ich je päť, vieme, že do reťazca pribudne päť **b**-čok, päť **c**-čok a päť **d**-čok. Zároveň nesmieme zabudnúť, že týchto päť **a**-čok z reťazca zmizlo, takže ich počet musíme vynulovať (respektíve zobrať do úvahy aj to, či sa písmeno **a** nenachádza aj v nahrádzajúcom reťazci).

Optimálne riešenie

Keďže možných znakov v reťazci je konštantne veľa (26), vytvoríme si pole alebo slovník, kde si pre každé písmenko budeme pamätať jeho aktuálny počet. Na začiatku si raz prejdeme vstupný reťazec a spočítame začiatkové výskyty, čo nám zaberie čas $O(n)$. Následne spracujeme m zmien. Pre každú zmenu zistíme aktuálny počet nahrádzaného písmena. Tento počet pripočítame každému písmenu, ktoré sa nachádza v nahrádzajúcom reťazci (tento reťazec musíme raz prejsť), a nakoniec aktuálny počet nahrádzaného písmena vynulujeme. Ak má nahrádzajúci reťazec maximálnu dĺžku k , potom spracovať jednu zmenu vieme v čase $O(k)$. Zmien je m , teda časová zložitosť spracovania zmien je $O(k \cdot m)$. Výsledná časová zložitosť tohto riešenia je $O(n + k \cdot m)$. Keďže si pamätáme len tabuľku výskytov pre 26 písmen, pamäťová zložitosť je konštantná, teda $O(1)$.

Filip

3. Predajne všade

(max. 12 b za popis, 8 b za program)

Cieľom bolo nájsť (resp. spočítať) všetky lokálne minimá (políčka, ktorých čísla sú ostro menšie ako čísla všetkých ich susedov) v mriežke (2d pole)

1. Subtask

Špeciálna vlastnosť prvej sady robí z 2d úlohy 1d úlohu. Stačí nám teda pozrieť sa od začiatkového políčka mriežky naľavo a napravo, a posúvať sa dokým sú čísla v ďalších políčkach menšie.

Prechádzame cez všetky políčka, takže časová zložitosť je $O(r \cdot s)$. Všetky si ich musíme aj pamätať, takže aj pamäťová zložitosť je $O(r \cdot s)$

Toto riešenie však prejde iba na prvej sade.

2. Subtask

Špeciálna vlastnosť druhej sady nám dovoľuje problém zredukovať na hľadanie lokálnych miním.

Prechádzame cez všetky políčka, takže časová zložitosť je $O(r \cdot s)$. Všetky si ich musíme aj pamätať, takže aj pamäťová zložitosť je $O(r \cdot s)$

Toto riešenie však prejde iba na druhej sade.

Úplné riešenie

Úplné riešenie tejto úlohy spočíva v tom, že si mriežku interpretujeme ako graf, kde každé políčko je vrchol a každá strana medzi políčkami je hrana. Na takomto grafe následne spustíme implementáciu ľubovlného prehľadávacieho algoritmu (napr. BFS alebo DFS), s tým že budeme navštevovať iba vrcholy, ktorých výška je menšia ako výška vrchola, z ktorého donich ideme. Vrchol z ktorého na základe takejto podmienky nemôžeme nikam ísť je lokálne minimum.

Časová zložitosť prehľadávacích algoritmov je vo všeobecnosti $O(n + m)$, kde n je počet vrcholov grafu a m je počet hrán. Tým že náš graf vznikol z mriežky, počet hrán je lineárne závislý od počtu vrcholov. Počet vrcholov je počet políčok v mriežke, teda $r \cdot s$. Časová zložitosť tohto riešenia je teda $O(r \cdot s)$.

Pamäťová zložitosť je tiež $O(r \cdot s)$, lebo si všetky políčka v mriežke musíme pamätať.

4. Upravovanie nevyšlo

(max. 12 b za popis, 8 b za program)

Cieľom úlohy bolo zistiť, koľko najmenej krokov potrebujeme na to, aby sme pole náhodne rozhádzaných čísel od 1 po n utriedili, s tým, že jediná operácia, ktorú môžeme urobiť, je, že zoberieme ktorékoľvek číslo z pola a dáme ho buď na začiatok, alebo na koniec.

Bruteforce

Ako bruteforce riešenie by sme v tomto prípade postupovali nasledovne:

Každý možný stav pola, teda každú permutáciu čísel od 1 po n , si predstavíme ako vrchol grafu. Ďalej každú podľa zadania platnú operáciu, ktorá zmení stav pola, si predstavíme ako hranu tohto grafu.

Takýto graf vieme z počiatočného stavu začať **prehľadávať do šírky (BFS)**¹ a pre každý vrchol si pamätať, koľko operácií sme potrebovali na to, aby sme sa naň dostali. V momente, keď sa dostaneme do vrcholu, ktorý reprezentuje utriedené pole, vieme, aký počet operácií sme potrebovali.

Existuje $n!$ permutácií, teda $n!$ vrcholov grafu, a z každej permutácie je $2n$ platných operácií, teda z každého vrcholu vedie $2n$ hrán. Všetkých hrán je teda dokopy $2n \cdot n!$. Časová zložitosť tohto riešenia je teda $O(2n \cdot n! + n!)$, respektíve $O(n \cdot n!)$.

Pamätať si treba všetky vrcholy (ktorých je $n!$) a všetky hrany (ktorých je $2n \cdot n!$). Pamäťová zložitosť je teda $O(2n \cdot n! + n!)$, respektíve $O(n \cdot n!)$.

Zaujímavé je uvedomiť si, že teoreticky tento graf vôbec nemusíme konštruovať (čím ušetríme značné množstvo pamäte), pretože na základe platných operácií vieme susednú permutáciu vždy vypočítať. Problém je však v tom, že susedných permutácií je vždy lineárne veľa, a vytvoriť každú z nich trvá lineárne dlho, teda sa nám rozpísaná forma časovej zložitosti zhorší z $O(2n \cdot n! + n!)$ na $O(2n^2 \cdot n! + n!)$.

Zaujímavá myšlienka

Dôležité v tejto úlohe bolo uvedomiť si, že sa dá zmeniť na inú, jednoduchšiu úlohu. Konkrétne ide o hľadanie najdlhšej postupnosti čísel stúpajúcich o 1, ktoré sú navzájom v správnom poradí, ale nie nutne hneď po sebe (v pôvodnom poli sa medzi nimi môžu vyskytovať aj iné čísla, ktoré nie sú súčasťou postupnosti).

Potom totiž už môžeme tieto čísla považovať za utriedené, a teda už treba len vyriešiť, ako utriedime tie ostatné okolo nich, pričom zjavne musí existovať nejaké optimálne poradie operácií, pri ktorom stačí každé z týchto ostatných čísel presunúť raz.

Navyše pre tie ostatné čísla platí, že na základe ich samých a ich polohy vzhľadom na nájdenú postupnosť je jednoznačne určené v akom poradí a na ktorú stranu pola ich treba presunúť.

My však nepotrebujeme vedieť, aké operácie treba vykonať, ale iba koľko ich treba vykonať. Keďže je tento počet rovný počtu čísel mimo postupnosti, dá sa s konštantnou časovou zložitosťou vypočítať, ak máme nájdenú postupnosť s vyššie opísanými vlastnosťami (a teda aj jej dĺžku).

Naivná implementácia

Najjednoduchší spôsob, ako nájsť takúto postupnosť, je vyskúšať všetky možné súvislé aj nesúvislé postupnosti. To sa dá reinterpretovať ako skúšanie každej možnej podmnožiny.

Nájsť všetky podmnožiny sa dá pomocou jednoduchej rekurzcie. Potom iba pri každej skontrolujeme, či čísla v nej rastú postupne (rozdiel medzi susednými by mal byť 1), a spomedzi takých vyberieme tú najväčšiu.

Podmnožín je 2^n , každá z nich najviac n veľká (a musíme ich všetky kvôli kontrole podmienky prejsť), takže časová zložitosť tohto riešenia bude $O(n \cdot 2^n)$, respektíve $O(2^n)$. Všetky podmnožiny musíme mať aj uložené v pamäti, takže pamäťová zložitosť bude rovnaká.

Niečo lepšie

Ak chceme vymyslieť efektívnejšie riešenie, musíme k úlohe pristupovať trochu inak. Spomeňme si, že hľadáme nie nutne súvislú, o 1 rastúcu, postupnosť čísel, teda postupnosť tvaru

$$l, l+1, \dots, r$$

kde l je najmenšie číslo v postupnosti a r je najväčšie. Teda l môže byť akékoľvek číslo od 1 do n a r môže byť akékoľvek číslo od $l+1$ do n . (Nemá zmysel riešiť situáciu $r = l$, pretože taká môže nastať pri akomkoľvek l , a teda môžeme proste počítat s tým, že akákoľvek platná postupnosť bude mať dĺžku aspoň 1.)

To znamená, že existuje iba kvadratické množstvo postupností, ktoré treba skontrolovať. (V každej pozícii v poli sa môže postupnosť začínať a môže mať maximálne n prvkov.) Keď pri každej z nich nekontrolujeme, či pokračuje niekde ďalej v poli, ale či sa jej pokračovanie nenachádza niekde ďalej v poli (čo sa dá zisťovať pažravo,

¹<https://school.ksp.sk/courses/graphs/graphs1-bfs>

lebo ak sme také číslo ešte nevideli, znamená to, že je ďalej v poli), vieme všetky postupnosti skontrolovať v časovej zložitosti $O(n^2)$.

Pamätať si potrebujeme iba pôvodné pole a pozície jednotlivých čísel v ňom, takže pamäťová zložitosť je $O(n)$.

Optimálne riešenie

Najlepšie riešenie dosiahneme použitím [dynamického programovania](#)². Opäť si budeme pamätať pozície jednotlivých čísel, no tentoraz si ich nepotrebujeme predpočítavať, ale stačí si ich zapisovať za chodu.

Pre každé číslo v poli sa spýtame, či už vieme, kde sa nachádza o jedno menšie od neho (okrem jednotky, pre tú to logicky nevieme za žiadnych okolností).

Ak nie, znamená to, že je niekde za aktuálnou pozíciou v poli, a teda pre aktuálne číslo začíname novú postupnosť a zapíšeme si preň dĺžku postupnosti 1.

Ak áno, znamená to, že pokračujeme v postupnosti toho o jedno menšieho čísla, a teda si pre aktuálne číslo zapíšeme dĺžku postupnosti o 1 väčšiu, ako máme zapísanú pre to o jedno menšie číslo.

Popri tom si pamätáme, akú najdlhšiu postupnosť sme videli.

Takýmto spôsobom nám stačí pole prejsť iba raz, a teda časová zložitosť takéhoto riešenia je $O(n)$.

Pamätať si musíme pôvodné pole, pozície jednotlivých čísel v ňom, a dĺžku postupnosti pre každú pozíciu v poli, teda pamäťová zložitosť je tiež $O(n)$.

Stanko

5. Jednoduché plavidlá

(max. 12 b za popis, 8 b za program)

Objem jedného plavidla je xyh . Toto číslo musí byť v intervale $[a, b]$. Takže pre každú dosku si vieme vypočítať možný interval dĺžky hranolov. Pre každé plavidlo teda platí $\lceil \frac{a}{xy} \rceil \leq h \leq \lfloor \frac{b}{xy} \rfloor$. Takže naša úloha sa transformuje na to, že máme na vstupe n čísel a n intervalov. Chceme medzi nimi vytvoriť čo najviac párov pre ktoré platí, že dané číslo je v intervale. Nech intervaly sú $[L_i, R_i]$ a čísla h_i .

Bruteforce

Ku každému číslu h_i chceme nájsť interval $[L_j, R_j]$. Čísla si utriedime a budeme ich párovať v poradí od najmenšieho. Pre každé číslo si potrebujeme nájsť interval do ktorého pasuje. Pre dané číslo chceme nájsť ten najlepší interval, teda taký do ktorého pasuje a zároveň hodnota R_j je čo najmenšia. Potom sa nám nestane, že sme nejaký už spárovali a potom nám niečo nebude sedieť. Inými slovami ak existuje nejaké optimálne riešenie, tak existuje rovnako dobré, ktoré páruje vždy tie najbližšie intervaly. Naopak to platiť nemusí. Pre každé číslo si nájdeme takýto najbližší nevyužitý interval a spárujeme ich spolu ak sa našiel. Inak ideme ďalej.

Časová zložitosť tohoto bude $O(n^2)$, lebo pre každé číslo budeme hľadať najbližší nepoužitý interval, teda všetkými intervalmi prechádzať pri každom čísle. Pamäť bude $O(n)$, lebo si pamätáme iba vstup.

Optimálne riešenie

Skúsme sa bližšie pozrieť na hľadanie toho správneho intervalu. Potrebujeme dátovú štruktúru, ktorá nám vie rýchlo povedať ten najbližší interval a odstrániť ho. Túto operáciu vieme realizovať napríklad [haldou](#)³. Budeme mať teda haldy, v ktorej budeme mať aktuálnych kandidátov na intervaly. Táto halda bude minimová, zoradená podľa hodnôt R_j . Keď nám teda príde číslo h_i , vieme ho v logaritmickej čase odstrániť z haldy a spárovať ho s tým najoptimálnejším intervalom.

Musíme ešte zariadiť, aby v halde boli len dobrí kandidáti, teda takí, ktorí sa ešte budú dať spárovať s nejakým číslom. Keďže čísla postupne rastú vieme všetky intervaly, kde $h_i \leq L_j$ z haldy odstraňovať. Už sa k nim nemá ako nájsť pár. Intervaly kde $L_j \leq h_i$ do haldy zase musíme pridať, lebo tie sa teraz, alebo niekedy v budúcnosti budú dať spárovať.

Pre rýchle hľadanie intervalov na pridanie do haldy si ich na začiatku utriedime podľa R_j a potom si pamätáme index, že pokiaľ sme tie intervaly do haldy pridali. Samotné pridávanie bude znamenať posúvanie tohoto indexu a pridanie intervalu do haldy.

Každý interval sa práve raz pridá do haldy a práve raz z nej odoberie. Časová zložitosť tohoto riešenia teda bude $O(n \cdot \log(n))$. Pamäťová zložitosť je $O(n)$, keďže si pamätáme len haldy a samotný vstup - obe lineárnej veľkosti.

²<https://school.ksp.sk/courses/techniques-2/dynamic-programming>

³<https://school.ksp.sk/courses/data-structures/heap>

6. Extrémna nákupná mánia

(max. 12 b za popis, 8 b za program)

Ako sa neprepočítat

Aby sme každú kliku započítali len raz, tak kliky budeme počítat podľa najmenších vrcholov, ktoré sú v nich. Týmto vrcholom budeme hovoriť koreň kliky. Toto vieme jednoducho docieľiť tak, že si zorientujeme každú hranu, aby išla z menšieho vrchola do väčšieho. Keď sa v tomto novom grafe pozeráme na susedov nejakého vrchola u , tak dostaneme len tie vrcholy, ktoré môžu byť v klikе s koreňom u .

Teraz môžeme využiť to, že každý vrchol má malý stupeň. Pre každý vrchol u spočítame, koľko existuje klík s daným vrcholom ako koreňom. Na to si vygenerujeme všetky podmnožiny susedov vrchola u s veľkosťou $k - 1$ (v zorientovanom grafe) a skontrolujeme, či táto k -tica tvorí kliku.

Označme si d ako maximálny stupeň vrchola v grafe. Pre každý vrchol musíme skontrolovať najviac $\binom{d}{k-1}$ kandidátov na kliku. V závislosti od implementácie jedna kontrola zaberie medzi $\mathcal{O}(kd)$ a $\mathcal{O}(k)$ času. Celková časová zložitosť je teda pri šikovej implementácii kontrolovania klíku $\mathcal{O}(n \binom{d}{k-1} k)$. Pamäťová zložitosť je $\mathcal{O}(n + m)$. Nižšie prezentujeme riešenie so zložitosťou na kontrolu $\mathcal{O}(kd)$, k lepšej časovej zložitosti sa dá dostať použitím podobných myšlienok ako v ďalšom riešení.

Riešenie s bitsetmi

Doteraz sme prišli na to, že počet klík veľkosti k zakorenených v u je rovnaký, ako počet klík veľkosti $k - 1$ medzi susedmi u . Túto myšlienku ale môžeme ďalej zopakovať. Vyberieme si nejakého suseda v vrchola u ako koreň kliky veľkosti $k - 1$ a v prieniku susedov vrcholov u a v hľadáme kliku veľkosti $k - 2$. Keď sa dostaneme do momentu, že hľadáme kliky veľkosti 1, tak je odpoveď jednoduchá, stačí vrátiť počet vrcholov v prieniku. Naopak ak nám zostáva príliš málo kandidátov na dokončenie kliky, nemusíme ďalej pokračovať. Keďže každá klika má jednoznačne určený najmenší vrchol a susedov vyberáme v zorientovanom grafe, tak každú kliku navštívime práve raz.

Aby sme tento postup vedeli efektívne implementovať, tak si potrebujeme udržiavať množinu všetkých prípustných vrcholov a vedieť rýchlo robiť jej prienik so susedmi nejakých vrcholov. Na to využijeme techniku tzv. *bitmasiek*, teda fakt, že čísla v počítači sú reprezentované binárne. Množinu teda vieme reprezentovať ako nejaké binárne číslo a prienik dvoch množín je len binárny **and** týchto množín.

Budeme si ale musieť prečíslovať vrcholy, aby reprezentácia množiny mala málo bitov. Keďže pri hľadaní kliky zakorenenej vo vrchole u sa pozeráme len na susedov vrchola u , tak si vrcholy prečísľujeme podľa poradia, v akom sú v zozname susedov vrchola u . Teda pre každý koreň u a suseda v si spočítame masku, ktorá hovorí, ktorých susedov majú u a v spoločných. Toto predpočítanie vieme spraviť napríklad v čase $\mathcal{O}(nd^3)$.

Analýza časovej zložitosti

V našom rekurzívnom algoritme pri prehľadávaní z vrchola u postupne pridávame vrcholy kliky a “zužujeme” ďalších kandidátov na prienik so susedmi už pridaných vrcholov. Keďže sa pozeráme na susedov v zorientovanom grafe, tak ich pridávame v rastúcom poradí. To znamená, že každú kliku zakorenenú v u navštívime práve raz.

Pre každú kliku veľkosti j zakorenenú v u teda máme jeden stav prehľadávania. Týchto stavov je pre dané u najviac $\binom{d}{j-1}$. Keďže pre jednu kliku robíme len $\mathcal{O}(1)$ práce (vďaka bitmaskám), tak celková časová zložitosť bude počet klík, ktoré navštívime, teda počet stavov. Ako je popísané vyššie, stačia nám stavy len pre $j \in \{1, \dots, k - 1\}$. Celková časová zložitosť prehľadávania je teda $\mathcal{O}(n \sum_{j=0}^{k-2} \binom{d}{j})$.

Výsledná časová zložitosť je teda $\mathcal{O}(nd^3 + n \sum_{j=0}^{k-2} \binom{d}{j})$. Ak $k - 1 \leq d/2$, tak binomické koeficienty budú rásť s j , teda ich môžeme ohraničiť najväčším koeficientom. Potom dostaneme zložitosť $\mathcal{O}(nd^3 + n \binom{d}{k-2} k)$. Pamäťová zložitosť je $\mathcal{O}(n + m + nd^2)$. Časová zložitosť je síce podobná ako pri šikovej implementácii predošlého riešenia (kde kontrolu robíme pomocou bitmasiek), ale v praxi bude toto riešenie oveľa rýchlejšie. Po prvé, pomer $\binom{d}{k-1}$ a $\binom{d}{k-2}$ pre malé k je približne d/k , teda rekurzívne riešenie je v tomto prípade asymptoticky d/k -krát rýchlejšie. Prípad $k - 1 = d/2$ je dokonca ten “najzložitejší”, lebo v tomto prípade môže graf obsahovať najviac klík veľkosti k . Vtedy sú ale obe kombinačné čísla približne rovnaké, teda asymptotická časová zložitosť oboch riešení je rovnaká. Po druhé, rekurzívne riešenie navštívi len toľko klík, koľko naozaj musí. Naopak, naše prvé riešenie vyskúša všetky podmnožiny susedov daného vrchola.

7. Mining

(max. 12 b za popis, 8 b za program)

Bruteforce

Úlohou je nájsť *najlepší* spôsob, ako rozdeliť počítače na dve skupiny. Najpriamejší spôsob, ako toto docieľiť je vyskúšať všetky spôsoby a zistiť, ktorý je najlepší. Budeme si pamätať, aký najlepší výsledok dokážeme dosiahnuť (na začiatku to bude 0).

Každý z počítačov môže buď ťažiť alebo ho predáme. To sú dve možnosti na každý počítač, teda celkový počet spôsobov, ako rozdeliť počítače je 2^k . Pre každý z nich sa pozrieme, aký je celkový zárobok (súčet predaných počítačov) a celkový výkon (súčin ťažiacich počítačov). Pokiaľ sú tieto dve čísla rôzne, tak ideme na ďalší spôsob. Ak sú rovnaké, tak vieme, že tento spôsob je vhodný, a teda aktualizujeme najlepší výsledok, ktorý si pamätáme.

Keďže celkový počet spôsobov je 2^k a pre každý sa pozeráme na k počítačov, celková časová zložitosť je $O(k2^k)$. Pamäťová zložitosť takéhoto prístupu je $O(k)$, lebo pre každý počítač si pamätáme, do ktorej skupiny patrí.

Niečo lepšie

O každom čísle vieme, že existuje práve jeden spôsob, ako ho rozdeliť na súčin prvočísel⁴. Keby sme si teda “tipli” odpoveď, tak vieme ľahko skontrolovať, či je “správna”. Presnejšie povedané, vieme zistiť, či sa dajú počítače rozdeliť tak, aby bol zárobok aj výkon rovnaký ako toto číslo x .

Aby bol výkon rovný x , tak vždy existuje iba jeden spôsob, ako môže vyzeráť sada počítačov, ktoré ťažia. Tú nájdeme tak, že zistíme prvočíselný rozklad čísla x . V prvom rade overíme, či máme dost počítačov, na dosiahnutie tohoto výkonu. Potom všetky ostatné počítače musíme prediť, tak zistíme zárobok (ich súčet). Nakoniec overíme, či aj zárobok je x a podľa toho rozhodneme, či toto x vyhovuje.

Keď vieme, ako overiť jednu odpoveď, tak stačí len overiť všetky možné odpovede. Všimnime si, že maximálny možný zárobok je m . Skúšať x väčšie ako m teda nemá zmysel, lebo to určite nemôže byť odpoveď. Stačí teda vyskúšať všetky čísla od 1 po m .

Prvočíselný rozklad sa dá jednoducho nájsť v čase $O(\sqrt{m})$. Potom v $O(n)$ môžeme zistiť, či máme z každého prvočísla dostatočný počet počítačov, a zároveň zistiť zárobok z ostatných počítačov. Celková časová zložitosť teda bude $O(m(\sqrt{m} + n))$. Pamäťová zložitosť je $O(n)$.

Zaujímavá myšlienka

Označme

$$P = \max_{i=1}^n p_i$$

Ako sme už zistili, zárobok a teda aj výkon nemôžu byť väčšie ako m . Teraz skúsime odhadnúť zhora *súčet* počítačov, ktoré ťažia. Keďže každý má aspoň 2 jadrá, tak ich môže byť najvyššie $\log_2(m)$. Zároveň, každý z nich má najvyššie p jadier, čo znamená, že ich súčet môže byť najvyššie $T \leq P \log_2(m)$.

Uvedomme si ale, že celkový počet jadier sa nedá zmeniť. Z toho vyplýva, že ak opäť označíme x zárobok (súčet predaných počítačov), tak platí

$$x + T = m$$

$$x = m - T \geq m - P \log_2(m)$$

Vidíme, že x je ohraničené aj zdola, teda môže nadobudnúť najvyššie $P \log_2(m)$ hodnôt. Menšie x sa už neoplatí skúšať, lebo ako sme zistili, už nemôžu vyhovovať.

Optimálne riešenie

Túto myšlienku môžeme ľahko zakomponovať do nášho lepšieho riešenia. Postup bude rovnaký, len nebudeme skúšať všetky x od 1 po m , ale začneme na $m - P \log_2(m)$. Ešte je tu ale jeden problém, a to že hľadať prvočíselný rozklad čísel rádu $m \approx 10^{17}$ nie je jednoduché.

My však nepotrebuje hľadať celý rozklad, pretože keby malo byť toto číslo deliteľné prvočíslom väčším ako P (alebo ľubovoľným iným, ktoré nemáme k dispozícii), tak už nemôže vyhovovať, lebo by sme aj tak nedokázali vyskladať takýto výkon. Teda stačí robiť niečo ako štandardný prvočíselný rozklad, ale rozkladané číslo budeme skúšať deliť iba n prvočíslami zo vstupu, ktoré reprezentujú počty jadier daných počítačov. To bude mať časovú zložitosť $O(n + \log_2(m))$

⁴Tento fakt je známy ako [základná veta aritmetiky](#)⁵.

Overiť jedno číslo vieme teda v čase $O(n + \log_2(m))$, čiže celková časová zložitosť je $O(P \log_2(m)(n + \log_2(m)))$. Pamäťová zložitosť je stále $O(n)$.

Ralbo

8. Ešte treba kúpiť internet

(max. 12 b za popis, 8 b za program)

Dostali sme strom s n miestami a $n - 1$ optickými linkami. Každú linku vieme buď raz odkúpiť za B peňazí, alebo prenajať za R peňazí za každú jednotku času, počas ktorej ju používame. Ďalej máme m požiadaviek tvaru „od času l po čas r musí byť použiteľná celá trasa medzi miestami a a b “. Chceme minúť čo najmenej.

Kľúčovým pozorovaním je, že o tom, či kúpime alebo prenajmeme každú linku, sa môžeme rozhodnúť nezávisle. Tým pádom celková cena je proste súčet cez všetky linky:

$$\text{cena} = \sum_{\text{linka } e} \min(B_e, R_e \cdot T_e)$$

kde T_e je počet jednotiek času, v ktorých linku e potrebuje aspoň jedna požiadavka. Pre každú linku sa teda nezávisle rozhodneme pre lacnejšiu z dvoch možností: buď ju raz odkúpime za B_e , alebo ju prenajímame za $R_e \cdot T_e$.

Zostáva zistiť T_e . Linka e je potrebná v tých časoch, ktoré pokrýva aspoň jedna požiadavka vedúca cez e . T_e je teda veľkosť zjednotenia intervalov $[l_i, r_i]$ tých požiadaviek, ktorých trasa prechádza cez e .

Celá úloha sa tým rozpadá na dve nezávislé otázky:

- ktoré požiadavky vedú cez danú linku a
- ako rýchlo vieme spočítať veľkosť zjednotenia ich intervalov.

Jednotlivé sady odmeňujú postupné odpovede na tieto otázky.

Riešenie hrubou silou

Strom si zakoreníme (napr. v mieste 1). Linky potom budeme indexovať pomocou spodnejšieho z vrcholov, ktoré spájajú - teda ak hrana spája vrcholy v a u , kde v je otcom u , linka bude mať index u .

Pre každú požiadavku prejdeme trasu $a \rightarrow b$: hlbší z koncov vždy posunieme k jeho otcovi, až kým sa oba nestretnú v ich najnižšom spoločnom predkovi. Každej linke na trase pripíšeme interval $[l, r]$.

Označme $t_{\max}$ najväčší možný čas (horná hranica r). Keď sú časy malé (sady 1–3, $r \leq t_{\max} \leq 1000$), vieme pre každú linku držať pole cez čas a priamo si odškrtnávať obsadené jednotky času; T_e je potom počet odškrtnutých. Aby sme každý interval neoznačovali po jednotkách, použijeme diferenčné pole: pri intervale $[l, r]$ pripočítame $+1$ na pozíciu l a -1 na pozíciu $r + 1$. Nakoniec pole každej linky raz prejdeme, spravíme prefixové súčty a spočítame pozície s kladnou hodnotou — to je T_e . Rozdávanie ± 1 stojí $O(n \cdot m)$ (každá z m požiadaviek prejde až n liniek) a záverečné prechody polí $O(n \cdot t_{\max})$, dokopy $O(n \cdot m + n \cdot t_{\max})$.

Veľké časy: zlievanie intervalov

Keď je rozsah času veľký ($t_{\max} \leq 10^9$, sada 4), políčka v poli cez čas si dovoliť nemôžeme. Namiesto odškrtnávania si pre každú linku len nazbierame zoznam intervalov tých požiadaviek, čo cez ňu vedú. Na konci intervaly každej linky utriedime podľa začiatku a zametaním zlejeme prekrývajúce sa a nadväzujúce kusy do blokov. Potom súčet dĺžok blokov je T_e .

Zametanie robíme jednoducho. Prechádzame utriedené intervaly podľa začiatku a pamätáme si, ktorý najbližší čas nie je pokrytý. Ak interval, ktorým prechádzame, začína až po najbližšom nepokrytom čase, tak si vyhodnotíme celý predchádzajúci zjednotený blok. Inak iba predĺžime predchádzajúci zjednotený blok.

Prechod každej trasy je $O(n)$ (keďže môže prechádzať až cez n vrcholov), takže na prejdienie všetkých trás použijeme spolu $O(n \cdot m)$, plus $O(m \log m)$ na triedenie. Pri $n, m \leq 2000$ (sada 4) to bez problémov stačí a riešenie zvláda ľubovoľne veľké časy. Na veľkej sieti (sady 5–8) však $O(n \cdot m)$ nestačí – na reťazi dĺžky $4 \cdot 10^4$ je jediná trasa dlhá až $O(n)$ a dokopy to je priveľa.

Veľká sieť

Na plný počet musíme obe čiastkové úlohy vedieť vyriešiť rýchlo.

Dĺžka zjednotenia.

Potrebujeme štruktúru, ktorá udržiava meniacu sa množinu intervalov a vie vrátiť dĺžku ich zjednotenia. Na to vieme použiť intervaláč. Časy najprv skomprimujeme (rôznych hraníc l a $r + 1$ je iba $O(m)$) a nad elementárnymi (skomprimovanými) intervalmi medzi nimi postavíme intervalový strom. V každom vrchole držíme,

koľkokrát je jeho rozsah celý pokrytý (`cely_pokryty`), a akú dĺžku v jeho rozsahu pokrýva aspoň jeden interval (`pokryty_rozsah`).

Dĺžku pokrytú aspoň jedným intervalom udržiavame bez posúvania lazy nadol. Pri update-e zostúpime do uzlov, ktoré upravovaný rozsah pokrýva celé, a v nich zmeníme `cely_pokryty` o ± 1 . Pri vracaní z rekurzie (teda už sme v oboch deťoch updatli `cely_pokryty`) prepočítame `pokryty_rozsah` uzla z jeho detí: ak má tento uzol `cely_pokryty` > 0 , je pokrytá celá jeho dĺžka; inak je `pokryty_rozsah` súčet hodnôt detí (v liste nula).

Pridanie či odobranie intervalu je range update na intervale o ± 1 v $O(\log m)$; dĺžku zjednotenia prečítame v koreni v $O(1)$.

Ktoré požiadavky vedú cez linku.

Linku (v , otec) používa požiadavka práve vtedy, keď v podstrome vrchola v leží práve jeden z jej dvoch koncov. Chceme teda zistiť, koľko liniek a v akých časových intervaloch bude prechádzať cez vrchol v do jeho otca. Toto zistíme z intervaláča, ktorý sme si vyššie vysvetlili. Pre daný vrchol si však tento intervaláč potrebujeme vytvoriť, a robíme nasledovne: Keď spracovávame nejaký podstrom, tak najprv rekurzívne spracujeme jeho deti (skonštruujeme si ich intervaláč). Potom množiny „aktívnych“ požiadaviek pre jednotlivé linky zlievame technikou *small-to-large*: Z dieťaťa s najväčším podstromom si zoberieme celý intervaláč. Pre ostatné deti si zoberieme zoznam ich „aktívnych“ požiadaviek a pridáme ich do tohto intervaláča alebo ich z neho uberieme, ak tam už sú (ak tam už sú, vieme, že táto cesta už vyššie do stromu nepôjde).

Prečo je to efektívne? Predstavme si ľubovoľnú požiadavku. Keď ju pridávame do intervaláča, podstrom, do ktorého ju dvíhame, má aspoň dvakrát toľko vrcholov ako ten, z ktorého prišla. (Inak by totiž bola vo väčšom z nich — a ten sme si ako ťažké dieťa zobrali „zadarmo“, takže by sme ju nepresúvali.) Počet vrcholov v jej podstrome sa takto môže zdvojnásobiť najviac $\log_2 n$ -krát, čiže každú požiadavku presunieme najviac $\log_2 n$ -krát. Spolu je to $O(m \log n)$ prepnutí, každé za $O(\log m)$ v intervaláči — celková časová zložitosť je teda $O((n + m) \log n \log m)$.

Vďaka tejto technike intervaláč pri spracovaní vrcholu v obsahuje presne tie požiadavky, čo majú v jeho podstrome jediný koniec — čiže tie, ktoré používajú linku k otcovi. Prečítame dĺžku zjednotenia z koreňa a pripočítame $\min(B_v, R_v \cdot T_v)$.

Dynamický intervaláč namiesto kompresie.

Zjednotenie sa dá počítať aj bez kompresie súradníc — dynamickým intervaláčom nad celým časom $[0, 2^{30})$, ktorý alokuje vrcholy až keď treba. Je to elegantné, ale každá operácia prejde 30 úrovni namiesto $\log m \approx 17$ pri kompresii, takže je asi 2-krát pomalší. Hlbku by sa dalo zraziť ešte nižšie kompresiou ciest (skracovaním jednodetných reťazí v intervaláči), lenže kvôli horším konštantám (strom je roztrúsený po pamäti, takže cache nefunguje efektívne) je to v praxi výrazne pomalšie ako súradnicová kompresia.