Korešpondenčný seminár z programovania
XLIII. ročník, 2025/26
FMFI UK,
Mlynská Dolina, 842 48 Bratislava

# Vzorové riešenia 1. kola zimnej časti

Autor

## 1. Kobry sú plaché

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

Na začiatok si musíme uvedomiť, že had pozostáva z hlavičky `~O`, tela tvoreného znakom `=` a chvostíka `>`. Teda medzi hlavičkou a chvostíkom sa musia nachádzať iba znaky `=`, inak to nie je had. Taktiež vieme, že všetky hady sú orientované rovnakým smerom, a to takým, že hlavička je vždy naľavo od chvostíka.

Teda nám stačí prejsť reťazec z ľava do prava, a vždy, keď nájdeme hlavičku hada, pamätať si, že sme v tele hada a počítať jeho dĺžku, až kým nenarazíme na chvostík alebo iný znak. Ak nájdeme chvostík, stačí nám pripočítať dĺžku daného hada k celkovému súčtu dĺžok hadov, ak nájdeme iný znak, pokračujeme v hľadaní hlavičky.

### Časová zložitosť

Časová zložitosť je $O(n)$, lebo prechádzame celým reťazcom iba raz.

### Pamäťová zložitosť

Pamäťová zložitosť je $O(1)$, lebo si program pamätá iba niekoľko premenných - súčet dĺžok hadov a premennú, ktorá kontroluje, či sa práve nachádzame v hadovi alebo nie. Vzorový python kód má pamäťovú zložitosť $O(n)$, pretože načítava celý rad do pamäte. Vzorový C++ kód má pamäťovú zložitosť $O(1)$, pretože načítava iba jeden znak naraz. Na vyriešenie úlohy za plný počet bodov stačí implementácia s pamäťovou zložitosťou $O(n)$.

Soňa

## 2. Ešte čakajte

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

### Priamočiare riešenie

Našou úlohou je čo najmenším počtom zmien upraviť reťazec tak, aby mal tvar najprv súvislého bloku núl, po ktorom nasleduje súvislý blok jednotiek. Znamená to, že reťazec musí vyzerať ako napríklad `0000111`.

Riešenie si môžeme predstaviť tak, že si vyberieme nejaké miesto v reťazci, nazvime ho hranica, kde má byť prechod z núl na jednotky. Všetky znaky naľavo od hranice by mali byť nuly a napravo od hranice jednotky. Ak tam nie sú, musíme ich zmeniť. Naším cieľom je nájsť takú hranicu, kde bude potrebné zmeniť čo najmenej znakov.

Počet zmien pre jednu konkrétnu hranicu vypočítame nasledovne: spočítame, koľko jednotiek je naľavo od hranice (tie treba zmeniť na nuly), a koľko núl je napravo od hranice (tie musíme zmeniť na jednotky). Súčet týchto dvoch čísel nám povie, koľko zmien musíme spraviť, ak by bola hranica práve na tomto mieste.

Ak by sme pre každú hranicu opakovane rátali počet jednotiek naľavo a počet núl napravo, trvalo by to príliš dlho. Pre každú hranicu by sme prešli celý reťazec, teda celkovo by sme $n$-krát prešli reťazec dĺžky $n$. Čo by viedlo k celkovej časovej zložitosti $O(n^2)$. To ale vieme urobiť oveľa efektívnejšie.

### Optimálne riešenie

Na začiatku si spočítame celkový počet jednotiek v reťazci. Potom prechádzame reťazec zľava doprava a sledujeme, koľko jednotiek sme už stretli. Z toho vieme koľko jednotiek je naľavo od aktuálnej hranice - teda koľko znakov musíme zmeniť naľavo od hranice. Taktiež si z toho vieme dopočítať aj koľko núl je napravo. Keďže vieme koľko jednotiek je celkovo a koľko z nich sme prešli, vieme odčítaním týchto dvoch čísel zistiť koľko jednotiek zostáva. Rovnako vieme koľko znakov zostáva do konca. Ak odčítame počet zvyšných jednotiek od počtu zvyšných znakov, zistíme počet zvyšných núl.

Pre každé možné umiestnenie hranice tak vieme v konštantnom čase vypočítať, koľko zmien by sme museli spraviť, a zároveň si priebežne pamätáme najmenší takýto počet. Po prejdení celého reťazca tak nájdeme najmenší počet potrebných zmien.

Toto riešenie má časovú aj pamäťovú zložitosť $O(n)$, keďže reťazec prechádzame len raz a pamätáme si len reťazec a niekoľko pomocných premenných.

Lauri

## 3. Dole kopcom

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

Naším zadaním bolo nájsť miesta, na ktorých mohol byť najvyšší kopec.

### Bruteforce

Predstavme si jednoduchší prípad, že tam otázniky nie sú. V tomto prípade stačí prejsť celú trasu a vypočítať si výšky kopcov. Taktiež si vyrátame maximálnu výšku počas celej trasy. Nakoniec vypíšeme miesta, na ktorých sa nachádzali najvyššie kopce, teda pozície s maximálnou hodnotou a koľko ich bolo.

Pridajme do úlohy otázniky. Za každý `?` vieme doplniť `+` alebo `-`. Skúsme úlohu vyriešiť hrubou silou. V takomto riešení chceme vygenerovať všetky možnosti. Tých je $2^q$, kde $q$ je počet otáznikov.

Pre každú z možností použijeme predchádzajúce riešenie. Tu nastáva problém, lebo môžeme nejakú pozíciu zarátať aj viackrát, v dvoch rôznych možnostiach bol najvyšší kopec na rovnakom mieste. Preto si chceme pamätať informáciu, či už daný kopec maximom bol.

Pozrime sa na časovú zložitosť tohto riešenia. Vygenerujeme si všetkých $2^q$ možností. Pre každú v lineárnom čase spočítame polohy všetkých maxím. Na záver prejdeme všetky polohy, čo už časovú zložitosť neovplyvní. Vo výsledku budeme mať exponenciálnu časovú zložitosť $O(n \cdot 2^q)$.

Pamäťová zložitosť je $O(n)$, lebo si musíme pamätať vstup, ktorý následne budeme upravovať, a finálne pozície najvyšších kopcov. Ale počas jedného prechodu si už nič väčšie pamätať nemusíme, iba priebežné maximum.

### Niečo lepšie

Skúsme sa na to pozrieť z inej strany. Na to aby bol kopec na $i$-tom mieste najvyšší, musí platiť, že pred príchodom na tento kopec sme sa snažili čo najviac stúpať, a po ňom čo najviac klesať. Preto chceme za všetky otázniky pred daným kopcom pridávať `+` a za všetky otázniky po ňom pridávať `-`.

Prejdime teda všetky pozície a pre každú overme, či by na danom mieste mohlo byť maximum.

Pre každý kopec musíme prejsť celé pole dĺžky $n$, takže časová zložitosť je $O(n^2)$.

Pamäťová zložitosť zostáva $O(n)$, pretože si musíme pamätať počiatočné informácie, ktoré budeme postupne prepočítavať.

### Optimálne riešenie

Skúsme sa zamyslieť nad tým, aké výpočty robíme opakovane v predchádzajúcom riešení. Pre každú pozíciu zisťujeme, či sa na nej nevie nachádzať najvyšší kopec. Teraz môžeme porozmýšľať, či pri rátaní $(i+1)$-vej pozície nevieme znova využiť niečo z tej predošlej pozície. V predchádzajúcom riešení sme to robili tak, všetko "naokolo" sme sa snažili znížiť.

Pri posune o 1 je teda výsledok takmer rovnaký. Všetky `?` pred $i$-tou pozíciou sme nahradili za `+` a všetky za $(i+1)$-vou pozíciou sme nahradili za `-`. Teda ak by sme si zapamätali predchádzajúce hodnoty stačí ich upraviť iba podľa znaku medzi $i$-tou a $(i+1)$-vou pozíciou.

Toto si vieme pomerne jednoducho predrátať. V jednom poli si zapamätáme výšky na pozíciách ak nahradíme `?` za `+` a v druhom ak ich nahradíme `-`.

Doteraz sme pre každú pozíciu zisťovali, či môže byť maximom celého poľa nasledovne: vždy sme prešli celé pole a zistili, aké výšky môžu jednotlivé pozície nadobudnúť a priebežne si pamätali tú najvyššiu. Tento postup je však zbytočne zdĺhavý a pre rôzne pozície vykonáva rovnaké výpočty. Môžeme ho výrazne zjednodušiť, ak si problém rozdelíme na dve časti: pre každú pozíciu si zapamätáme najväčšiu výšku, ktorú vieme dosiahnuť pred ňou, a najväčšiu výšku, ktorú vieme dosiahnuť za ňou. Na výpočet prvej použijeme predrátané hodnoty, v ktorých sme za `?` dopĺňali `+` a na výpočet druhej tie hodnoty kde sme dopĺňali `-`.

Ak je výška na danej pozícii väčšia ako všetky výšky pred ňou a zároveň väčšia alebo rovná všetkým výškam za ňou, potom môže byť práve táto pozícia najvyšším kopcom.

Aby sme tieto hodnoty vedeli efektívne zistiť, predpočítame si ďalšie dve pomocné polia maxím. Prvé bude prefixové maximum, kde všetky otázniky `?` nahradíme za stúpanie `+` a budeme postupne sledovať maximálnu výšku dosiahnutú zľava doprava. Druhé bude sufixové maximum, kde všetky otázniky `?` nahradíme za klesanie `-` a budeme sledovať maximálnu výšku zprava doľava.

Pomocou týchto dvoch predpočítaných polí vieme pre každú pozíciu v konštantnom čase rozhodnúť, či môže byť najvyšším kopcom.

Časová zložitošť bude lineárna $O(n)$, potrebujeme si zrátať štyri polia a nakoniec tieto polia piatykrát vyhodnotiť. Polia výšok kopcov (1 - vždy ideme dole, 2 - vždy ideme hore), prefixové maximum a suffixové maximum. Nakoniec overíme, ktoré kopce môžu byť tými najvyššími.

Pamäťová zložitosť bude $O(n)$, lebo v pamäti máme uložené 4 polia veľkosti $n$, rovnako veľký vstup a zopár premenných.

Jeden z prípadov, ktoré musíme pri implementácii ošetriť je ak na prvej pozícii (nultej v poli) v suffixovom maxime hodnota menšia alebo rovná 0, znamená to, že prvý kopec môže byť najvyšší. Prečo? Lebo ak všetko za ním iba klesá, tak náš prvý kopec musí byť buď najvyšší, alebo jeden z najvyšších.

## 4. Yoooy, pauza na obed

Andrej
(max. 0 b za popis, 20 b za program)

## 5. Obedujeme koláčiky

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

Graf ktorý vznikne z povôdného ponechaním iba n-1 hrán a zároveň zostane súvislý sa volá kostra. Preferencie kamarátov (teda banánový alebo ríbezlový koláč) budeme volať farba - takže budeme sa rozprávať o farbení hrán a vrcholov.

### Riešenie podúlohy

Začneme so sadami 1 a 3. Tieto sady su velmi ľahké ako na myšlienku tak aj na implementáciu, kedže graf, ktorý dostaneme na vstupe je už rovno kostra. Vôbec sa teda nemusíme zaujímať o to, ktoré hrany použijeme (kedže graf ktorý dostaneme už ich má n-1) a len chceme zistiť, či sú všetky vrcholy nášho grafu šťastné.

Pre každý vrchol prejdem všetky jeho hrany a ak nemá hranu svojej farby tak neexistuje riešenie a rovno to môžem vypísať. Inak program dobehne a ja na konci vypíšem riešenie - vypíšem hrany zo vstupu. Časová zložitosť tohto programu je O(n + m), kedže si musím načítať aj vstup. Pamäťová je taktiež O(n + m).

### Bruteforce

Teraz už vieme skontrolovať, či kostra robí graf šťastným. Čo môžeme spraviť ďalej je vygenerovať všetky možné kostry grafu a pre každú z nich skontrolovať, či daná kostra spĺňa podmienky zo zadania. Postupne generovať kostry sa dá viacerými spôsobmi napr. za pomoci bitmasiek ale jednoduchšie je možno použiť rekurzívne DFS. Bude to pomalé a výrazne náročnejšie na implementáciu ako vzorové riešenie. Ak máme napríklad $10^5$ hrán, 1001 vrcholov a chceme vybrať všetky kombinácie 1000 hrán tak to bude $\binom{10^5}{1000}$ a to vám kalkulačka odmietne vypočítať nakoľko je to príliš veľa. Odhad zložitosti bude približne $m!$ (m je počet hrán) a to by neprešlo ani na prvej sade veľmi pravdepodobne.

### Zaujímavá myšlienka

Zamyslime sa teraz nad tým, že väčšina kostier, ktoré by sme vygenerovali bruteforce riešením by nespĺňali podmienku, že všetky vrcholy sú šťastné, hlavne teda ak by kostra, ktorej vrcholy sú všetky šťastné neexistovala a my by sme museli generovať všetky možnosti. Namiesto toho, aby sme vytvorili kostru a potom kontrolovali, či sú vrcholy šťastné cheme kontrolovať či sú šťastné už keď ju vytvárame.

### Optimálne riešenie

Hrany budeme pridávať vo viacerých fázach. Stav kostry si budeme pamätať ako union-find. Pre každý vrchol si chceme pamätať, či sa nachádza v komponente a aký vrchol je v jeho komponente koreňom. Začneme tým, že prejdeme všetky hrany a pokiaľ majú oba vrcholy a aj hrana rovnakú farbu a zároveň sa ešte nenachádzajú v rovnakom komponente tak ich pridáme do kostry a upravíme rodičov ich komponentu. Teraz spustíme z rodičov komponentu BFS a budeme prechádzať cez jednotlivý komponent a pridávať také hrany ktoré spĺňajú podmienku že ešte nie sú v danom komponente a vedú do vrcholu ktorý má ich farbu, tiež samozrejme upravíme komponenty.

Po tejto fáze by mali byť všetky vrcholy spokojné. Pokiaľ nie sú tak kostra neexistuje. Prepokladajme sporom, že existuje riešenie ale náš algoritmus ho nenašiel. Postupovali sme spôsobom opísaným vyžšie a kedže riešenie existuje tak existuje aj hrana ktorú sme nepoužili ale je súčasťou riešenia. Tá hrana musí buď spájať vrcholy s ktorými oboma zdieľa farbu alebo spájať vrchol rovnakej farby s vrchol druhej farby ktorý je už šťastný (ak nerobí žiaden vrchol štastný tak zjavne nezmení to či to riešenie existuje alebo nie takže nás nezaujíma). To je ale v spore s prepokladom, že sme použili vyžšie opísaný algoritmus. Pokiaľ by robila oba vrcholy s ktorými je

spojená šťastné tak by bola pridaná už pri prvotnom prechádzaní hrán. Ak robí iba jeden vrchol šťastný tak by bola pridaná v druhej fáze, keď pripájame nové vrcholy cez BFS (všimnime si, že ak robí jeden vrchol šťastným ale druhý ostane neštastný tak tiež nie je relevantná pre riešenie lebo po jej pridaní ostane vrchol smutný).

Treba si teraz ešte uvedomiť to, že sme uspokojili všetky vrcholy neznamená, že máme kompletné riešenie ale iba vieme že existuje, mohli nám totiž ostať viaceré nespojené komponenty. To vyriešime tak, že prejdeme všetky hrany a už sa nezaujímame o to akú majú farbu a pridáme všetky také, ktoré nám spoja dva rôzne komponenty. Keďže graf na vstupe je súvislý tak určite existujú hrany, ktorými vieme aj našu kostru spraviť súvislú.

Ak ste nikdy nepočuli o union find tak odporúčam pozrieť materiály na ksp school (https://school.ksp.sk/courses/data-structures-2/union-find/union-find-intro/), skrátená verzia je taká že si pamätáme koreň komponentu a tiež hlbkú stromu keď si ho v tom vrchole zakoreníme. Keď spájame dva komponenty tak ako rodiča komponentu s nižšou hĺbkou nastavíme koreň väčšieho komponentu. Keď chceme zistiť v akom komponente je vrchol tak sa rekurzívne budeme volať na rodiča každého vrcholu až kým neprídeme k vrcholo ktorý je rodič samého seba a to bude hladaný koreň, pokial majú dva vrcholy rovnaký koreň komponentu tak sú v rovnakom komponente.

Stanko

## 6. Bazošové pásy

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

Prvé, čo si vieme všimnúť, je to, že nezáleží na poradí v akom budeme pásy umiestňovať.

### Bruteforce

Ak riešenie existuje, tak musí existovať nejaká podmnožina pásov, ktoré používa. Teda stačí skúsiť všetky podmnožiny a vybrať z nich tú najlepšiu. Toto riešenie bude mať časovú složitosť $O(2^n)$, lebo máme $n$ pásov a pre každú podmnožinu pásov musíme skúsiť, či spĺňa podmienky.

### Zaujímavá myšlienka

Tým, že chceme čo najväčšiu priepustnosť, tak môžeme postupne skúšať to vyrobiť z stále horších pásov. Teda pásy si vieme zoradiť od najviac priepustných po najmenej priepustné. Potom ak budeme pridávať pásy od najviac priepustných, tak vieme, že ak nájdeme riešenie, tak musí byť optimálne.

### Niečo lepšie

Postupne si generujeme všteky kombinácie pásov. Pre každú si pamätáme jej dĺžku a počet robotov. Vždy keď pridáme nový pás, tak prejdeme cez všetky predošlé kombinácie a ku každej z nich spravíme novú kombináciu, ktorá má už nový pás napojený. Ak je pás moc dlhý vieme túto kombináciu ignorovať - nemôže viesť k správnemu riešeniu. Zároveň, ak máme viac ako $r$ robotov v nejakej kombinácii, tak je to identické ako keby sme mali tých robotov práve $r$. Celkovo teda máme $O(l * r)$ kombinácii. Pri každom pridaní nového pásu musíme cez všetky prejsť. Takže časová zložitosť bude $O(n * l * r)$.

### Optimálne riešenie

Od optimálneho riešenia nás delí iba jedno pozorovanie - ak vieme spraviť pás dlhý 200, ktorý má 4 robotov alebo pás dlhý 200, ktorý má 2 robotov, tak tá prvá možnosť je lepšia. Ak bude totiž riešenie existovať využívajuce druhú kombináciu, tak musí existovať aj iné, ktoré využíva prvú. Naopak to však neplatí. Čiže nám stačí len pre každú dĺžku $x$ vedieť na koľko najviac robotov vieme vyskladať pás dlhý $x$. Nadpájanie bude fungovať tak, že pre všetky dĺžky tento pás napojíme, ak to ide a aktualizujeme maximálny počet robotov na novej dĺžke. Naraz si teda pamätáme iba $l$ čísel - maximálne počty robotov. Každý nový pás vyriešime jedným prejdením týmto poľom. Časová zložitosť bude $O(n * l)$.

Patrik

## 7. Exotické korenia

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

### Pozorovanie

Ako prvé si vieme všimnúť, že sa nám nikdy neoplatí navštíviť vrch (resp. spodok) nejakej priehradky dvakrát.

Dole sa robot musí presunúť práve raz, a preto vo všeobecnosti optimálny pohyb bude vyzerať nasledovne:

1. Robot sa presunie (pohybmi doprava alebo doľava) na vrch nejakej priehradky (resp. ostane tam, kde sme začínali)

2. Robot prejde na spodok priehradky
3. Robot prejde (pohybmi doprava alebo doľava) na spodok cieľovej priehradky (resp. ostane na mieste, ak tam už po druhom kroku je)

Navyše pohyb v prvom a treťom kroku bude zjavne iba doprava, alebo iba doľava (inak by robot navštívil vrch, resp. spodok nejakej priehradky dvakrát). Otázka je teda, ako vybrať priehradku, v ktorej robot vykoná pohyb dole.

### Bruteforce

Keď je $n$ malé vieme použiť hrubú silu, a vyskúšať všetky možnosti, kde by mohol robot ísť dole.

Ak by sme sa chceli dostať z vrchu priehradky číslo $x$ na spodok priehradky číslo $x'$ tak, že robot prejde dole na pozícii $i$, bude to trvať čas za ktorý sa dostame z $x$ na $i$ (teda $\min(|x-i|, n-|x-i|)$) plus čas na prejdenie dole priehradkou $i$ (čiže $a_i$), plus čas na dostanie sa z vrchu priehradky $i$ na vrch priehradky $x'$.

V čase $O(n)$ na otázku vieme prejsť všetky možnosti a vybrať minimum. Časová zložitosť tohoto riešenia je preto $O(nq)$ – všimnite si, že aj keď vieme udalosti typu ! simulovať v konštantnom čase, v najhoršom prípade nám každá otázka zaberie lineárne dlho.

Pamäťová zložitosť je $O(n)$, pamätáme si iba vstup v poli, ktorý postupne meníme.

### Špeciálny prípad – $|x - x'| = n/2$

Predstavme si teraz, že pre každú udalosť typu ? platí $|x - x'| = n/2$.

Vieme si všimnúť, že nech vyberieme akékoľvek číslo priehradky $i$ v ktorej robot vykoná pohyb dole, tak v horizontálnom smere vždy prejdeme iba vzdialenosť $n/2$.

Tým pádom je optimálne riešenie vybrať $i$ tak, aby sme minimalizovali $a_i$, takže vždy vyberieme najmenšie $a_i$, to vieme robiť rýchlo aj so zmenami použitím obyčajného intervalového stromu alebo haldy.

Časová zložitosť daného riešenia je $O(n + q \log n)$.

### Optimálne riešenie

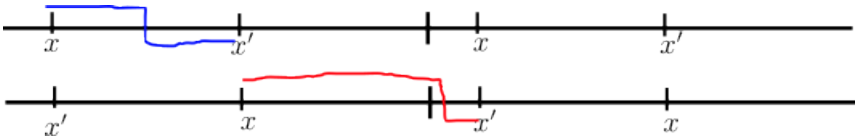
Ako sme si už všimli, optimálne riešenie vždy vieme rozložiť na 3 typy pohybov:

- Pohyb po vrchoch priehradiek v jednom smere.
- Pohyb dole.
- Pohyb po spodkoch priehradiek v jednom smere.

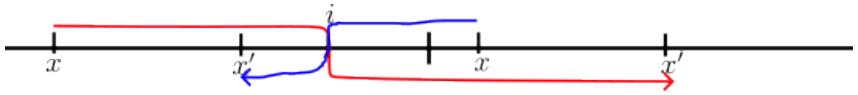
Pozrime sa na to podrobnejšie. V prvom a druhom kroku máme dve možnosti: ísť doprava, alebo doľava. Takto máme štyri typy ciest, ktorými vie robot ísť: doľava-dole-doľava, doľava-dole-doprava, doprava-dole-doľava, a doprava-dole-doprava.

Následne si ešte rozdelíme prípady podľa toho, či $x < x'$, alebo $x \geq x'$.

Pozrime sa na prípad, že náš pohyb je doprava-dole-doprava. Podľa relatívneho poradia $x$ a $x'$:




Pozorný čitateľ si môže všimnúť, že sme zdanlivo vynechali dva prípady: vo vrchnom obrázku by sa $i$ mohlo nachádzať za $x'$ (teda by sme cestou doľava prešli okolo $x'$). Avšak vtedy by sme horizontálne prešli vyše $n$ priehradiek. V tomto prípade sa však cesta ku $i$ dá zrýchliť, ak by sme išli iba doľava, viď obrázok:

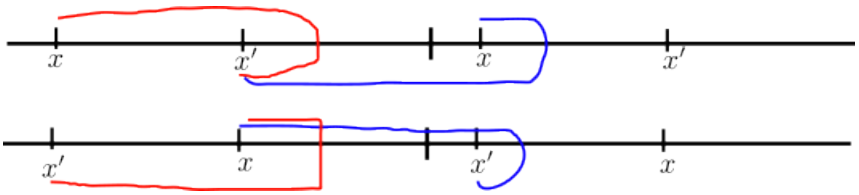



Takže máme dva prípady. Pozrime sa najskôr na prípad, kde $x < x'$, a dole prejdeme na priehradke číslo $x \leq i \leq x'$. Koľko to bude trvať? $a_i + (i - x) + (x' - i) = x' - x + a_i$. Takže na nájdenie najlepšieho indexu $i$ *pre tento prípad* nám stačí použiť intervalový strom na nájdenie $x \leq i \leq x'$ s najmenším možným $i$.

Čo sa stane ak $x \geq x'$. Rovnakým počítaním si môžme všimnúť, že sa nám oplatí jednoducho nájsť $i$ s najmenším $a_i$ pre ktoré buď platí $i \leq x'$ alebo $i \geq x$). Toto tiež vieme spraviť pomocou intervalového stromu.

Prípad doľava-dole-doľava, je v princípe identický, takže zostávajúce dva zaujímavé prípady sú doľava-dole-doprava a doprava-dole-doprava.

Ďalej sa pozrime na prípad doprava-dole-doľava. V tomto prípade máme štyri možnosti[1], ako môžeme uvidieť na nasledujúcom obrázku:




Aby sme sa vyhli ďalšej analýze prípadov (prechod cez $n$), všimnime si, že ak máme pole veľkosti $2n$, kde $a_i = a_{n+i}$, vtedy si môžme predstaviť, že namiesto $i < x$ (pri ktorom by sme museli prejsť cez $n$) prechádzame cez $i + n$. Teraz si vieme spraviť analýzu prípadov:

- Prípad $x < x'$
  - $x' \leq i \leq x+n$ (tento príklad je na obrázku zvýraznený červenou): táto cesta trvá $a_i+(i-x)+(i-x') = a_i + 2i - x - x'$
  - $x + n \leq i \leq x' + n$ (tento príklad je na obrázku zvýraznený modrou): táto cesta trvá $a_i + (i - (x + n)) + (i - x') = a_i + 2i - x - x' - n$ Prípad $x > x'$:
  - $x \leq i \leq x'+n$ (tento príklad je na obrázku zvýraznený červenou): táto cesta trvá $a_i+(i-x)+(i-x') = a_i + 2i - x - x'$
  - $x' + n \leq i \leq x + n$ (tento príklad je na obrázku zvýraznený modrou): táto cesta trvá $a_i + (i - x) + (i - (x' + n)) = a_i + 2i - x - x' - n$

Všimnime si, že v každom z týchto štyroch prípadov sa nám cena rozloží na $a_i + 2i$ a konštantu závisiacu iba od otázky (buď $x + x' + n$ alebo $x + x'$). Prvé číslo vieme pre každý prípad zistiť intervalovým stromom ktorý si na $i$-tej priehradke pamätá čísla $a_i + 2i$, druhé číslo záleží od prípadu.

A napokon, aj prípad doľava-dole-doprava vieme riešiť nápodobne, len miesto $a_i + 2i$ potrebujeme hodnoty $a_i - 2i$. Ďalšia možnosť je pozorovanie, že toto je to isté ako doprava-dole-doľava ak vymeníme $x$ a $x'$.

Takže aby sme to zhrnuli, potrebujeme niekoľko (dva alebo tri) intervalové stromy[2] s hodnotami $a_i$, $a_i + 2i$ (resp. $a_i - 2i$). Aby sme nemuseli riešiť prechody cez $n$, cyklickosť poľa implementujeme tak, že máme hodnoty dvakrát za sebou. Pri udalostiach typu ! jednoducho zmeníme hodnoty v intervalových stromoch na vhodných miestach.

Pri udalostiach typu ? si poriadne rozpíšeme možnosti, a spýtame sa vhodné otázky.

Ako rýchlo toto riešenie beží? Pre každú otázku použijeme konštantný počet queries pre intervalový strom, a teda použime $O(\log n)$ operácií pre každú udalosť. Na vybudovanie intervalových stromov a načítanie vstupu potrebujeme lineárny čas, takže dostaneme výslednú časovú zložitosť $O(n + q \log n)$.

Pamäťová zložitosť je $O(n)$, keďže si musíme uložiť vstup a intervalové stromy.

### Implementácia

Vzhľadom na veľké množstvo jednotlivých prípadov je sa v tejto úlohe veľmi ľahké pomýliť, preto pri implementácii si odporúčame použiť prístup lenivého programátora[4] a prípady si poriadne rozpísať, resp. nakresliť.

Maťo

## 8. Diverzita obedov

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

Táto úloha je pekný príklad problému, kde je kľúčové nájsť nejakú jednoduchú charakteristiku optimálneho riešenia a potom prehľadať výrazne menší priestor možností, ktoré podľa našej charakteristiky môžu byť optimálne.

V našom prípade bude pre skúsenejšieho riešiteľa pravdepodobne náročnejšia prvá časť, kde sa budeme snažiť vymyslieť peknú charakteristiku optimálneho riešenia, ktorá nám zredukuje priestor možností, ktoré musíme vyskúšať. Efektívne prehľadanie tohto priestoru sa bude potom dať vymyslieť s použitím pomerne štandardných techník.

[1] V skutočnosti je relevantná presne jedna, v závislosti na hodnote $x - x'$. Pre jednoduchosť uvažujme všetky.

[2] viď kuchárku[3]

[4] Princíp lenivého programátora je nasledovný: Bežný človek vidí problém, a hneď sa do neho vrhne. Výsledkom toho po troch hodinách bude dlhánsky program, plný chýb. Lenivý programátor sa najskôr hodinu bude zamýšľať, ako si ušetriť čo najviac roboty a okrajových prípadov. Potom to za pol hodinu naprogramuje. (Všimnime si, že lenivý programátor má kratší program, s menej chybami a ešte to celé spravil dvakrát rýchlejšie.)

### Nestačí orientovať hrany?

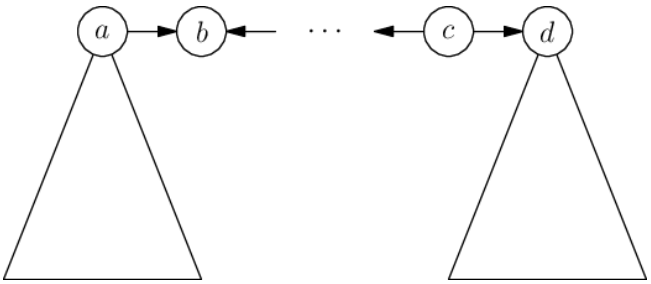
Hovoríme, že hrana je orientovaná z $u$ do $v$, ak $p(u) > p(v)$, kde $p(x)$ udáva číslo vrcholu $x$ po našom prečíslovaní. Nie je ťažké si všimnúť, že orientácia hrán v strome jednoznačne určuje počet klesajúcich ciest v prečíslovanom strome. Teda ak dostávame z viacerých prečíslovaní rovnakú orientáciu všetkých hrán, stačí vyskúšať jedno z nich.

Tuto môžeme využiť to, že na vstupe dostávame stále strom. Nech v ňom orientujeme hrany akokoľvek, ostane vždy acyklický. Teda po orientovaní hrán dostávame orientovaný acyklický graf. Vieme, že takýto graf má stále aspoň jedno topologické usporiadanie[5]. Ku každému možnému orientovaniu hrán vieme priradiť nejaké topologické usporiadanie takto orientovaného grafu a podľa neho prečíslovať vrcholy. Z toho už vyplýva, že ku každému orientovaniu hrán máme aspoň jedno korešpondujúce prečíslovanie a stačí nám prejsť všetky možné orientovania hrán.

Všetkých prečíslovaní vrcholov je $n!$ a možných orientácií hrán je $2^{n-1}$. Spočítať počet klesajúcich ciest, teda počet ciest po orientovaní hrán vieme v $O(n^2)$. Takto sme z triviálneho brute forcu, ktorý má zložitosť $O(n^2 \cdot n!)$, dostali lepší brute force so zložitosťou $O(n^2 \cdot 2^n)$. Ďalej sa budeme zamýšľať iba nad verziou, kde orientujeme hrany, keďže sme ukázali, že odpoveď bude rovnaká.

### Vnútorné cesty

Vnútornú cestu z vrcholu $c$ do vrcholu $b$ nazveme takú orientovanú cestu v orientovanom strome, že existujú orientované hrany z $a$ do $b$ a z $c$ do $d$. Dokážeme, že vnútorné cesty v optimálnom riešení byť nemôžu.




Pozrime sa, čo sa stane, ak otočíme hrany v celom podstrome $a$ a hranu $a \to b$. Nech:

- $A$ je počet ciest, ktoré končia v $a$ (vrátane cesty dĺžky 1, ktorá obsahuje iba $a$)
- $B_{out}$ je počet ciest dĺžky $\geq 2$, ktoré začínajú v $b$
- $B_{in}$ je počet ciest dĺžky $\geq 2$, ktoré končia v $b$ a nejdú cez hranu $a \to b$

Môžeme si všimnúť, že veľa ciest, ktoré boli orientované pred touto úpravou budú orientované aj po tejto úprave (iba sa v nich každá hrana otočí). Jediné cesty, ktoré sa mohli zmeniť musia obsahovať hranu $a \to b$. Preto sa stačí pozrieť na to, ako sa zmenil ich počet.

Pred úpravou týchto ciest bolo $A \cdot B_{out}$ : Všetky cesty končiace v $a$ vieme predĺžiť do nejakého vrcholu, do ktorého sa dá dostať cez $b$.

Po úprave ich bude $A \cdot B_{in}$ : Všetky cesty, z ktorých sa dá dostať do $b$, okrem tých, ktoré vedú cez $a \to b$ (to vyjadruje $B_{in}$), vieme predĺžiť všetkými cestami, ktoré sa končili v $a$ pred otočením, lebo po otočení sa v $a$ začínajú.

Počet orientovaných ciest sa nám teda zmení o $\delta_1 = A(B_{in} - B_{out})$. Rovnako vieme vypočítať, čo sa stane, ak otočíme hranu $c \to d$ a podstromy vrcholu $d$ neobsahujúce vrchol $c$. Počet orientovaných ciest sa v tomto prípade zmení o $\delta_2 = D(C_{out} - C_{in})$.

Teraz je dôležité si všimnúť, že každú orientovanú cestu, ktorá končí v $c$, vieme predĺžiť tak, že bude končiť v $b$ (lebo existuje orientovaná cesta z $c$ do $b$). Z toho vyplýva, že $B_{in} \geq C_{in} + 1$. Podobne vieme dokázať, že $C_{out} \geq B_{out} + 1$. Ak sčítame tieto nerovnosti, dostaneme $(B_{in} - B_{out}) + (C_{out} - C_{in}) \geq 2$.

Teda $\max(B_{in} - B_{out}, (C_{out} - C_{in}) > 0$ a keďže $A > 0$, tak aj $\max(\delta_1, \delta_2) > 0$. Teda po jednej z týchto úprav vieme zvýšiť počet orientovaných ciest. Týmto sme dokázali, že takzvaná vnútorná cesta nemôže v optimálnom riešení existovať, inak by sme vedeli získať lepšie riešenie.

Toto tvrdenie nám, ako ďalej uvidíme, veľmi pomôže. Výrazne a hlavne pre algoritmické riešenie veľmi príjemne nám zredukovalo priestor možností, ktoré nám naozaj treba vyskúšať. Ďalej si ukážeme, ako presnejšie musia vyzerať riešenia, ktoré budeme skúšať.

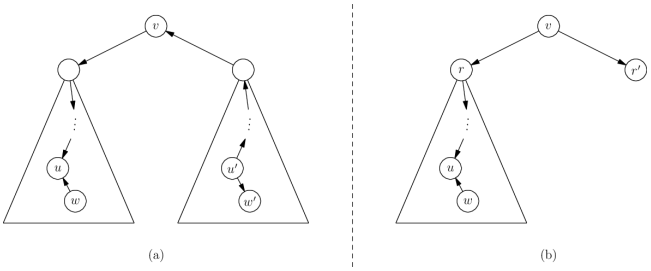
[5] https://en.wikipedia.org/wiki/Topological_sorting

### Centrálny vrchol

Centrálny vrchol nazveme taký vrchol orientovaného stromu, pre ktorý každá neorientovaná cesta končiaca v tomto vrchole je aj orientovaná cesta. Poriadne si premyslite, alebo nakreslite, ako môže orientovaný strom s centrálnym vrcholom vyzerať. Dokážeme, že takýto vrchol v orientáciách bez vnútorných ciest (len takéto orientácie majú šancu byť optimálne) musí existovať.

Najprv si zakoreníme strom v ľubovoľnom vrchole $v$. Zlý vrchol nazveme taký vrchol $u$, pre ktorý existuje cesta $(v, \ldots, u, w)$, ktorá mení orientáciu po vrchole $u$. Teda hrany sú od $v$ po $u$ orientované jedným smerom a hrana $u, w$ je orientovaná opačným smerom.

Ak vrchol $v$ nemá žiadny zlý vrchol, môžeme ho vyhlásiť za centrálny vrchol, pretože k nemu existuje orientovaná cesta z každého vrcholu v strome. Inak musí platiť, že všetky zlé vrcholy sa nachádzajú v jednom podstrome vrcholu $v$. Dokážeme to sporom s tým, že v optimálnom orientovaní nie je vnútorná cesta. Nech je zlý vrchol aspoň v dvoch podstromoch vrcholu $v$. Prípady si môžeme rozdeliť podľa toho, ako vyzerajú hrany pri vrchole $v$ na ceste medzi podstromami so zlými vrcholmi:




- Ak sú tieto hrany orientované jedna smerom ku $v$ a druhá od $v$, podľa obrázka (a) vidíme, že dostaneme vnútornú cestu, ktorá začína v $u'$ a končí v $u$.
- Ak sú tieto hrany orientované obe smerom od $v$, podľa obrázka (b) vidíme, že dostaneme vnútornú cestu z $v$ do $u$.
- Ak sú tieto hrany orientované obe smerom ku $v$, obrázok by bol zhodný s obrázkom (b), len by mal otočené hrany. Dostali by sme teda vnútornú cestu z $u$ do $v$.

Teda máme spor s tým, že v optimálnom orientovaní nie je vnútorná cesta, keďže v každom prípade, kedy sú zlé vrcholy v aspoň dvoch podstromoch, dostaneme nejakú vnútornú cestu. Stačí teda zmeniť vrchol $v$ na koreň podstromu, v ktorom je zlý vrchol a aplikovať rovnakú argumentáciu. Takto môžeme postupovať po ceste v strome, pokým nie je aktuálny vrchol $v$ centrálny.

Dokázali sme tak, že v optimálnom orientovaní hrán musí byť nejaký centrálny vrchol. Toto nám už výrazne zjednodušilo úlohu a môžeme sa začať zamýšľať nad efektívnym nájdením optimálneho riešenia v zostávajúcom priestore orientovaní, ktoré môžu byť optimálne.

### Prvé polynomiálne riešenie

Ako prvé sa črtá vyskúšať všetky vrcholy ako centrálne a pre každý nejak vypočítať, koľko najviac ciest vieme dostať, ak je tento vrchol centrálny. To sa po pár optimalizáciach ukáže aj ako vzorové riešenie.

Môžeme si všimnúť, že ak zvolíme vrchol za centrálny, ostáva už len rozhodnúť, ktoré podstromy budú orientované smerom ku centrálnemu vrcholu a ktoré smerom od neho. Hrany v celom podstrome musia byť orientované rovnako, lebo z každého vrcholu musí existovať orientovaná cesta k centrálnemu, alebo naopak. Tiež si môžeme všimnúť, že menenie orientácie celých podstromov zmení iba počet orientovaných ciest, ktoré vedú z jedného podstromu centrálneho vrcholu do iného podstromu centrálneho vrcholu. Počet ostatných orientovaných ciest ostáva nezmenený (iba sa celé otočia).

Poďme teda skúmať, koľko najviac orientovaných ciest vedúcich medzi dvoma rôznymi podstromami centrálneho vrcholu vieme dostať. Zaujíma nás, koľko ciest vieme viesť k centrálnemu vrcholu z každého z podstromov centrálneho vrcholu. To je celkom jednoduché - je to počet vrcholov v tomto podstrome.

Ak orientujeme smerom ku centrálnemu podstromy so spolu $x$ vrcholmi a smerom od všetky ostatné, dostaneme $x \cdot (n - 1 - x)$ ciest, ktoré vedú cez 2 rôzne podstromy. Z každého vrcholu v podstrome orientovanom ku centrálnemu vrcholu sa vieme dostať do každého vrcholu v podstrome orientovaného od centrálneho vrcholu.

Ak tento výraz upravíme dostaneme $x \cdot (n - x - 1) = \frac{(n-1)^2}{4} - (x - \frac{n-1}{2})^2$ Prvý člen je nezávislý na $x$, teda stačí minimalizovať druhý člen. Ten zjavne minimalizujeme, ak zvolíme $x$ čo najbližšie k $\frac{n-1}{2}$.

### Štandardná dynamika

Vidíme, že potrebujeme zistiť, či je možne rozdeliť množinu veľkostí podstromov centrálneho vrcholu na 2 časti s rovnakým súčtom. Existuje pomerne známy algoritmus založený na dynamickom programovaní, ktorý tento problém vie riešiť v čase $O(n \cdot k)$, kde $k$ je súčet prvkov v celej množine. Viac sa o ňom dočítate tuto[6]. Pre nás je ale $k = n - 1$, teda s jeho použitím dostaneme v najhoršom prípade časovú zložitosť $O(n^2)$ pre každý skúšaný centrálny vrchol. Ak sa takto rozdeliť množinu veľkostí podstromov nedá, chceme nejaké možné čo najrovnomernejšie rozdelenie. Dynamika, ktorú používame však vždy vypočíta aj odpoveď na túto otázku, pretože počas jej behu pre každú veľkosť rozdelenia zistí, či ju je možné dosiahnuť.

### Dopočítame ostatné cesty

Ostáva už len zistiť, koľko ciest je celých obsiahnutých v každom z podstromov centrálneho vrcholu. To vieme vypočítať jednoduchým dynamickým programovaním prejdením všetkých podstromov centrálneho vrcholu. Pre každý vrchol $u$ si budeme pamätať, koľko ciest z jeho podstromu v ňom môže končiť, vrátane cesty obsahujúcej iba tento samotný vrchol. To je vlastne počet vrcholov v celom jeho podstrome vrátane $u$. Túto hodnotu nazveme $u_{in}$. Budeme si tiež pamätať, koľko ciest je v celom jeho podstrome. Túto hodnotu nazveme $u_{cnt}$. Ak vieme tieto údaje pre deti vrcholu $u$, tak ich vieme vypočítať aj pre vrchol $u$:

- $u_{cnt} = \sum_{c \in children(u)} (c_{cnt} + c_{in})$ (cesty v podstrome vrcholu $c$ sa zachovajú a pribudne $c_{in}$ ciest končiacich v $u$)
- $u_{in} = \sum_{c \in children(u)} c_{in}$ (do $u$ vieme predĺžiť z každého podstromu iba tie cesty, ktoré končia v $c$)

Ak je vrchol $u$ list, tak stačí nastaviť $u_{cnt} = 0$ a $u_{in} = 1$. Týmto spôsobom vieme s použitím DFS jednoducho vypočítať počet ciest v strome, ktoré sú obsiahnuté celé v jednom podstrome aktuálneho centrálneho vrchola $v$. Maximálny počet ciest tak získame, ako $y \cdot (n - 1 - y) + v_{cnt}$, kde $y$ je $x$, ktoré nám dalo najväčší počet ciest medzi podstromami.

### Časová zložitosť

Ostáva ešte zistiť, akú časovú zložitosť toto celé vlastne bude mať. Pozrime sa, koľko práce vykonáme pre nejaký pevný centrálny vrchol $v$. Najprv chceme zistiť veľkosti jeho všetkých podstromov a počas toho môžeme aj vypočítať cesty, ktoré vedú celé v jednom podstrome. To sme si ukázali, že vieme jedným DFS, ktoré bude mať v našom prípade zložitosť $O(n)$.

Potom chceme vypočítať počet ciest vedúcich cez 2 podstromy pri optimálnom orientovaní podstromov. To dokážeme pomocou vyššie popísanej dynamiky v čase $O(n^2)$. Ukážeme, že aj keď to tak na prvý pohľad nevyzerá, aj výsledná časová zložitosť bude $O(n^2)$.

Označme počet susedov vrchola $v$ ako $deg(v)$. Potom pri zisťovaní optimálnej orientácie máme vlastne zložitosť $O(n \cdot deg(v))$. Je síce pravda, že $deg(v)$ môže byť až $n - 1$, ale použijme radšej tento predpis pri celkovom odhade časovej zložitosti. Dostaneme tak $O(\sum_{v=1}^{v=n}(n \cdot deg(v)) = O(n \cdot \sum_{v=1}^{v=n} deg(v))$. Súčet stupňov v celom strome je ale stále $2n-2$, teda $\sum_{v=1}^{v=n}(deg(v)) = 2n-2$ a celková časová zložitosť je $O(n \cdot (2n-2)) = O(n^2)$. Inak povedané, prechod poľa s možnými súčtami pri dynamike prechádzame v podstate pre každú hranu raz pre jej jeden koncový vrchol a raz pre druhý koncový vrchol. Hrán v strome a aj veľkosť poľa je $O(n)$, teda časová zložitosť je $O(n^2)$.

### Vzorové riešenie

### Iba centroidy nie sú triviálne

Centroid je vrchol v strome, ktorého žiaden podstrom nemá veľkosť väčšiu ako $\frac{n}{2}$. Pre vrchol, ktorý nie je centroid stačí orientovať veľký podstrom jedným smerom a ostatné druhým smerom. Teda len pre centroidy je potrebné riešiť úlohu spomenutým dynamickým programovaním. Je známe, že centroidy sú v strome stále najviac dva, teda dynamiku zbehneme konštantne veľa krát.

To samo o sebe ale nezlepší časovú zložitosť, keďže centroid môže mať až $O(n)$ detí a v tom prípade bude mať celý algoritmus kvôli dynamike časovú zložitosť $O(n^2)$. Skúsme preto zefektívniť dynamiku na súčty podmnožín.

### Triky z Ameriky

Teraz popíšeme možno nie až tak známy trik, pomocou ktorého je možné vyriešiť dynamiku na súčty podmnožín v čase $O(n \cdot \sqrt{n})$, kde $n$ je najväčší dosiahnuteľný súčet. Najprv je dobré si uvedomiť, že rôznych hodnôt,

[6] https://en.wikipedia.org/wiki/Pseudopolynomial_time_number_partitioning

ktoré sa sčítajú na $n$ musí byť najviac $O(\sqrt{n})$. To si vieme zdôvodniť napríklad pomocou vzorcu na výpočet súčtu aritmetickej postupnosti. Aj ak budeme používať čísla od 1 s diferenciou 1, dostaneme po $2 \cdot \sqrt{n} = O(\sqrt{n})$ číslach väčší súčet ako $n$. Všetky iné množiny rôznych čísel s aspoň $2 \cdot \sqrt{n}$ prvkami nám dajú zjavne väčší súčet ako je tento. Teda problém môžeme mať len vtedy, ak sa nejaké malé čísla často opakujú.

Povedzme, že máme 3 trojky. Môžeme si všimnúť, že čo sa týka dosiahnuteľných súčtov sme v tej istej situácii, ako keď máme 1 trojku a 1 šestku. Toto platí všeobecne. Ak máme prvok $a$ $a_{cnt}$-krát, kde $a_{cnt} \geq 3$. Ekvivalentné by bolo mať $1 + (x-1) mod 2$-krát prvok $a$ a $\lfloor (x-1)/2 \rfloor$-krát prvok $2a$. Touto úpravou zjavne nezmeníme súčet všetkých prvkov.

Ak v nejakom dosiahnuteľnom súčte použijeme $a$ párny počet krát, vieme to vyskladať iba z prvkov $2a$. Prípadne použijeme oba prvky $a$ ak bolo $a_{cnt}$ párne.

Ak v nejakom dosiahnuteľnom súčte použijeme $a$ nepárny počet krát, vieme to vyskladať z prvkov $2a$ a jedného prvku $a$. Keďže prvky $a$ po úprave najviac dva, nikdy sa nám nestane, že by sme nemali dosť prvkov $2a$ na vyskladanie daného súčtu.

Prišli sme teda na spôsob ako upraviť počty tak, aby sa množina možných súčtov nezmenila a zároveň sa prvok $a$ nachádzal v množine 1 alebo 2-krát. Ak túto úpravu spravíme so všetkými najmenšími $\sqrt{n}$ prvkami, nutne budeme mať $O(\sqrt{n})$ prvkov. Prvkov väčších ako $\sqrt{n}$ môže byť najviac $\sqrt{n}$ a kvôli úpravám, ktoré sme spravili máme medzi menšími prvkami ako $\sqrt{n}$ najviac $2 \cdot \sqrt{n}$ prvkov.

Dokopy máme teda $\sqrt{n} + 2 \cdot \sqrt{n} = O(\sqrt{n})$ prvkov. Týmto sme dostali algoritmus, ktorý spočíta všetky možné súčty v čase $O(n \cdot \sqrt{n})$. Najprv spravíme úpravy na malých prvkoch a potom použijeme predchádzajúci algoritmus.

### Už iba prekoreňovať

Aj keď teraz už časová zložitosť vyzerá nádejnejšie, stále je v skutočnosti $O(n^2)$. Pri počítaní ciest, ktoré sú celé v nejakom podstrome stále pre každý vrchol prejdeme celý strom. Ak by sme sa tohto zbavili, získali by sme už naozaj lepšiu časovú zložitosť.

Takáto situácia je už skúsenejšiemu riešiteľovi pomerne známa. Stačí použiť techniku prekoreňovania. Ak spravíme najprv DFS z vrcholu 1 na vypočítanie $u_{in}$ a $u_{cnt}$ pre všetky vrcholy, získame potrebné hodnoty pre podstromy detí (podľa zakorenenia vo vrchole 1) každého vrcholu. Chceme teda už len vypočítať hodnoty pre "nadstrom" nášho rodiča (podľa zakorenenia vo vrchole 1). Presne toto vieme vďaka technike prekoreňovania. Stačí použiť jedno DFS na predpočítanie hodnôt pre každý podstrom podľa zakorenenia v 1 a pri druhom DFS si už budeme vedieť správne udržiavať hodnoty pre náš nadstrom. Ak túto techniku nepoznáte, môžete sa o nej viac dočítať v USACO guide[7]. Pre náš konkrétny prípad sa môžete pozrieť do implementácie.

### Finálna časová a pamäťová zložitosť

Už len stačí overiť, že je časová zložitosť naozaj taká, ako chceme. Najprv spravíme DFS na vypočítanie potrebných údajov o podstromoch v $O(n)$. Potom počas druhého DFS pre necentroidy vypočítame odpoveď v $O(1)$ a pre centroidy v čase $O(n \cdot \sqrt{n})$. Centroidov je ale konštantne veľa, teda finálna časová zložitosť bude $O(n \cdot \sqrt{n})$. Pamäťová zložitosť je $O(n)$, pretože si pamätáme iba strom a pole s možnými súčtami.

### Bonusy

Prekoreňovanie na vyriešenie tohto problému dokonca ani nebolo treba. Dá sa dokázať, že optimálny centrálny vrchol bude vždy centroid a teda stačí strom zakoreniť v ňom (resp. v obidvoch z nich, ak sú 2). Časovú zložitosť nám to však nezhoršuje, tak sme sa kvôli lenivosti dokazovať ďalšie veci rozhodli použiť prekoreňovanie :)).

Algoritmus sa dá zrýchliť použitím C++ bitsetov[8]. Pri dynamike si uložíme boolovské pole ako bitset `b`. Pri pridávaní prvku `x` stačí spraviť `b |= (b << x)`. Táto neasymptotická optimalizácia vie občas výrazne zrýchliť kód v praxi. Na získanie plného počtu bodov za program to ale nebolo treba.

[7] https://usaco.guide/gold/all-roots?lang=cpp
[8] https://usaco.guide/plat/bitsets?lang=cpp