Korešpondenčný seminár z programovania
XLI. ročník, 2023/24
FMFI UK,
Mlynská Dolina, 842 48 Bratislava

# Vzorové riešenia 1. kola letnej časti

Jacub

## 1. Lúka neštastia

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

Keďže všetky obdĺžniky pretínajú os $y = 0$, či sa pretínajú alebo nie závisí len od ich $x$-ovej súradnice. $y$-ovú súradnicu tak môžeme ignorovať.

Naše riešenie použije greedy pravidlo: Prejdeme zľava doprava a vyberieme každý obdĺžnik, ktorý sa neprekrýva so žiadnym čo sme už vybrali. Čiže ak sa nejaké dva prekrývajú, vyberieme ten ľavejší z nich. Keby sme vybrali ten viac vpravo, a za nimi by bol tretí čo sa prekrýva s druhým ale nie s prvým, nemohli by sme ho vybrať.

Táto úvaha sa opiera o fakt, že obdĺžniky majú v tejto úlohe rovnakú šírku. Vďaka tomu vieme, že ten čo viac vľavo začína aj viac vľavo končí. Ak ho zoberieme, nemôže nám to uškodiť a možno vďaka tomu neskôr budeme môcť vybrať ďalší obdĺžnik, ktorý by bol v inom prípade zbytočne zablokovaný. Keby mali rôzne šírky mohlo by nastať, že ten ľavý je hrozne široký a nehodí sa ho zobrať, čo by bola ťažšia úloha.

Takže implementácia bude vyzerať takto: budeme si pamätať koľko obdĺžnikov sme už vybrali a kde sa nachádzal posledný vybraný. Potom postupne čítame polohy obdĺžnikov – našťastie už sú na vstupe zoradené podľa $x$. Ak sa neprekrýva s posledným čo sme vybrali alebo je prvý na vstupe, inkrementujeme počet a zapamätáme si kde je. Časová zložitosť je $O(n)$.

Všimnime si, že polohy obdĺžnikov ani nemusíme načítať do poľa, môžeme každú vyhodnotiť a spracovať okamžite keď ju prečítame. Optimálna pamäťová zložitosť je preto $O(1)$. Mimochodom takéto algoritmy, ktoré dokážu spracovať vstup za behu, bez toho, že by ho celý vopred načítali, sa volajú "online algoritmy".

Tomi

## 2. Elity filmového plátna

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

Pôvodné zadanie síce bolo detailne naplánovať, kto presne bude kedy moderovať, ale vlastne stačí keď určíme v ktorých momentoch sa majú striedať. Na konkrétnych moderátoroch nezáleží. Chceme iba rozdeliť ceremóniu na niekoľko úsekov, v rámci ktorých sa moderátor nemení. Čo o nich musí platiť? Toto: Rozdelenie na úseky je korektné práve vtedy, keď sa v každom úseku vyskytuje maximálne $h - 1$ rôznych víťazov, t.j. keď je aspoň jeden herec čo počas toho úseku nič nevyhrá.

(Dôkaz: Z každého rozdelenia, ktoré spĺňa túto podmienku, by sme ľahko mohli v ďalšom kroku vyrobiť detailný plán. Proste pre každý úsek vyberieme ako moderátora toho herca, ktorý nič nevyhral. Ak sú viacerí, ľubovoľného. Ale náš program nemusí vypisovať zoznam moderátorov, iba počet striedaní, takže toto nebolo treba. A naopak rozdelenie, ktoré túto podmienku nespĺňa, zjavne nemôže byť korektné, lebo v takom úseku by niekto určite odovzdal cenu sám sebe.)

Na výpočet optimálneho rozdelenia použijeme greedy (pažravý) algoritmus. Proste ideme od začiatku do konca, a vždy sekáme úseky čo najneskôr ako môžeme. Pamätáme si množinu hercov čo niečo vyhrali v aktuálnom úseku. Keď uvidíme herca ktorý v nej ešte nie je, pridáme ho tam. Ale ak už ich tam je $h-1$, respektíve keby po pridaní počet stúpol na $h$, musíme začať nový úsek. V tom prípade sa ten herec stáva prvým hercom z nového úseku. (Dajte si pozor – asi najčastejšia chyba bola na toto zabudnúť a začať prázdny nový úsek bez neho.)

Aktuálnu množinu hercov si môžeme pamätať ako hashset ich mien (v Pythone `set()`, v C++ `std::unordered_set<std::string>`). Sú aj iné dobré možnosti, ale asi druhá najčastejšia chyba bola použiť proste zoznam mien (`[]`, `std::vector<std::string>`). To je príliš pomalé, lebo kontrolovať či sa hodnota nachádza v zozname (`if herec in zoznam`) trvá lineárne od jeho dĺžky.

Časová zložitosť je $O(h + k)$ a pamäťová $O(h)$.

### Dôkaz

Ako vždy, pri greedy algoritmoch treba zdôvodniť alebo formálne dokázať, či naozaj fungujú optimálne. Vo všeobecnosti séria lokálne optimálne vyzerajúcich krokov nie vždy naozaj vedie k globálne optimálnemu cieľu.

Dokážeme to napríklad takto: majme *naše rozdelenie* ktoré by vypočítal náš algoritmus, a nejaké *konkurenčné rozdelenie* ktoré sa od neho líši. Ukážeme, že konkurenčné rozdelenie môžeme postupnými malými zmenami

prerobiť na naše. Po každej zmene bude rovnako dobré alebo lepšie, a bude ich konečne veľa. Z toho vyplynie, že naše rozdelenie je optimálne – rovnako dobré alebo lepšie než ľubovoľné konkurenčné.

Pozrime sa na najskoršie miesto, na ktorom sa líšia – jedno tvrdí "teraz striedaj" a druhé nie. Určite to konkurenčné tvrdí "teraz striedaj" a naše nie, lebo náš algoritmus z definície strieda najneskôr kedy môže. Tak to konkurenčné rozdelenie trošku upravme: posuňme to striedanie o jedna ďalej.

Takto upravené konkurenčné rozdelenie je stále korektné: predošlý úsek síce trochu narástol, ale ešte stále má maximálne $h - 1$ rôznych hercov (inak by náš algoritmus tiež na tomto mieste striedal), a ďalší úsek sa trochu zmenšil, ale to je určite vždy v poriadku. Počtom úsekov je nové konkurenčné rozdelenie rovnako dobré ako doterajšie konkurenčné, alebo dokonca lepšie (ak tam doteraz bol úsek dlhý 1 a v upravenom zmizne).

Tento postup opakujeme, až kým z konkurenčného rozdelenia nedostaneme postupnými úpravami presne to isté rozdelenie ako vyrobil náš algoritmus. V každom kroku sme zachovali korektnosť a nezmenili alebo zlepšili počet úsekov. Opakovanie časom musí skončiť, lebo dĺžka prefixu čo sa zhoduje s naším algoritmom stále rastie.

Z toho vyplýva, že nech bolo pôvodné konkurenčné rozdelenie akékoľvek, náš greedy algoritmus určite nájde rovnako dobré alebo lepšie.

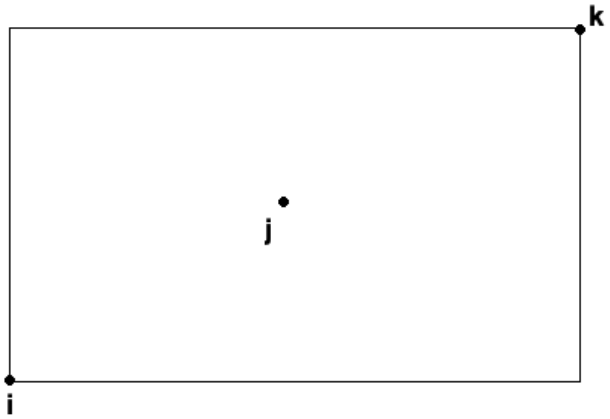
Janka

## 3. Gigantický Legoland

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

Prvé dôležité pozorovanie pre riešenie tejto úlohy je uvedomiť si, aké vlastnosti má trojica, ktorá sa v úseku nemôže nachádzať. Keď sa v Legolande hýbeme medzi dvoma atrakciami, bez ujmy na všeobecnosti si vieme povedať, že pri prechode z atrakcie číslo $i = (i, r_i)$ k atrakcii číslo $k = (k, r_k)$ musíme spraviť $d = k - i$ horizontálnych skokov a $v = max(r_i, r_k) - min(r_i, r_k)$ vertikálnych skokov. Poradie, v ktorom však týchto $d$ skokov doprava/doľava a $v$ skokov hore/dole spravíme môže byť ľubovoľné. To znamená, že ak si nakreslíme obdĺžnik tak, že atrakcie $i$ a $k$ sú jeho protiľahlé rohy, môžme si vybrať kombináciu skokov tak, že vieme navštíviť každý bod v tomto obdĺžniku.

Teda pre trojicu atrakcií $i, j, k$, platí $v(i, k) = v(i, j)+v(j, k)$, ak atrakcia $j$ leží v obdĺžniku, ktorého protiľahlé vrcholy sú atrakcie $i$ a $k$ (pri predpoklade, že sú v tomto poradí aj usporiadané).




@H

Formálne $i < j < k$ a zároveň $min(r_i, r_k) \leq r_j \leq max(r_i, r_k)$.

Z tohto vieme usúdiť, že atrakcie $i$, $j$ a $k$ môžu tvoriť zlú trojicu len v dvoch prípadoch. Buď ich ypsilonové súradnice musia tvoriť nerastúcu podpostupnosť alebo neklesajúcu podpostupnosť v rámci nami vybratého úseku. Ak teda chceme aby atrakcie $i$, $j$ a $k$ netvorili zlú trojicu, musí pre ne platiť, že atrakcia, ktorá sa nachádza horizontálne v strede ($j$) musí byť vertikálne buď najvyššie, alebo najnižšie zo všetkých troch atrakcií. Formálne, ak sa atrakcia $j$ nachádza v strede, musí platiť $r_i, r_k < r_j$ alebo $r_i, r_k > r_j$. V opačnom prípade budú určite tvoriť zlú trojicu. Toto môžme generalizovať a povedať, že každá atrakcia v úseku, ktorá je obklopená aspoň jednou atrakciou aj zprava aj zľava sa musí nachádzať vrámci celého úseku vertikálne buď najvyššie alebo najnižšie zo všetkých atrakcií. V opačnom prípade, keď nie je ani najvyššie, ani najnižšie, bude v úseku na jednej strane iná atrakcia vyššie (alebo zarovno) a na druhej iná atrakcia nižšie (alebo zarovno). Tieto tri atrakcie by tvorili zlú trojicu.

### Veľmi hrubá sila

Bruteforce riešenie tohoto problému spočíva v prechádzaní všetkých úsekov a skontrolovaní všetkých trojíc v každom úseku.

Počet úsekov dĺžky $\geq 3$ je $O(n^2)$ a v každom úseku je $O(d^3)$ trojíc ktoré treba skontrolovať, kde $d$ je dĺžka úseku. Horný odhad je teda $O(n^5)$.

Nemusíme si pritom pamätať nič okrem bodov na vstupe, pamäťová zložitosť je teda $O(n)$.

### Zaujímavá myšlienka

Poďme sa pozrieť, aké dlhé môžu byť úseky, v ktorých sa zlá trojica nenachádza. Je očividné, že úseky dĺžky 1 a 2 určite zlú trojicu obsahovať nebudú a teda všetky tieto úseky môžme rovno zarátať k výsledku. Pri úsekoch dĺžky 3 je už možné, že 3 vrcholy v úseku budú tvoriť neklesajúcu alebo nerastúcu postupnosť vtedy, keď sa stredná atrakcia nachádza vertikálne medzi zvyšnými atrakciami. Zároveň však vieme jednoduhco nájsť prípad, kedy úsek dĺžky 3 neobsahuje zlú trojicu a môžme ho prirátať k výsledku - t.j. keď je stredná atrakcia najvyššie alebo najnižšie. To znamená, že úseky dĺžky 3 musíme prejsť a skontrolovať, aby sme vedeli, či zlú trojicu obsahujú.

Poďme sa pozrieť na úseky dĺžky 4. V tomto prípade sa v úseku nachádzajú presne dve atrakcie, ktoré sú obklopené inými atrakciami z oboch strán a môžu byť stredom zlej trojice. No zároveň stále existuje situácia, keď jedna z týchto atrakcií je v rámci úseku najvyššie a druhá najnižšie a vtedy je celý úsek bez zlej trojice a môže byť prirátaný k výsledku. To znamená, že aj všetky úseky dĺžky 4 musíme po jednom prejsť a osobitne skontrolovať.

Poďme sa pozrieť na úseky dĺžky 5. V tomto prípade sa v úseku už nachádzajú 3 atrakcie, ktoré sú obklopené z oboch strán. Ak každá obklopená atrakcia musí byť buď najvyššie alebo najnižšie, prichádzame k sporu. Ak napríklad druhú atrakciu necháme nachádzať sa najvyššie a tretiu najnižšie, štvrtá atrakcia sa už nebude môcť nachádzať ani najvyššie, ani najnižšie. Môžme teda povedať, že úseky dĺžky 5 (a viac) nemusíme kontrolovať, lebo budú vždy obsahovať zlú trojicu.

### Vzorové riešenie

Vzorové riešenie teda len využije toto pozorovanie - stačí manuálne overiť všetky trojice v úsekoch dĺžky 3 a 4 na zistenie, či sú úseky dobré alebo zlé, a prirátať počet úsekov dĺžky 1 a 2, ktoré sú určite dobré.

Prejdenie úsekov má časovú zložitosť $O(n)$ a skontrolovanie všetkých trojíc v úsekoch dĺžky 3 a 4 má časovú zložitosť O(1). Teda časová zložitosť nášho riešenia je $O(n)$.

Zároveň si stále nemusíme pamätať okrem bodov a konštantne veľa pomocných premenných nič naviac, pamäťová zložitosť je teda $O(n)$.

Fipo

## 4. Odporné psiská, FUJ!

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

Mesto, kde Dog Show Winner býva si vieme reprezentovať ako graf – ulice sú hrany a miesta kde sa ulice spájajú (väčšinou sa im hovorí križovatky) budú vrcholy. Každej hrane potom priradíme váhu – to, koľko ňuchá pes na danej ulici. Keď chceme nájsť odpoveď, musíme nájsť najmenšie číslo $R$ také, že existuje kostra grafu tvorená hranami, ktorých bitový $OR$ je rovný $R$.

Nech najväčšia $OR$ všetkých váh hrán na vstupe je $W$.

### Bruteforce

Najviac bruteforce prístup by bol nájsť všetky kostry grafu a z nich vybrať tú s najmenším bitovým _OR_om ich hrán. Časová zložitosť je $O(v \cdot \binom{e}{v-1})$. Toto riešenie by prešlo iba prvou sadou.

Iný, oveľa lepší bruteforce je skúšať nie všetky kostry, ale všetky výsledky. Nech $R$ je želaný výsledný $OR$ kostry a k nemu skúsime dorobiť kostru grafu. $R$ budeme hľadať tak, že postupne budeme prechádzať čísla od 0 po $W$. Čo potrebujeme skontrolovať je, či existuje kostra grafu s bitovým _OR_om $R$. To môžeme spraviť tak, že "povyhadzujeme" z grafu všetky hrany, ktoré by nám pokazili želaný výsledok, a teda také, ktorých bitový $OR$ s $R$ je väčší ako $R$. Takto vytvorený nový graf môže mať bitový $OR$ najviac $R$.

V tomto novom grafe sa už nemusíme zaoberať váhami hrán, ale iba ich existenciou. Teda iba potrebujeme skontrolovať, či je náš graf spojitý. To môžeme spraviť pomocou Union-Find alebo DFS (ak sa dostaneme pomocou DFS do každého vrcholu grafu, tak graf je aj napriek vyhodeným hranám súvislý). Keďže prechádzame $R$ v rastúcom poradí, tak prvé $R$ na ktoré narazíme je správna odpoveď. Toto riešenie má časovú zložitosť $O(W \cdot e)$ a pamäťovú zložitosť $O(e)$.

Upozornime, že riešenie ktoré hrany reálne nevyhadzuje (teda nevytvára nový graf), je aj rýchlejšie aj jednoduchšie na implementáciu. Prekvapivo, aj keď má Union-Find horšiu časovú zložitosť ($O(e\alpha(e))$), tak v praxi je na tejto úlohe oveľa rýchlejší než DFS.

### Optimálne riešenie

Na predchádzajúcom riešení je časovo dominantný faktor iterovanie $R$ po $W$. Musíme nájsť efektívnejší spôsob hľadania $R$. Vieme, že $R$ môže byť ľubovoľné číslo od 0 do $W \approx 2^{31}$. Budeme sa na potenciálne riešenie pozerať po bitoch.

Najskôr zistíme, či môže byť bit s najväčšou váhou nastavený na 0. To zistíme tak, že vyhodíme z grafu všetky hrany, ktoré na tom mieste majú 1. Robíme teda rovnaké riešenie ako bruteforce, ale s $R = 011111...111_2$. Ak náš graf zostal po vyhodení hrán spojitý, tak určite existuje $R$ také, že má na najvyššej pozícii 0. Naopak, ak graf po vyhodení hrán s 1 na najvyššom mieste prestal byť spojitý, nemôže existovať žiadne $R$ také, že má na najvyššom mieste 0 (nespojitý graf nemá kostru). Keď sme našli najväčší bit, presunieme sa na ten menší.

Toto riešenie má časovú zložitosť $O(e \log(W))$. Ale keďže počet bitov môže byť v tejto úlohe najviac 30 a väčšinou sa ako premenná neudáva, môžeme to písať ako $O(e)$. Pamäťovú zložitosť má toto riešenie $O(e)$.

### Dôkaz

Bitmi musíme prechádzať od najväčšieho po najmenší. Dáva to intuitívne zmysel, keďže chceme najprv prioritizovať bity, ktoré majú väčší dopad na výsledok. Taktiež ľahko vidno, že opačné poradie absolútne nefunguje na grafe, kde existuje kostra s váhami iba 1 a druhá s váhami iba 2. Na tomto grafe by opačné poradie zahodilo všetky 1 hrany a skončilo by s $R = 2$.

Vidno, že opačný postup vôbec nefunguje. Aby sme ukázali, že postup od najväčších po najmenšie bity funguje, musíme ukázať, že ak existuje riešenie, tak ho tento postup nájde a zároveň bude najmenšie možné.

To, že nejaké riešenie nájde, je zrejmé – prinajhoršom prídeme ku $R = 111...111_2$, pri ktorom vieme použiť všetky hrany, a teda určite nájdeme kostru. Prečo bude najmenšie možné? Vyplýva to z toho, že najvyšší bit je dôležitejší než všetky ostatné dokopy. Ak nemusíme použiť najvyšší bit, nikdy sa nám neoplatí ho použiť, keďže $0111...111_2 < 1000...000_2$.

Výborne, máme riešenie a ukázali sme, že je správne. Ako nás ale mohlo niečo takéto napadnúť? No toto čo robíme je iba binárne vyhľadávanie výsledku. Ale čo je na ňom špeciálne je, že ho musíme robiť opatrne. Štandardne, keď robíme binárne vyhľadávanie výsledku, tak ak riešenie existuje pre nejaké $R$ tak existuje aj pre všetky $R' > R$ a podobne ak neexistuje pre $R$ tak neexistuje ani pre $R' < R$. V tomto prípade si vieme ľahko predstaviť, že ak existuje riešenie pre $R = 10010_2$, tak nemusí existovať pre $R' = 10100_2$ alebo naopak. Ale platí, že ak neexistuje pre $R$ tak nebude existovať ani pre žiadne $R'$, ktoré majú iba podmnožinu bitov nastavených na 1. V našom prípade teda zoberieme $R$ s čo najviac bitmi nastavenými na 1 a ak neexistuje riešenie, tak nemusíme skúšať ani žiadnu podmnožinu. A ak naopak riešenie existuje, tak ako sme spomínali, môžeme sa sústrediť iba na tieto podmnožiny.

fezjo

## 5. LTT

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

### Najjednoduchšie riešenie

Skúsme najprv vymyslieť a naprogramovať čo najjednoduchšie riešenie. No a čo je jednoduchšie, než vyskúšať všetky možnosti? Dôležité však je, aby aj implementácia bola jednoduchá – v tomto prípade sa nám preto hodí rekurzia.

Povedzme, že sa nachádzame na pozícii *poz* a máme obuté šľapky veľkosti *vel*. Nechceme tu zostať trčať, takže sa pohneme. Pred tým nás ale čaká rozhodnutie (prezuť či neprezuť) a my vyskúšame obe možnosti. Toto vieme jednoducho vyjadriť rekurzívnym vzťahom:

```
n, way = input().split()
n = int(n)
velkosti = list(map(int, input().split()))

def kroky(poz, vel):
    if poz >= n:
        return 0
    return 1 + min(kroky(poz + vel, vel), kroky(poz + velkosti[poz], velkosti[poz]))

print(kroky(0, velkosti[0]))
```

Keďže sa vždy priblížime k cieľu o aspoň jeden meter, vykonáme najviac $n$ krokov. Časová zložitosť tohto riešenia je $O(2^n)$ a pamäťová $O(n)$ (pamätáme si pole veľkostí, ale nezabudnime ani na rekurziu hĺbky $n$). Za takýto kód dostaneme **2 body** a celkovo má iba 10 riadkov!

Akú časovú zložitosť by mala rekurzia, ktorá by sa vetvila iba ak je `vel` rôzne od `velkosti[poz]`? Odpoveď: [1]

### Oveľa lepšie a pritom skoro rovnaké

Spomenuté riešenie má tri problémy na zvyšných sadách:

1. Dostávame `EXC`, pretože rekurzia je príliš hlboká.
2. Ak by sme aj nedostali `EXC`, tak by sme dostali `TLE`, pretože máme exponenciálnu zložitosť.
3. Ak by sme aj nedostali `TLE`, tak by sme dostali `WA` na poslednej sade, pretože nezohľadňujeme kroky smerom späť.

Čo s tým? Prvý problém opravíme jednoducho tak, že zvýšime limit rekurzie pomocou `sys.setrecursionlimit`.

Druhý problém opravíme tiež jednoducho, **memoizáciou**. Funkcia `kroky` pre rovnaké parametre vždy vráti rovnaký výsledok, takže si ho môžeme uložiť a pri ďalšom volaní ho vrátiť. Keďže máme dva parametre, oba v rozsahu od 0 do $n$, existuje iba $n^2$ možností. Časová zložitosť tohto riešenia bude potom iba $O(n^2)$. V Pythone vieme memoizáciu jednoducho prilepiť na každú funkciu pomocou dekorátora `functools.cache`.

Tretí problém vieme tiež jednoducho opraviť tak, že do rekurzívneho vzťahu pridáme aj kroky smerom späť.

No aha, aké jednoduché!

Avšak trochu som klamal. Je pravda, že každý problém sme takto vyriešili, druhá a tretia oprava ale nie sú navzájom kompatibilné. Ak však upustíme od chodenia dozadu, dostávame riešenie s časovou a pamäťovou zložitosťou $O(n^2)$, pridali sme iba 4 riadky a dostaneme až **6 bodov**. Great success!

### Memoizácia smerom dozadu

Čo je vlastne ten problém s chodením dozadu? V rekurzii vieme spraviť v šľapkách krok tam a späť čím sa dostaneme na rovnakú kombináciu parametrov funkcie. Keďže sme pre túto kombináciu ešte nič nevypočítali, začneme ju počítať znova a zacyklíme sa.

Keď zvyčajne robíme DFS na cyklickom grafe, pamätáme si o každom vrchole informáciu, či sme ho už navštívili, a značíme si ju, už keď do neho prichádzame, nie až keď z neho odchádzame (ako to vlastne robíme teraz).

```
def dfs(v):
    if navstiveny[v]: return
    navstiveny[v] = True
    ...
```

Možno nás teda napadne, že by sme si do memoizácie tiež mali niečo zapísať hneď, ako vstúpime do funkcie. Potom v prípade, že sa začneme cykliť (čo budeme vedieť detegovať), vrátime ako odpoveď nejakú veľkú hodnotu (nekonečno). Predsa sa nám neoplatí vrátiť do situácie kde sme už boli, takže je to to isté, ako keby sme sa tam ani nepokúsili dostať.

Alebo? To, že sa nám neoplatí vrátiť síce je pravda pre nejakú konkrétnu postupnosť krokov, avšak tento výsledok sa bude memoizovať a bude mať dopad aj na iné postupnosti krokov. Ak sa odmietneme vrátiť do situácie, kde sme už boli, je to ako keby sme z grafu po ktorom sa pohybujeme odstránili danú orientovanú hranu. Ak však optimálna cesta vedie cez túto hranu, tak ju už nikdy nenájdeme! Skúste si nájsť príklad, kde toto nastane.

### Riešenie smerom späť

Samozrejme, kde je memoizácia, tam je aj dynamické programovanie. Predpokladáme, že viete, čo je to dynamika a ako ju implementovať, keď už sme si vysvetlili riešenie na memoizácii. Ak nie, tak toto je výborná príležitosť, aby ste sa to naučili. Nebudeme tu rozpisovať detailný princíp dynamiky, ten viete nájsť v kuchárke[2]. Prečítajte si ju, vráťte sa sem a naprogramujte si túto úlohu.

Rovnako ako pri rekurzií s memoizáciou, aj tu máme časovú a pamäťovú zložitosť $O(n^2)$ a nevieme sa vracať, lebo by sme sa zacyklili alebo dostali zlý výsledok.

[1] $O(2^{\frac{n}{1.5}}) = O(1.5874^n)$

[2] https://www.ksp.sk/kucharka/dynamicke_programovanie/

No, keď už máme tú dynamiku, nemohli by sme ju nejako využiť aj na poslednú sadu? Pri jednom výpočte dynamiky vypočítame všetky optimálne hodnoty pre prípad, že sa vieme hýbať iba smerom dopredu. Spravme preto druhý výpočet dynamiky, spustený na výsledkoch prvého výpočtu, avšak opačným smerom. Pre každý stav si zapamätáme minimum z oboch výpočtov – dostávame hodnotu, ako sa do tohto stavu dostaneme najrýchlejšie, ak sa môžeme hýbať buď smerom dopredu, alebo najprv smerom dopredu a potom niekoľko krokov smerom dozadu. Potom môžeme spraviť ďalší výpočet dynamiky, tentoraz zasa smerom dopredu. Získame výsledky pre trasy, v ktorých môžeme nanajvýš dva krát zmeniť smer. Takto vieme pokračovať v striedaní smerov a počítaní dynamiky, až kým sa nám neprestanú zlepšovať výsledky.

Kedy sa nám prestane zlepšovať výsledok? Keďže optimálne kráčanie má nanajvýš $n$ krokov, v najhoršom prípade môžeme zmeniť smer $\frac{n}{2}$ krát. Takže celková časová zložitosť riešenia je $O(n^3)$ (viete vymyslieť vstup, na ktorom tento najhorší prípad nastane?).

Túto techniku vieme uplatniť na takmer každý prístup, ktorý vie vypočítať optimálne hodnoty jedným smerom, čím sa nám lineárne zhorší časová zložitosť.

### Úloha ako graf

Už sme to tu raz spomenuli a zadanie nám to aj celkom naznačuje – to, čo tu robíme, je vlastne hľadanie najkratšej cesty v grafe.

Reprezentujme si teda úlohu ako graf s ohodnotenými orientovanými hranami. Graf bude mať $n$ vrcholov, jeden pre každú pozíciu, na ktorej môžeme stáť. Predstavme si, že sa vždy prezujeme, keď vojdeme do vrcholu. Z vrcholu $i$ nepôjdu len blízke hrany predstavujúce jeden krok, ale aj drahšie hrany do všetkých vrcholov, kam vieme bez prezutia dokráčať v šľapkách z vrcholu $i$. Čiže pre každý vrchol $i$ a počet krokov $k$ spravíme hranu $(i, i + k \cdot v_i)$ a $(i, i - k \cdot v_i)$ ktorých cena bude $k$.

Takto sme sa zbavili rozhodovania o prezúvaní a môžeme použiť Dijkstrov algoritmus na hľadanie najkratšej cesty v grafe.

Z každého vrcholu vie ísť hrana do každého iného vrcholu (viete, na akom vstupe sa to stane?). Počet vrcholov je $n$ a počet hrán $O(n^2)$. Časová zložitosť bude $O(n^2 \log n)$ a pamäťová $O(n^2)$. Takto za program môžeme dostať až **8 bodov** a stačilo naprogramovať Dijkstru. Pche, však to viem aj poslepiačky!

Ale vieme to spraviť aj lepšie. Samozrejme si nemusíme pamätať všetky hrany, vieme ich kedykoľvek ľahko vypočítať. Pamäťová zložitosť sa nám teda zmenší na $O(n)$. Navyše, Dijkstrov algoritmus vieme upraviť tak, aby na hustých grafoch nepoužíval haldu, ale pole. Takto dostaneme časovú zložitosť $O(n^2)$.

### Riedky graf

Máme 8 bodov, päťka je triviálna, KSP ide dolu vodou. Ale čoby, poďme to vyriešiť ešte lepšie!

Pozrime sa na ten "graf", ktorý sme používali pri memoizácii a dynamike. Tam je každý stav vrcholom a hrany sú prechodmi medzi týmito stavmi. V tomto prípade máme $n^2$ vrcholov a každý z nich má nanajvýš tri hrany (krok dopredu, krok dozadu, prezuť šľapky). Moc sme si nepomohli.

Avšak! V tomto má bežná rekurzia s memoizáciou výhodu voči dynamike – navštívime iba vrcholy, ktoré sú *relevantné* pre riešenie. Ak na vstupe nie sú šľapky veľkosti 17 alebo šľapky veľkosti 42 sú iba na pozícii 9, tak napríklad stavy `[poz=5, vel=17]` alebo `[poz=8, vel=42]` nikdy nenavštívime. Koľko je teda takýchto "*relevantných*" vrcholov takých, ktoré v rekurzii navštívime?

Pri riešení Dijkstrou sme mali problém na vstupe s $n$ jednotkami. Tam sme mali kompletný graf s $n$ vrcholmi a $O(n^2)$ hranami. V tomto riešení by však náš graf obsahoval iba $n$ *relevantných* vrcholov (všetky vrcholy kde `vel=1`). Výborne!

### Najhorší prípad

Čo je teda najhorší možný prípad? Ak máme na vstupe šľapky veľkosti $k$ na pozícii $p$, potom určite všetky vrcholy `[poz=p+x*k, vel=k]` (kde $x$ je celé číslo a $1 \leq p + x * k \leq n$) budú *relevantné*.

Ľahko vidno, že pre najhorší prípad sa nám neoplatí mať viac ako $k$ šľapiek veľkosti $k$, pretože potom by určite niektoré dvoje šľapky generovali rovnakú množinu *relevantných* vrcholov. Konkrétne chceme šľapky veľkosti $k$ rozmiestniť na pozície s rôznymi zvyškami po delení $k$.

Ak toto spravíme, potom šľapky veľkosti $k$ pridávajú približne $\frac{n}{k}$ ($\lfloor \frac{n}{k} \rfloor$ alebo $\lceil \frac{n}{k} \rceil$) *relevantných* vrcholov. Ak chceme maximalizovať počet *relevantných* vrcholov, mali by sme zvoliť čo najmenšie $k$.

Najhorší vstup bude teda tvaru `1 2 2 3 3 3 4 4 4 4 5 ...`. Aké je posledné číslo v tomto vstupe? Chceme vedieť, pre aké najmenšie $m$ je $n \leq \sum_{k=1}^{m} k = \frac{m(m+1)}{2} < \frac{(m+1)^2}{2}$, čiže $\sqrt{2n} - 1 < m$.

Pre jednoduchosť predpokladajme, že pre každé $1 \leq k \leq m$ máme na vstupe $k$ šľapiek veľkosti $k$. Keďže každé $k$ pridáva približne $\frac{n}{k}$ *relevantných* vrcholov, z každého $k$ máme približne $k \cdot \frac{n}{k} = n$ *relevantných* vrcholov. Takže celkový počet *relevantných* vrcholov je $n \cdot m = n \cdot \sqrt{2n} = O(n^{1.5})$.

### Prechod riedkeho grafu

Ukázali sme, že *relevantný* graf má iba $O(n^{1.5})$ vrcholov a hrán. Uvedomme si, že keby sme vedeli spraviť memoizáciu pôvodnej rekurzie, dosiahli by sme takúto časovú zložitosť – to sa nám ale nepodarilo.

Pri prechode je totiž dôležité, aby sme sa necyklili, ale stále prešli po každej hrane (keďže vie byť potenciálne súčasťou optimálnej cesty). To sa dá jednoducho dosiahnuť, ak budeme vrcholy prechádzať v správnom poradí – s klesajúcou vzdialenosťou od cieľa (a teda s rastúcou vzdialenosťou od štartu).

Vieme znova použiť Dijkstru, ale spomeňme si, že v tomto prípade máme neohodnotené hrany, takže nám stačí použiť BFS.

Keďže *relevantný* graf je riedky, vzdialenosti si musíme tiež pamätať v riedkom formáte – použijeme hešovací slovník.

Dosiahli sme časovú a pamäťovú zložitosť $O(n^{1.5})$.

### Všetko to spolu súvisí

Uvedomme si, čo všetko sme doteraz robili:

- BFS,
- Dijkstra,
- striedavé dynamické programovanie,
- jednosmerné dynamické programovanie,
- memoizácia,
- exponenciálna rekurzia.

Možno je to zjavné, keďže všetky riešia tú istú úlohu, ale principiálne naozaj všetky tieto algoritmy robia to isté – skúšajú každú cestu v grafe, aby z nich vybrali tú najkratšiu.

Odmyslime si, že majú tieto algoritmy rôzne názvy a časové zložitosti.

Exponenciálna rekurzia naozaj skúsi (aspoň) raz prejsť každú možnú cestu a zapamätá si tú najlepšiu. V tomto prípade je poradie, v akom to robí, poradím DFS prechodu. Ale rovnako by to mohlo byť ľubovoľné iné poradie, ktoré prejde každú cestu aspoň raz.

Ostatné spomenuté algoritmy robia všetky to isté – menia poradie skúšania ciest. Správnym zvolením poradia skúšania ciest potom vedia dosiahnuť zrýchlenie. Ak som už pre nejaký vrchol našiel optimálny prefix (BFS, Dijkstra, dynamika) /suffix (memoizácia)/, tak môžem zahodiť cesty, ktoré prechádzajú týmto vrcholom a majú iný prefix /suffix/. Každou takouto úvahou zahodím exponenciálne veľa ciest, čím nakoniec dosiahnem polynomiálnu zložitosť.

V každom z týchto algoritmov sme však museli v nejaký moment zohľadniť každú z tých exponenciálne veľa možných ciest. Ak by sme nejakú možnosť vynechali, mohli by sme prísť o optimálne riešenie. Pre každú neoptimálnu cestu existuje v priebehu daného algoritmu moment, kedy sme sa "pozreli" na jej prefix /suffix/ a povedali si "*táto druhá cesta je aspoň taká dobrá, a preto túto prvú môžem zahodiť*".

Samozrejme, nemohli sme sa priamo individuálne pozrieť na každú z exponenciálne veľa možných ciest (inak by sme mali exponenciálnu zložitosť). Ale každá neoptimálna cesta patrila do nejakej triedy ciest, ktorú sme naraz "zahodili".

Najlepší algoritmus je teda ten, ktorý pre danú úlohu aj v najhoršom prípade zvolí najlepšie poradie skúšania ciest.

### Optimalizácie

No tak sme si teoreticky ukázali šesť algoritmov, ktoré všetky riešia tú istú úlohu. A to sa ešte každý z nich dá naprogramovať rôznymi spôsobmi.

Časové limity boli v tejto úlohe celkom voľné, takže ste mohli byť *kreatívni* v štýle implementácie. Ale to neznamená, že sa nedali napísať rýchle riešenia. Každé riešenie sa spoliehalo na nejakú štruktúru na prehľadávanie "grafu". Pozrime sa konkrétne na BFS riešenie a v krátkosti spomeňme niektoré pamäťové optimalizácie, ktoré môžeme v tomto riešení použiť.

Všetok kód, ktorý tu budeme spomínať, je úplne identický, s rozdielom jednej štruktúry na pamätanie si navštívených vrcholov. Cieľom tejto štruktúry je pamätať si $O(n^{1.5})$ stavov, ktoré sme už navštívili. Počas behu programu do tejto štruktúry pristupujeme veľa krát. Je preto dôležité, aby bola čo najrýchlejšia.

Je intuitívne na tento účel použiť množinu. Ak sme programovali v C++, v štandardnej knižnici máme na výber medzi `std::set` a `std::unordered_set` (mimo STL existujú lepšie možnosti). To, ktorú použiť, sa mení od prípadu k prípadu.

`std::unordered_set` funguje na princípe hešovania, čo znamená, že prístup do nej trvá v priemere $O(1)$ času, avšak s veľkou konštantou, a štandardne nepodporuje ukladanie štruktúr ako `std::pair`.

Ak sa ale rozhodneme použiť `std::unordered_set`, určite sa oplatí použiť `reserve`, tak ako bolo spomenuté v zadaní. V týchto riešeniach nám to zrýchlilo programy od **1.6x** ($C$ vs $D$) až po **2.1x** ($F$ vs $G$). Je samozrejme dôležité zvoliť dostatočne veľkú rezervu, inak nebude mať táto optimalizácia žiaden efekt.

Ak sme sa rozhodli použiť `std::set`, boli sme veľmi smutní, pretože v tomto prípade to bolo vždy **niekoľko krát** pomalšie než rovnaké riešenie s pridaným slovom `unordered_` ($B$ vs $C$, $A$ vs $F$).

Pozrime sa na dve možnosti, ako si pamätať dáta pre stavy, ktoré sú dvojice – množina dvojíc alebo dvojrozmerná štruktúra. Môžeme si to predstaviť tak, že sme namiesto hľadania `[poz, vel]` v množine obsahujúcej $O(n^{1.5})$ prvkov *veľmi rýchlo* v poli na indexe `poz` našli množinu obsahujúcu $O(\sqrt{n})$ prvkov. V Pythone bolo pole množín **2.7x** rýchlejšie než množina dvojíc. V C++ to je trochu komplikovanejšie. V prípade použitia `std::set` bolo pole množín tiež **3x** rýchlejšie ($A$ vs $B$). Keďže `std::unordered_set` nepodporuje ukladanie dvojíc, musíme si napísať vlastnú hešovaciu funkciu. Použitím zlej hešovacej funkcie sme dostali program, ktorý bežal **stovky krát** pomalšie ($C$ vs $E$), ale použitím dobrej program, ktorý bol **1.6x** rýchlejší než pole množín ($C$ vs $F$). Ak sme zároveň použili `reserve` na veľkosť $4 * n^{1.5}$, slovník dvojíc bol dokonca **2.2x** rýchlejší než pole množín ($D$ vs $G$)!

Toto vieme posunúť ešte o krok ďalej. Prístup do `std::vector` aj do `std::unordered_set` je v čase $O(1)$, ale líšia sa v konštantách. Chceli by sme teda, aby sme sa čo najčastejšie dostali k dátam cez prístup do poľa. Rozdelíme si dáta na malé a veľké šľapky do dvoch štruktúr. Jedna bude dvojrozmerné pole a druhá pole množín/množina dvojíc. Dáta pre malé šľapky, teda tie, ktoré sú veľkosťou menšie ako $\sqrt{cn}$, vieme ukladať do poľa s rozmermi $N \times \sqrt{cn}$. Dáta pre veľké šľapky ($\sqrt{cn} <$ veľkosť $\leq n$) budeme ukladať do poľa množín/množiny dvojíc. Konštantu $c$ si môžeme zvoliť tak, aby sme sa zmestili do pamäťového limitu, strávili primerane veľa času vytváraním štruktúr a zároveň sme mali čo najviac prístupov k dátam cez prístup do poľa. Podobným argumentom ako pri počítaní veľkosti *relevantného* grafu vieme ukázať, že takáto optimalizácia v najhoršom prípade zrýchli program o konštantný faktor. A naozaj nám to program zrýchli **3.4x** ($G$ vs $H$)!

Vstupy v tejto úlohe sú také malé, že môžeme použiť $O(n^2)$ pamäťové riešenie a vôbec sa neobťažovať s množinami. Je ale dobre vedieť, že takéto optimalizácie existujú, lebo sa nám môžu hodiť na iných súťažiach. Keďže máme priestor na $O(n^2)$ pamäte, môžeme si dovoliť pole booleanov alebo takzvanú bitovú množinu (`std::bitset` v C++). Kód bude nakoniec veľmi rýchly a veľmi jednoduchý.

Všimnime si, že `std::bitset` je **2x** rýchlejší než `std::vector<bool>`, ten je **3x** rýchlejšie než `std::vector<char>` a ten je zase približne **5x** rýchlejší než `std::vector<int>`. Nie je to náhoda. Čím menej dát potrebujeme ukladať, tým viac sa nám ich zmestí do CPU Cache. Avšak pozor! Často sa stáva, že naopak `std::vector<bool>` je pomalší než `std::vector<char>` (detaily si vygooglite). Podobne sa viete hrať s nahrádzaním `int` za `short/unsigned short`, keďže $n < 2^{16}$. Ak sa vám vhodne podarí zmestiť do CPU Cache, ucítite vietor vo vlasoch. Program $H$ vieme tiež vylepšiť použitím `std::bitset`.

Bol by som zvedavý, aký najrýchlejší kód sa dá napísať. Ak si myslíte, že máte výbornú implementáciu, môžete sa mi pochváliť.

Možno ste si všimli, že som veľa krát spomenul, ako to môže byť v iných prípadoch s optimalizáciami celkom inak. Toto je skutočnosť, ktorú pozná každý kto sa kedy pokúšal žmýkať z programu všetku jeho rýchlosť. Slepá optimalizácia s veľkou pravdepodobnosťou nepovedie k 10x zrýchleniu. Ak chcete niečo optimalizovať, vygenerujte si reprezentatívny ťažký vstup a zmeny v kóde testujte na ňom.

Tabuľka pre vstupy veľkosti $n = 50\,000$:

| ID | Štruktúra | Čas (ms) | Zrýchlenie |
|---|---|---|---|
| $A$ | `set<pair<int, int>>` | 29 805 | 0.15x |
| $B$ | `vector<set<int>>` | 9 944 | 0.45x |
| $C$ | `vector<unordered_set<int>>` | 4 467 | základ |
| $D$ | `vector<unordered_set<int>>` ~ `reserve` | 2 878 | 1.55x |
| $E$ | `unordered_set<pair<int, int>>` ~ zlý heš | $> 100s$ | 0.00x |
| $F$ | `unordered_set<pair<int, int>>` | 2 763 | 1.62x |
| $G$ | `unordered_set<pair<int, int>>` ~ `reserve` | 1 309 | 3.41x |
| $H$ | `vector<vector<char>> + unordered_set<pair<int, int>>` ~ `reserve` | 388 | 11.51x |
| $I$ | `vector<vector<int>>` | 7 256 | 0.62x |
| $J$ | `vector<vector<char>>` | 1 483 | 3.01x |
| $K$ | `vector<vector<bool>>` | 556 | 8.03x |
| $L$ | `vector<bitset<50'001>>` | 223 | 20.03x |

Janči

## 6. Aaaaaaaaa! (max. 12 b za popis, 8 b za program)

### Ako neriešiť úlohu

Pri prvom pohľade by sa mohlo zdať, že úlohu pôjde vyriešiť nejakým jednoduchým prehľadávaním do šírky, pri ktorom skúsime vždy najskôr spraviť krok doprava a doľava a potom kopať. Už z príkladov je však jasné, že to tak jednoduché nebude. Občas totiž môže byť najefektívnejšia možnosť (alebo jediná možnosť) vykopať v jednom riadku niekoľko súvislých kociek, aby sme aj v riadku pod ním mohli vykopať niekoľko ďalších kociek (pričom musíme stáť na predtým vykopaných políčkach), a tak ďalej – ako môžeme vidieť na treťom príklade.

Nestačí nám teda vedieť, ako sme sa na nejaké políčko dostali – dôležité je aj to, ktoré políčka sme cestou vykopali.

### V jaskyni sa hodí dynamit

Takýto typ zadaní vedie k myšlienke dynamického programovania, ktorého princípom je, že riešime čiastkové problémy (napríklad ako najefektívnejšie sa vieme dostať na dané políčko) a tieto čiastkové riešenia využijeme na efektívne vyriešenie ďalších podproblémov, až po riešenie pôvodného problému (ako najefektívnejšie sa vieme dostať na políčko v spodnom riadku).

Kľúčovú úlohu zohráva takzvaný stavový priestor – definujeme si stav ako nejakú $n$-ticu podstatných informácií, pre každý možný stav potom vyriešime podproblém pomocou iných stavov. Množinu všetkých stavov nazývame práve stavový priestor. Priestor preto, že si ho zvykneme predstavovať $n$-rozmerný, s "osami", ktoré tvoria jednotlivé prvky $n$-tice.

Implementujeme ho často ako $n$-rozmerné pole – tento prístup funguje, pokiaľ vieme, aké má celý priestor hranice v každej súradnici. Druhá možnosť je pamätať si dosiahnuté stavy v slovníku, tá sa zíde najmä vtedy, keď očakávame, že navštívime omnoho menej stavov, než všetky možné, a preto sa neoplatí vyrábať si veľké pole.

V naivnom nefunkčnom riešení by stav vyzeral ako dvojica súradníc políčka, na ktorom sa práve nachádzame. Stavový priestor by preto tvoril dvojrozmerné pole veľkosti šírka krát výška (teda rovnakého tvaru, ako jaskyňa). Pre každý stav by sme si pamätali najmenší počet kociek, ktoré bolo treba vykopať, aby sme sa do neho dostali.

Stavový priestor je užitočný koncept aj bez dynamického programovania – môžeme ho napríklad prehľadávať spomínaným BFS alebo použiť Djikstru. Pokiaľ však dokážeme vymyslieť takú reprezentáciu, že stavy ukladáme v poli, riešime jeden po druhom a máme istotu, že aktuálny stav závisí len na "predošlých" a nie na "nasledujúcich", je najjednoduchšie proste prejsť postupne všetky stavy, a to sa nazýva Dynamické programovanie (DP).

### Čo bude náš stav

Opäť si pre každý stav budeme pamätať najmenší počet kociek, ktoré treba vykopať, aby sme sa do neho dostali.

Ako sme zistili skôr, definovať stav len pomocou súradníc nestačí, treba vedieť aj to, ako sme kopali, kým sme sa na dané políčko dostali. Mohli by sme teda skúsiť stav reprezentovať ako `(súradnica Y, súradnica X, voľné políčka v jaskyni)`. Keďže však jaskyňa môže mať až $V \times S$ políčok, mohlo by existovať $V \times S \times 2^{V \times S}$ stavov – rozhodne by sme teda nedokázali všetky reprezentovať v poli a časovo aj pamäťovo by bolo prehľadávanie priestoru veľmi neefektívne.

V takejto reprezentácií si toho pamätáme zbytočne veľa – vôbec nás totiž nemusí zaujímať, aké políčka sa nachádzajú v iných riadkoch, než je aktuálny riadok a riadok pod ním. Keďže sa nevieme pohybovať smerom dohora, všetky políčka v nižších riadkoch vyzerajú presne tak, ako pôvodná jaskyňa (zatiaľ sme ich nemohli vykopať) a všetky políčka vyššie sme už prešli (a poznáme najefektívnejší spôsob, ako sa cez ne dostať do aktuálneho stavu).

Ďalej si vieme všimnúť, že pokiaľ začneme v jednom riadku kopať, môžeme pokračovať v kopaní už len do jednej strany (cúvať) – nevieme už vykopané políčko preskočiť, môžeme len cezeň spadnúť o riadok nižšie (a vtedy v danom riadku kopať prestaneme).

Takisto nemá zmysel kopať prerušovane, pretože všetky políčka uvoľnené pred preskočenou kockou (prerušením kopania), by sme vykopali zbytočne – v ďalšom riadku sa cez nevykopanú kocku nevieme dostať a v neskorších riadkoch už budú tieto políčka príliš vysoko, aby čokoľvek ovplyvnili. Má teda zmysel kopať len jeden súvislý úsek.

Vďaka tomu si stav vieme definovať ako štvoricu `(súradnica Y, súradnica X, začiatok vykopaného úseku v aktuálnom riadku, koniec vykopaného úseku)`. Tento stavový priestor vieme ešte zmenšiť, keď si uvedomíme, že:

1) každý úsek prestaneme kopať stojac na políčku, ktoré je na jednom jeho konci (teda stačí riešiť kopanie úsekov končiacich tesne pred políčkom, kde stojíme)
2) a zároveň nás na každom políčku vo vykopanom úseku zaujíma počet vykopaných kociek len v jednom smere (totiž aj tak budeme z aktuálneho políčka kopať len jedným smerom, pretože podľa 1. chceme, aby koniec kopaného úseku bol vedľa políčka, kde stojíme).

Dostaneme tak stavový priestor `(súradnica Y, súradnica X, dĺžka vykopaného úseku, smer kopania)` s rozmermi $V \times S^2 \times 2$ (posledný prvok štvorice môže nadobúdať len dve hodnoty, preto sa do asymptotickej zložitosti nepočíta).

### Ako môžeme prechádzať medzi stavmi

Tu by sme mohli zvoliť jednu z dvoch základných možností – pre každý stav vyriešiť buď ako najlepšie sa dá do neho dostať (prechádzaním predošlých stavov), alebo ako najlepšie sa dá dostať do iných stavov z neho (prechádzaním nasledujúcich stavov a prepisovaním ich hodnôt lepšími).

Keďže dosť stavov pravdepodobne bude nedosiahnuteľných, oplatí sa zvoliť si druhú možnosť, ktorá nám dovolí pri navštívení nedosiahnuteľného stavu (čo spoznáme napríklad tak, že si na začiatku do všetkých stavov priradíme väčšiu hodnotu, než kedy vieme dosiahnuť, napríklad $V \times S$) tento stav jednoducho preskočiť.

Teraz už treba len vymyslieť prechody medzi stavmi. Stav, v ktorom sa nachádzame nám vraví, že sme na súradniciach $X, Y$ a smerom (doprava alebo doľava) je určite voľných (predtým vykopaných) $Z$ políčok (voľných políčok tam samozrejme môže byť viac, než $Z$, a zároveň nemáme garantované ani to, že tých $Z$ políčok sa dá dosiahnuť, pretože v riadku nižšie môžu byť diery).

Pohyb bude teda vyzerať tak, že sa skúsime pohnúť doľava aj doprava tak ďaleko, kým to pôjde – teda kým bude na našom riadku vykopané alebo voľné políčko a v nižšom riadku nebude voľné políčko (cez ktoré by sme spadli). V každom navštívenom stave sa pokúsime aktualizovať jeho hodnotu za hodnotu aktuálneho stavu, ak je menšia (našli sme cestu do aktuálneho stavu, ktorá celkovo potrebuje menej kopaní). Pokiaľ sme sa pohli rovnakým smerom, ako vraví aktuálny stav, počet určite voľných políčok ($Z$) sa zníži o počet prejdených polí, inak sa o toľko zvýši. Zároveň pre každé dosiahnuté políčko skúsime aktualizovať aj stav s opačným smerom, so zodpovedajúcimi počtami voľných políčok (zvýšenými / znížených o koľko sme zatiaľ počas pohybu prešli). Keď narazíme na políčko, cez ktoré by sme spadli nižšie, aktualizujeme aj dopadové políčko a skončíme pohyb.

Kopanie je veľmi podobné, skúsime ísť doprava a doľava (rovnako ako pri pohybe), ale budeme aktualizovať políčka o riadok nižšie – ako keby sme prišli k nejakému vzdialenenmu políčku, vykopali od neho smerom späť k aktuálnemu políčku všetky políčka (vrátane samotného vzdialeného políčka) a potom zoskočili na prvé vykopané políčko, teda sa pohli o jedno doprava alebo doľava.

### Ako vyriešime padanie

Hoci sa táto časť môže zdať zložitá (aj keď možno oproti zbytku úlohy ani nie), vieme padanie jednoducho vyriešiť pomocou predpočítania si, kam by sme dopadli, keby sme začali padať z každého políčka.

Pôjdeme od spodku jaskyne po vrch a vždy, keď nájdeme políčko, pod ktorým je kocka, prejdeme postupne všetky voľné políčka nad ním až po prvé políčko s kockou (vrátane, pretože tá by mohla byť v budúcnosti vykopaná), a poznačíme si na ne, o koľko políčok nižšie spadneme, ak sa na nich ocitneme. Môžete si rozmyslieť, že pre každé políčko takto máme len konštantný počet operácií. Dôležité je, že nikdy nenastane situácia, že by sme spadli cez viac než jedno vykopané políčko naraz, keďže sa nevieme hýbať smerom dohora.

Ak potom niekedy zistíme, že by pohyb na voľné políčko alebo pohyb po kopaní spôsobil príliš vysoký pád, jednoducho cieľový stav neaktualizujeme.

### Akú to má zložitosť

Pamäťová zložitosť závisí na uložení si jaskyne, výšok pádu (obe zaberajú $O(V \times S)$ miesta) a najmä samotného stavového priestoru (ten zaberá $O(V \times S^2)$ miesta, pretože počet vykopaných políčok v každom riadku môže byť až $O(S)$). Celková pamäťová zložitosť je teda $O(V \times S^2)$.

Časová zložitosť závisí na počte stavov ($O(V \times S^2)$) a zložitosti prechodov medzi jednotlivými stavmi. V tomto vzoráku sú opísané prechody, ktoré majú zložitosť (pohyb aj kopanie) $O(S)$, keďže z každého stavu sa pokúšame aktualizovať $O(S)$ ďalších. Celková časová zložitosť je teda $O(V \times S^3)$.

Na záver prezradíme, že síce nejde stavový priestor zmenšiť, ale je možné trochu ho zmeniť a vymyslieť zložitejšie prechody, ktoré majú konštantnú zložitosť (čím by sa dala dosiahnuť celková časová zložitosť $O(V \times S^2)$). Za odovzdanie takéhoto riešenia v popise bolo možné získať bonusové body.

Jakub Konc

## 7. Nečakaný výlet

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

### Bruteforce

Prvý prístup, ktorý by nám mohol napadnúť je prechádzať postupne všetkými dňami a v každom sa pozrieť, koľko ľudí má vtedy voľno. Ak je ich $x$, potom vtedy vieme vybrať $\binom{x}{k}$ skupín. Keď však toto naimplmenetujeme, dostaneme od testovača verdikt nesprávna odpoveď. Prečo? Ak má totiž nejaká skupina prienik viac než jeden deň, započítame ju viackrát, čo nechceme.

Prvý spôsob ktorý by nám mohol napadnúť na riešenie tohoto problému je si už zarátané skupiny do set-u a opäť ich už tak nerátať. Toto by určite fungovalo, ale akú by to malo časovú zložitosť? Označme si $L = \max b_i$. Potom v každom dni môžeme mať až $n$ ľudí s voľnom a tak až $\binom{n}{k}$ skupín. Každú z nich musíme vygenerovať a vyskúšať, či sa nachádza v set-e. A dní je $L$, teda celková časová zložitosť je $O(Lk\binom{n}{k})$. To je jedna z najhnusnejších zložitostí, čo som kedy videl a samozrejme za ňu náš program nedostane žiadne body.

### Dačo rozumnejšie

Tak teraz by to asi chcelo nad úlohou sa skutočne zamyslieť a nie len do nej búchať dátovými štruktúrami. Chceli by sme teda vymyslieť spôsob, ako každú skupinu priradiť jedinému dňu. Asi najprirodzenejší a najjednoduchší spôsob je si jednoducho vybrať prvý deň jej prieniku.

Tento deň bude ale jednoducho posledný zo začiatkov intervalov voľna. To znamená, že túto skupinu zarátame keď narazíme na posledného.

Nový algoritmus je teda takýto: Prechádzame si postupne dňami a udržiavame si počet momentálne aktívnych intervalov (teda ľudí, ktorí momentálne majú voľno). Vždy keď narazíme na začiatok nového intervalu, zvýšime odpoveď o $\binom{x}{k-1}$, kde je $x$ je počet aktuálne platných intervalov (mimo tohoto nového) – skupinu vieme vytvoriť z tohoto nového človeka a ľubovoľných $k-1$, ktorých sme už videli.

Takéto riešenie bude mať časovú zložitosť $O(n+L)$ – každého človeka spracujeme len dvakrát pri zmene počtu aktívnych intervalov a raz pri prepočítavaní odpovede. Pamäťová bude tiež $O(n+L)$, keďže si potrebujeme pre každý deň predpočítať, ktoré intervaly v ňom začínajú a končia.

### Odbočka: kombinačné čísla

V odhade časovej zložitosti sme sa tvárili, že kombinačné čísla vieme počítať v konštantnom čase. Ako však na to? Použijeme klasický vzorček $\binom{a}{b} = \frac{a!}{b!(a-b)!}$. Prvým krokom bude predpočítať si faktoriály čísel do $n$ (evidentne $a, (a-b) \leq n$). Avšak delenie vo zvyškovej triede tiež nie je triviálne. Hľadanie modulárneho inverzu https://www.ksp.sk/kucharka/modularna_aritmetika/. V tomto kontexte je síce korektné povedať, že to je konštanta, keďže modulo je konštantné, avšak ide to aj lepšie.

Všimnime si, že jediné čísla, ktorými delíme sú faktoriály, ktoré máme predpočítané. A keďže je ich rozumne málo – $O(n)$ – môžeme si k nim predpočítať aj inverzy v $O(n \log \mathrm{MOD})$ a tak naozaj vedieť kombinačné čísla počítať v konštantnom čase.

### Optimálne riešenie

Od predošlého riešenia nám ku vzorovému chýba iba kúsok. Všimnime si, že v dňoch kde nezačína ani nekončí žiadny interval nič nerobíme a tak nám nič nepokazí, ak ich proste preskočíme. Tomuto sa hovorí zametanie – zoberieme si udalosti, ktoré nás zaujímajú, zoradíme si ich a prechádzame len nimi.

Postup je teda jednoduchý – zoberieme si všetky začiatky a konce intervalov, zoradíme si ich podľa času a spracovávame ich v takomto poradím.

Takéto riešenie má časovú zložitosť $O(n \log n)$ – udalosti potrebujeme zoradiť podľa času. Pamäťová zložitosť je $O(n)$.

Johanko

## 8. Divné diery

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

Táto úloha mala veľmi nechutne dlhé zadanie. Ak ste sa však nenechali odradiť ani dĺžkou zadania ani číslom úlohy, riešenie na 4 body nebolo vôbec ťažké. Pred tým ako sa bezhlavo pustíme do programovania, je užitočné si ťažkú úlohu čo najviac zľahčiť.

### Základné pozorovania

Ako prvé sa zamyslime, ako si zjednodušiť úvahy s dierami.

Môžeme si uvedomiť, že ak sme najskorej chodili peši a potom použili nejakú dieru, mohli sme rovno použiť tú istú dieru a opäť, mali by sme aspoň rovnako dobré riešenie. Teda diery sa oplatí používať len na začiatku, keď sme vo vrchole 0.

Druhé pozorovanie je, že vždy sa nám oplatí použiť najviac jednu dieru. Ak by sme použili nejaké dve, najprv $A$ a potom $B$, mohli sme rovno použiť $B$, a mali by sme aspoň také dobré riešenie ako doteraz. To vyplýva z toho, že nezáleží kde sa nachádzame, diera nás k sebe premiestni vždy rovnako rýchlo.

Posledné užitočné aj keď nie nutné je pridať si do vrcholu 0 fiktívnu dieru s časom 0 a cenou 0. To bude zodpovedať možnosti, kde nepoužijeme žiadnu dieru. Zvyšok vzoráku rátame s tým, že sme tento trik spravili.

Naraz sa nám úloha značne zjednodušila: Máme strom, v niektorých vrcholoch sú diery. Ak by sme chceli v $i$-tej otázke použiť $j$-tu dieru, vieme sa z nej pohnúť o vzdialenosť $b_i - w_j$. Chceme teda vybrať dieru z najnižšou cenou, z ktorej sa vieme dostať do vrcholu $a_i$.

Toto nám stačí na pohodlný bruteforce.

### Bruteforce

Nasledovné riešenie mohlo získať 4 body:

Z každej diery spustíme prehľadávanie a tým pre každý vrchol spočítame vzdialenosti od všetkých dier. Potom pri query $i$ nájdeme v zozname pre vrchol $a_i$ všetky diery $j$ vo vzdialenosti najviac $b_i - w_j$ a vyberieme z nich dieru s najlepšiou cenou.

Celková časová zložitosť tohoto riešenia bude $O((n+q) \cdot m)$, čo stačilo na získanie 4 bodov.

### Strom je plytký!

V tretej sade má strom hĺbku najviac 100. Skúsme sa zamyslieť, ako nám to pomôže pri riešení úlohy.

Zoberme si $i$-tu query a predstavme si, že by sme na ňu použili dieru $j$. Potom cesta z diery do $a_i$ musí prejsť nejakým predkom $a_i$ (každý vrchol je sám sebe predkom). Zoberme si tak každého predka $a_i$ a diery ktoré sú v jeho podstrome. Všimnime si, že pre každú z týchto dier vieme vypočítať hodnotu "koľko času by nám trvalo sa dostať do $a_i$ pomocou diery $j$ ak by sme išli cez predka $u$" ako súčet $w_i$ + (dĺžka cesty z $u_i$ do diery $j$) + (dĺžka cesty z $u_i$ do $a_i$). Počítanie týchto hodnôt samo o sebe nie je dobrý nápad, lebo to je pomalé, ale uvidíme, že to povedie k lepšiemu riešeniu.

Túto hodnotu vieme vypočítať ako súčet dvoch hodnôt, jedna závisí od diery $j$ a $u$, druhá od $a_i$ a $u$, vidíme, že voľba $j$ ovplyvní len prvú hodnotu a voľba $i$ len druhu, teda ich môžme spočítať pre všetky $j$ a $u$ zvlášť a pre všetky $i$ a $u$ zvlášť. Toto počítanie už nebude pomalé, lebo každý vrchol má najviac 100 predkov, teda sa vyskytne v 100 výpočtoch.

Pre každý vrchol spočítame jeho vzdialenosť od dier v jeho podstrome. Ak by sme mali túto informáciu spočítanú, ľahko nájdeme odpoveď na otázku: *ak sme vo vrchole $u$ a máme ešte $t$ času, aká je najlacnejšia diera v podstrome vrcholu $u$, do ktorej sa vieme za čas $t$ dostať?*

Ukážeme si ako:

V každom vrchole $u$ si budeme pamatať všetky diery v jeho podstrome, ale namiesto ich hodnoty $w$ k nej ešte pripočítame vzdialenosť do vrcholu $v$ – vrchol kde sa diera nachádza. Tieto hodnoty označíme $w'$. Teraz platí:

Ak sa pýtame na otázku pre vrchol $u$ a čas $t$, vyberieme tie diery, ktoré majú $w' \leq t$ a na tie sa budeme vedieť dostať. (Ak hovorím, že sa na ne vieme dostať, myslím že sa vieme z vrcholu 0 premiestniť do nich a potom peši prejsť do $u$, to celé v čase najviac $t$.)

Teraz si diery vo vrchole $u$ usporiadame podľa hodnoty $w'$. Potom nás zaujíma len prvých niekoľko dier s dosť malou $w'$. Z nich vieme vybrať tú s najlepšou cenou napríklad pomocou prefixových miním, keďže sa vždy pýtame na prefix a hodnoty sa nemenia.

Ak sa teraz budeme pýtať postupne na predkov vrcholu $a_i$, pričom vhodne upravíme čas, ktorý nám ešte zostáva, a hľadať pomocou tejto otázky najlepšie diery, dostaneme sa zjavne k riešeniu.

Celé predpočitanie teda bude vyzerať takto:

1. Každú dieru $i$ zaznačíme aj do jej predkov, pričom k $w_i$ pripočítame vždy dĺžku cesty do toho konkrétneho predka, tým získame hodnoty $w'$.
2. Pre každý vrchol diery usporiadame podľa $w'$.
3. Pre každý vrchol spočítame prefixové minimá podľa hodnôt $c$.

Každú dieru zaznačujeme do jej predkov a potom týchto $O(mH)$ záznamov utriedime, teda dostávame zložitosť $O(mH \log m)$.

Odpoveď na otázku: *"som vo vrchole $u$ a mám ešte $t$ času"* potom získame tak, že v zozname pre vrchol $u$ binárne vyhľadáme poslednú dieru, ktorú vieme použiť a následne sa spýtame na prefixové minimum.

Celé riešenie jednej query teda bude:

1. Začíname vo vrchole $u = a_i$ s časom $t = b_i$.
2. Spýtame sa na otázku *"som vo vrchole u a mám t času"* a upravíme výsledok.
3. Ak nie sme v koreni stromu priradíme $t = t - $`čas na prejdenie do otca u`, $u =$ `otec u`. Ideme znova na krok 2.
4. Zohľadnili sme všetky možnosti, teda vieme výsledok.

Pre každú query H krát vyhľadáme odpoveď, čo dáva zložitosť $O(qH \log m)$.
Celé riešenie má časovú zložitosť $O(n + mH \log m + qH \log m)$. Pamäťová zložitosť bude $O(mH + n)$.

### Na čo nám to bolo?

Prečo sme mali dlhočizný vzorák o nejakom riešení, ktoré ani neprešlo na plný počet bodov? Odpoveď znie, že často sú podúlohy nastavené tak, aby pomohli k riešeniu celej úlohy. A táto úloha nie je výnimkou.

### Nechcem spoilovať, tak tento nadpis nenapíšem

Kebyže máme strom plytký, úlohu ľahko vyriešime. Všeobecne, ak je strom hlboký a chceme, aby bol plytký, správnym nástrojom sú centroidy. Ak ste nikdy nepočuli o centroidoch a centroidovej dekompozícií, môžete si o tom prečítať tu[3].

V rýchlosti sa to da opisať takto: Centroidovou dekompozíciou vhodne usporiadame vrcholy stromu tak, že sa úplne nezachová samotná štruktúra zadaného stromu, ale vznikne nový, úplne iný strom hĺbky najviac $\log n$ - čo je dobré znamenie, lebo pre takto hlboké stromy už poznáme riešenie.

Tento strom však bude mať jednu peknú vlastnosť. Keď sa pozrieme na jeho ľubovolný vrchol v centroidovom a pôvodnom strome, a pozrieme sa na podgrafy, ktoré vzniknú keď ho odstránime, budú mať tieto podgrafy rovnaké vrcholy pre centroidový aj pôvodný strom.

Toto sa ukáže ako veľmi užitočná vlastnosť, ak sa pýtame na cesty v strome, lebo pre vrchol na ceste v pôvodnom aj centroidovom strome platí, že ak ho odstránime, bude začiatočný a koncový vrchol cesty v rôznych podstromoch – z formulácie vidíme, že je práve to čo centroidové stromy poskytujú.

Keď si teraz porovnáme, čo sme robili v predošlom odseku za 6 bodov, jediná podstatná vlastnosť, ktorú sme potrebovali, bola, že aspoň v jednom vrchole, ktorý je na ceste medzi nejakou dierou a query, zohľadníme túto dieru. Stále je to tá istá vlastnosť, čo som prakticky napísal 3x za sebou, aby to bolo vidieť, a čo sa v centroidových stromoch zachováva. (toto celé ešte nižšie dokážeme)

Otázka znie, či si aj pre takýto strom budeme vedieť predpočítať podobné hodnoty ako minule. Bystrému čitateľovi hneď dojde, že to je len rečnícka otázka, a že to ide :p

Začneme rovnako ako predtým, každú dieru zaznačíme aj do jej predkov v centroidovom strome. Problém bude, že novú hodnotu $w'$ diery nebudeme tak ľahko vedieť spočítať. Nebudeme sa môcť vždy iba pozrieť na cenu hrany do otca a túto cenu pripočítať. Na chvíľu ale predpokladajme že máme funkciu $tree\_dist(a, b)$, ktorá vráti vzdialenosť vrcholov $a$ a $b$ v strome zo vstupu.

Pomocou tejto funkcie už ľahko dokončíme predpočítanie tak ako sme to robili minule, s výnimkou, že hodnota, ktorú si uložíme vo vrchole $u$, ak sme pridali dieru do vrcholu $v$ bude $w + tree\_dist(u, v)$.

Ak by sme aj druhú časť riešenia upravili dostávame to isté riešenie, len s drobnými zmenami:

1. Prvý krok je rovnaký.
2. Spýtame sa otázku na vrchol $u$ s časom $b_i - tree\_dist(a_i, u)$ – uvedomme si, že presne toto robíme aj v predošlej verzií.
3. Nastavíme $u$ na jeho otca v centroidovom strome.

Vidíme, že sa naozaj oplatilo robiť celé to predošlé riešenie, keďže toto nie je o moc iné – teda samozrejme za predpokladu, že to čo sme tu teraz napísali a (vyššie som povedal, že to platí) naozaj platí.

### Ukážeme, že to naozaj funguje a implementácia

Môžeme si všimnúť, že v tomto riešení sa nám dosť často stane, že keď v nejakom vrchole $v$ zohľadňujeme nejakú dieru $d$, museli by sme, kebyže si to nakreslíme v pôvodnom strome, viackrát prejsť po tej istej hrane-/ceste. (formálne vrchol $v$ je predok $lca(v, d)$) Teda hodnota $w'$ bude v takýchto prípadoch moc veľká. Nikdy sa nám našťastie ale nestane, že by sme mali pre nejakú dieru hodnotu $w'$ moc malú, keďže vždy pripočítame $tree\_dist$.

[3] https://discourse.opengenus.org/t/centroid-decomposition-of-tree/3409

Teda určite nájdeme výsledok aspoň tak veľký ako má byť. To je fajn. Teraz nám stačí ukázať, že aspoň pri jednom vrchole $v$, ktorý budeme pri query kontrolovať budeme mať cestu do diery $d$, kde ani jednu hranu nepoužijeme viac krát. Inými slovami, že nájdeme výsledok taký malý, resp. práve taký, aký má byť.

Nie je ťažké si uvedomiť, že správne vzdialenosti bez opakovaných hrán dostaneme práve pre vrcholy $v$, ktoré sú v centroidovom strome predkami $d$ aj $a_i$ (alebo sú to $d$ alebo $a_i$) a pre ktoré $a_i$ a $d$ ležia v rozdielných centroidových podstromoch, a teda aj v rozdielných podstromoch v pôvodnom strome. Chceme ukázať, že existuje aspoň jeden vrchol, ktorý spĺňa túto vlastnosť. Na to aby sme to ukázali, zíde sa nám vedieť ako centroidový strom vytvárame – z tohoto postupu to už priamo vyplýva.

### Centroidová dekompozícia

Pre nejaký strom vieme nájsť centroid jedným prehľadávaním. Ak pre vrchol platí, že veľkosti podstromov vo všetkých jeho susedoch je $\leq$ polovici vrcholov v strome, je tento vrchol centroid. Inak je centroid v podstrome, ktorý má najväčšiu veľkosť a na ten sa rekurzívne zavoláme. Každý strom má centroid a týmto postupom ho nájdeme. (Dôkaz tohoto algoritmu neuvádzame.)

Keď sme našli centroid, musíme nájsť centroidy v podstromoch jeho synov a na to rekurzívne opäť použijeme vyššie popísaný algoritmus.

Pauza na pitie.

Pokračujme v dôkaze. Pozrime sa na vrcholy na ceste medzi $a_i$ a $d$. Ak za centroid zvolíme ľubovoľný iný vrchol a graf rozdelíme na jeho podstromy, celá táto cesta bude stále v jednom podstrome. Čiže z hľadiska cesty sa nič zaujímavé nestalo.

A ak vyberieme za centroid jeden z vrcholov na ceste, cesta sa rozdelí do jeho podstromov, a to tak, že vrchol $a_i$ bude v podstrome jedného jeho syna a vrchol s dierou $d$ bude v nejakom inom. Teda minimálne prvý vrchol z cesty, ktorý v algoritme vyššie vyberieme za centroid bude spĺňať požadovanú vlastnosť, a teda pri ňom dostaneme správne vzdialenosti od $a_i$ aj $d$.

Teraz vieme, že máme správne riešenie, stačí nám implementovať posledný detail.

### Funkcia *tree_dist*

Na implementáciu tejto funkcie vieme napríklad použiť binary lifting, alebo teda klasický LCA algoritmus. Ak ste o tom ešte nikdy nepočuli, môžete si viac prečítať v tomto článku[4], kvôli dĺžke vzoráku to tu už neuvádzame.

Ak sa nám ale nechce kódiť binary lifting, ide to aj krajšie! Celkovú časovú zložitosť to síce nezlepší, ale v praxi to program môže niekoľkonásobne zrýchliť.

Môžme si uvedomiť, že vždy sa pýtame len na vzdialenosti medzi vrcholom a nejakým jeho centroidovým predkom. Tých je však len $O(\log n)$ preto môžeme mať zapamätanú pre každý vrchol vzdialenosť do jeho centroidových predkov. Jeden spôsob ako takúto informáciu získať, je priamo pri dekompozícií si pre novonájdený centroid spočítať vzdialenosť do všetkých vrcholov v jeho podstrome. Zložitosť na predpočítanie zostane $O(n \log n)$, ale na query sa zmenšila na $O(1)$, lebo sa stačí len hodnoty pamätať v poli. Detaily implementácie prenecháme na čitateľovi.

### Časová a pamäťová zložitosť

Centroidovú dekompozíciu zvládneme v čase $O(n \log n)$, rovnako rýchlo bude trvať aj predpočítanie vzdialeností do centroidových predkov.

Predpočítanie dier nám zaberie $O(m \log m \log n)$ času – najskôr diery zapisujeme do predkov, čo trvá $O(m \log n)$, teda to zanedbáme, potom to celé utriedime a spravíme prefixové súčty – to je výsledná zložitosť.

Samotné odpovede na queries budú trvať $O(q \log n \log m)$ – pozrieme $O(\log n)$ predkov a binárne vyhľadáme, ktoré diery sú použiteľné.

Celá časová zložitosť bude $O(m \log m \log n + q \log n \log m + n \log n)$.

Pamäťová bude $O(n \log n + m \log n)$, lebo si pamätáme vzdialenosti z vrcholov do centroidových predkov a každú dieru v jej centroidových predkoch.

Postup, ktorý sme si tu ukázali má tú výhodu, že je online – teda otázky spracuváva postupne, každú dostatočne rýchlo. Na precvičenie môžeš skúsiť vymyslieť riešenie na rovnakú úlohu s tým, že vrámci query môže pribudnúť nová diera. Hint – stačí len vhodne upraviť toto riešenie.

[4] https://codeforces.com/blog/entry/100826