Korešpondenčný seminár z programovania
XLII. ročník, 2024/25
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola zimnej časti

Prutky

1. Prvý medzihviezdny fastfood

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

Ako prvé si môžeme všimnúť, že keď chceme nájsť stred najmenšieho obdĺžnika, ktorý obsahuje všetky znaky `#`, tak musíme nájsť okraje tohto obdĺžnika. Stred získame tak, že nájdeme stred medzi pravým a ľavým okrajom a stred medzi horným a dolným okrajom. Ako však vieme tieto okraje nájsť?

Ukážeme si to pre pravý okraj. Aby sme dostali pravý okraj, musíme nájsť taký stĺpec, pre ktorý platí, že všetky znaky `#` sú buď v ňom alebo naľavo od neho. Takže s pozíciou x menšou alebo rovnou tomuto stĺpcu. Keďže chceme najmenší obdĺžnik, tak chceme najlavejší takýto stĺpec. A ten bude obsahovať `#`. Ak by neobsahoval `#`, tak by sme mohli stĺpec posunúť viac doľava a stále by sme mali platný obdĺžnik. Takže nájdeme najľavejší stĺpec, ktorý obsahuje `#` a to bude pravý okraj.

Takto sa vieme pozrieť na všetky okraje. Dojdeme k tomu, že hľadáme pozíciu najvyššieho, najnižšieho, najviac vľavo a najviac vpravo položeného `#`, a to budú okraje nášho obdĺžnika.

Ako vieme teda nájsť `#` ktoré je najviac vľavo a ktoré je najviac v pravo? Môžeme si prejsť celé pole a v každom riadku nájsť prvý a posledný výskyt `#` a následne tieto hodnoty porovnať s doterajšími krajnými `#`. Ak sú viac na kraji, ako doterajšie maximum tak si ho prepíšeme. Takto získame súradnice x_l a x_p a výsledná hodnota x pre stred obdĺžnika bude ich priemerom.

Horné a dolné `#` zistíme tak, že keď budeme prechádzať každý riadok a narazíme na prvé `#`, tak tento riadok bude horný okraj. Uložíme si ho do y_h. Spodným okrajom bude posledný riadok, v ktorom sme uvideli `#`. Ten si označme ako y_d. y-ová súradnica stredu bude teda priemer y_h a y_d.

Pri riešení si treba dať pozor, či na vstupe bolo aspoň raz `#`. Ak nie, musíme vypísať stred odĺžnika tak, aby bol stredom celej galaxie. Takže 0-tého a $x-1$-ého stĺpca a 0-tého a $y-1$-ého riadka.

Keďže na vstupe dostávame postupne riadky, vieme ich vyhodnocovať tiež postupne. Tým vieme znížiť pamäťovú zložitosť nášho riešenia na $O(x)$. Kebyže veľmi chceme, vstup vieme načítavať znak po znaku, čím dosiahneme konštantnú pamäťovú zložitosť. To ale naprogramovať v pythone nie je až také jednoduché a presahuej to rámec tohoto vzoráku. Ako to spraviť v `C++` si môžete pozrieť vo vzorovom riešení (ktoré pribudne po doprogramovávaní).

Časová zložitosť bude $O(x \cdot y)$, keďže musíme prejsť celý vstup. Pre každý znak spravíme konštantne veľa operácií.

Vladko

2. Lakťovaná

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

Rozoberme si úlohu po prípadoch, ktoré môžu nastať

1. **Majoritný element**: Pretože hľadáme dĺžku najkratšieho stringu, sa v tomto prípade budeme snažiť ulakťovať čo najviac členov majoritného tímu. Teda riešením bude priradiť každého, kto nie je z majoritného tímu k niekomu z majoritného tímu. Teda ak je v nejakom tíme viac hráčov ako všetkých zvyšných hráčov, tak odpoveď je jednoducho $X - (N - X)$, kde X je počet hráčov v najpočetnejšom tíme a n je počet všetkých hráčov. $N - X$ je počet hráčov všetkých zvyšných tímov, a teda rozdiel počtu hráčov v najpočetnejšom tíme a všetkých zvyšných hráčov je minimálny počet hráčov, ktorým neostane súper, teda prežijú.
2. **Iba jeden tím**: Vtedy je odpoveď počet všetkých hráčov, pretože sa nemá kto laktovať.

3. **Ostatné prípady**: Všimnime si, že v tomto prípade vieme vytvárať dvojice až kým nám neostane iba jeden alebo nula hráčov. Kedže stále odoberáme po dvoch hráčoch, teda po párnym počte, tak sa nám párita zachová až dokonca. Teda odpoveď je zvyšok po delení dvojkou pôvodného počtu hráčov.

Časová a Pamäťová zložitosť:

Potrebujeme jeden prechod stringom na to, aby sme si zapamätali počty výskytov pre každé písmenko. Písmenká si môžeme pamätať v obyčajnom poli veľkom 26 tak, že `a` by bolo na indexe `0` a `z` na indexe `25`. Toto vieme v lineárnom čase, teda v `O(n)`.

Ďalej už len nájdeme maximum v poli veľkom 26, čo je `O(1)`, a skontrolujeme pár podmienok, čo je tiež `O(1)`.

Teda celková časová zložitosť je `O(n)`.

Čo sa týka pamäte: Teoreticky si vieme načítavať znaky zo vstupného stringu po jednom a pamätať si len pole veľké 26. Teda pamäťová zložitosť je `O(1)`, samozrejme ale akceptujeme aj keď niekto uviedol `O(n)`.

Fipo

3. Asteroidné T-rexy útočia

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

Začiatočné pozorovania

Rozdeľme si kravy na dva druhy: 1) pomalé - tie, ktoré nestihnú dobehnúť do chlieva ani keby ich neblokovala žiadna iná krava 2) rýchle - tie, ktoré stihnú dobehnúť do chlieva ak ich nebude blokovať žiadna iná krava

Ak je nejaká krava za pomalou, tak sa nemá šancu dostať do chlieva bez toho, aby Ralbo zodvihol tú pomalú. Ak nejaká rýchla krava narazí na pomalú, tak v ten moment Ralbo určite nespraví zodvihnutie naviac, ak zdvihne tú pomalú. Ak by tú pomalú nezdvihol hneď, tak by sa tá rýchla nemusela dostať do chlieva.

Ak nejaká rýchla krava narazí na pomalšiu rýchlu, tak sa obe stihnú dostať do cieľa, takže ich výmena by bola zbytočná.

To znamená, že na vyriešenie úlohy potrebujeme nájsť p rýchlych kráv, ktoré sú najbližšie k chlievu. Potom pre každú rýchlu kravu potrebujeme zistiť, koľko pomalých je pred ňou.

Bruteforce

Spravíme si pole boolov, ktoré si označíme napríklad A. V tomto poli si pre každú kravu budeme pamätať, či je pomalá alebo rýchla. To, či je krava rýchla spočítame tak, že vydelíme jej vzdialenosť od chlieva jej rýchlosťou a zistíme, či je menší ako čas príchodu t-rexov (dá sa to zvládnuť aj bez delenia, to nechám na vás :)). Keď nájdeme rýchlu kravu na indexe i a nemáme ešte p kráv Tak prejdeme poľom od 0-tého indexu po $i-1$ a za každú pomalú pripočítame k odpovedi 1. Nakoniec už iba vypíšeme odpoveď. Toto riešenie má časovú zložitosť $O(n^2)$ keďže pre každú kravu potrebujeme skontrolovať všetky pred ňou. Pamäťovú má $O(n)$, pretože si pamätáme pole kráv.

Zaujímavá myšlienka

Nás nezaujíma, kedy sme narazili na pomalú kravu. Pre každú rýchlu kravu potrebujeme vedieť iba koľko pomalých kráv je pred ňou. Na to nepotrebujeme pole, stačí nám na to iba jedna premenná.

Optimálne riešenie

Označme si premmenú *slow*, kde si budeme pamätať počet pomalých kráv, ktoré sme už videli. Vždy, keď príde krava, skontrolujeme, či je pomalá. Ak je pomalá, inkrementujeme *slow*. Ak je rýchla, tak k odpovedi pripočítame aktuálnu hodnotu *slow*. Toto riešenie má časovú zložitosť $O(n)$, pretože každú kravu spracujeme v konštantnonm čase. Pamäťovú má $O(1)$, pretože si potrebujeme pamätať len konštante veľa premenných.

MisQo

4. Nekončiaca Existenčná Kríza

(max. 10 b za popis, 10 b za program)

Vzorák tejto úlohy čaká na koniec série Prasku, potom sa tu objaví.

Sebik

5. Éra fanúšika hviezdnych bitiek

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

Štandardný pavúk je v informatických pojmoch vlastne úplný binárny strom, čo je taký strom, pre ktorý má každý vrchol okrem listov (tých naspodu) práve dvoch synov. V našom prípade má každý zápas, teda vrchol,

cenu, a v listoch (tímy) máme uložené, koľko najviac zápasov si vieme dovoliť vymeškať pre tento tím. Hľadáme teda nejakú množinu vrcholov, ktorá by spĺňala, že pre každý tím (list) je na jeho ceste do koreňa (finále) najviac a_i nekúpených zápasov.

Bruteforce

Najprv sa pozrime na to, čo sa dalo robiť s druhou sadou, kde sme mali zápasov najviac 16. Tu sme vedeli vyskúšať každú kombináciu zápasov a skontrolovať, či nám pokryje tímy tak ako chceme, a potom z tých fungujúcich vybrať tú najlacnejšiu. Na to by sme potrebovali vygenerovať každú kombináciu. To vieme spraviť napríklad tak, že sa pre každý vrchol rozhodneme, či do množiny ktorú kupujeme patrí alebo nie, a teda máme dve možnosti pre 2^r zápasov, teda totálne musíme vyskúšať 2^{2^r} možností. Pre každú možnosť potrebujeme však skontrolovať, či naozaj pokrýva tímy podľa zadania. To vieme spraviť DFSkom v čase $O(2^r)$, kedy si budeme jednoducho počítať, koľko zápasov sme na ceste od finále k tímu kúpili. Celková časová zložitosť je teda $O(2^{2^r})$, čo je naozaj pomalé, ale pre túto sadu to stačí.

Zaujímavá myšlienka

Pozrime sa teraz na prvú sadu, kde majú všetky zápasy rovnakú cenu. Určite sa nám teda vždy oplatí pre každý tím kupovať tie zápasy, čo by mohli hrať najneskôr, lebo týmto pokryjeme najväčšie množstvo tímov. Teda ak pre nejaký tím potrebujeme mať kúpený aspoň jeden zápas, tak sa nám určite oplatí kúpiť finále, pretože to stojí rovnako ako iné zápasy a ešte nám to pokryje najviac iných tímov. Teda náš výsledný strom bude vyzerať asi tak, že si pre každý tím necháme prvých a_i zápasov nekúpených, a potom zvyšné až do finále kúpime. Toto nám pekne pokryje všetky tímy, a vieme tento algoritmus pekne spraviť DFSkom v $O(r2^r)$, čo úplne stačí.

Niečo všeobecnejšie

(Tu už referujem na zápasy ako na vrcholy) Čo by sme však museli spraviť, ak by sa nám zrazu tie ceny lístkov všelijako pomenili? Povedzme, že sme si už vyplnili strom tak ako v prvej sade, a zmenili sa nám ceny. Ako vieme zistiť, čo sa nám už teraz neoplatí? No a tu prichádzame k zaujímavej myšlienke: **Riešenie nie je optimálne, ak kupujeme vrchol A, a v jeho podstrome existujú také nekúpené vrcholy s cenou v súčte menšou ako A, že ak by sme nekúpili A a kúpili tieto vrcholy, riešenie je validné.** Hľadáme teda, či nevieme nejaký vrchol jednoducho nahradiť nejakými inými z jeho podstromu, ktoré sú v súčte lacnejšie. Napríklad, namiesto finále by sa nám možno mohlo oplatiť kúpiť oboje semifinále. Ako by sme to mohli spraviť?

Vieme pustiť nejaké DFS z nášeho vrcholu A. Pre každý vrchol, ktorý navštívime, sa ho opýtame: nájdi mi, ako najlacnejšie by si mohol pokryť tímy vo svojom podstrome ešte jedným zápasom (to je ten, ktorý odstraňujeme vyššie vo vrchole A). Potom keď obaja synovia pošlú túto hodnotu, máme dve možnosti. Buď je optimálne ten zápas v A v tomto podstrome nahradiť týmto vrcholom, alebo nejakými z jeho podstromov. A tak súčet tých hodnôt od synov porovnáme s cenou vrcholu, v ktorom sme (ak už nie je kúpený). Tú lepšiu cenu pošlem svojmu rodičovi. Potom keď prídem do A, už viem, ako najlacnejšie by bolo možné "nahradiť" to, že odstránime tento vrchol A.

Takže odhora si pre každý aktuálne kúpený vrchol skontrolujem, či ho neviem nejako lacnejšie nahradiť vrcholmi v jeho podstrome. Je dôležité, aby som to robil odhora, pretože takto keď už prejdem tie horné vrcholy mám garantované, že sú v optimálnom stave. Musím si však dať pozor na to, aby som nejaký tím nepokryl viac ako potrebujem. To vyriešim napríklad tak, že v každom vrchole si pamätám, o koľko viac zápasov ako aktuálne potrebuje má nad sebou. Toto vieme tiež updatovať DFSkom po každej iterácii. Kadžé spustené DFS nám potrvá $O(2^r)$, tak zložitosť celého algoritmu bude $O(2^r * 2^r)$. Takéto riešenie by nám stačilo na získanie 6tich bodov, za predpokladu, že v prvej sade robíme algoritmus popísaný vyššie. Čo sa však dá robiť na plný počet bodov? No, zamyslime sa (najťažšia časť príkladu).

Ako to zlepšiť?

Už teraz si v DFSku počítame, ako zlepšiť hodnoty pokrytia pre každý vrchol, tak nevieme pustiť jedno DFSko a spočítať to všetko naraz? DFSko musíme samozrejme upraviť, lebo teraz počíta najlacnejšie ceny pre aktuálny stav toho čo kupujeme, a my by sme ich menili počas toho počítania. Vyriešime to tak, že si pre každý vrchol vyrátame, koľko "navyše" by sme chceli v jeho v podstrome tých pokrytí kúpiť. Respektíve, pýtame sa niečo takéto: Odstránil som t vrcholov nad tebou, ako najlacnejšie vieš pokryť vo svojom podstrome to, čo hore ubudlo? Toto nám hovorí o všetkých úpravách zároveň.

Na túto otázku viem zodpovedať podobne ako pri prvom DFSku, avšak si proste pre každý vrchol pamätám každú hodnotu, na ktorú sa vrchný vrchol môže pýtať. Všimnime si, že týchto hodnôt môže byť len málo (iba r), takže ani časová zložitosť nebude taká zlá. V tom našom DFSku si potom pre vrchol prejdeme všetky požiadavky

na pokrytie čo môže dostať (teda r hodnôt), a vyplníme ich najlacnejšie pokrytie pomocou najlacnejších pokrytí takých istých požiadaviek na pokrytie od jeho detí, prípadne zarátame aj seba. Triviálne.

A v tomto momente si možno chytáte hlavu, pretože nechápete o čom to točím, alebo s víziou implementačného pekla a debuggovacej bolesti pred očami. Avšak s 25 riadkami implementácie nás príde zachrániť dynamické programovanie.

Optimálne riešenie

Niektorí ste už možno robili dynamické programovanie na poli. Robili ste ho už na stromoch?

Ešte raz prečítame náš návrh na to nové, upravené DFSko. Parafrázujem: pre každý vrchol chceme vedieť pre každú hodnotu x od 0 do r, ako najlacnejšie viem kúpiť zápasy tak, aby nad týmto vrcholom mohlo chýbať o x vrcholov viac. Všimnime si, ako nešikovné a ťažko spracovateľné je toto zadefinovanie. Doteraz sme ho používali, lebo viedlo od našej pôvodnej myšlienky kontrolovať každý vrchol zvlášť (teda x sa vtedy rovnalo 1), avšak to už teraz nechceme robiť. Zadefinujme si teda nový cieľ: **Pre každý vrchol chceme vedieť, koľko nás najmenej bude stáť, aby sme ešte mohli na ceste do finále vymeškať x zápasov.**

Takéto zadefinovanie je veľmi šikovné, lebo môžeme pekne prirodzene ísť od listov stromu. Pre každý vrchol si chceme vypočítať pole dĺžky $r+1$. V i - tom políčku tohoto poľa (indexované od nuly) bude, koľko najmenej by nás stálo, aby sme mohli vo zvyšných zápasoch odtiaľto až do finále vymeškať ešte i zápasov. Potom postupujeme takto: Ako vypočítať i - tú hodnotu v tom poli pre vrchol? Napíšem tu taký dlhý vzorček a potom ho vysvetlím. $DP[vrchol][i] = min(DP[syn1][i] + DP[syn2][i] + cena[vrchol], DP[syn1][i+1] + DP[syn2][i+1])$

Poďme si to rozobrať. Na to, aby sme vedeli po tomto vrchole vymeškať i ďalších, môžeme buď zaplatiť: - Zobrať najlacnejší spôsob ako môžem vymeškať ešte i zápasov od mojich synov, a kúpiť samého seba. $(DP[syn1][i] + DP[syn2][i] + cena[vrchol])$ - Zobrať najlacnejší spôsob ako môžem vymeškať ešte $i+1$ zápasov od mojich synov, a vymeškať samého seba $(DP[syn1][i+1] + DP[syn2][i+1])$

Vždy si vezmem lepšie riešenie z týchto dvoch, a to bude tým najlepším riešením. Takto viem ísť pekne od listov po kolách hore, alebo aj DFSkom. To je však náročnejšie. No a aká by bola časová zložitosť? V každom vrchole musíme zbehnúť ten vzorček napísaný vyššie pre každé i. Teda to bude jednoducho $O(r * 2^r)$, a toto nám teda zbehne vo všetkých sadách.

Macker

6. Taylor sendvič

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

Pár slov na začiatok

Tento príklad sa dal na plný počet riešiť dvoma spôsobmi.

Jeden je náročnejší, no myšlienky v ňom sú možno trochu všeobecnejšie. Ten sa dá vymyslieť celkom jednoducho, pokiaľ poznáte správne nástroje. Je pri ňom možno oveľa jasnejšie, že funguje, no je o niečo ťažší na implementáciu. Tento rozoberieme postupne po subtaskoch na začiatku.

Na konci je druhý prístup, ktorý je jednoduchší na vysvetlenie a implementáciu, no ťažší na vymyslenie a uvedomenie si, že naozaj funguje. Ten rozoberieme v poslednom odstavci.

Bruteforce

Je ťažké pozerať sa naraz na veľa farieb, pozrime sa teda na jednu. Pre jednu farbu `x` vzniknú "územia", kde sú iné farby ohraničené farbou `x`. Cesta, čo sa začína na okraji tohto územia a končí niekde inde na okraji je jedna z hľadaných ciest. Začnime prehľadávanie v jednom z týchto okrajových bodov a vždy, keď nájdeme vrchol farby `x`, ďalej už nepokračujeme. Všetky takéto konce nám tvoria validné cesty zo začiatku prehľadávania. Keď pustíme prehľadávanie z každého vrchola (každý je niekedy okrajom nejakého územia), dostaneme výsledok.

Toto riešenie má časovú zložitosť $O(n^2)$, čo samo o sebe stačilo na polovicu bodov.

Prečo sa nepozerať na všetky vrcholy naraz?

Keď prechádzame veľakrát DFS-kom strom, prechádzame kopu vrcholov, s ktorými nič nerobíme, lebo nie sú správnej farby. Poďme to teda robiť všetko naraz.

Strom si v ľubovoľnom vrchole zakoreníme a spustíme z neho prehľadávanie do hĺbky. V každom vrchole si budeme pamätať pre každú farbu, ku koľkým takýmto "okrajovým vrcholom" v jeho podstrome môže viesť takáto cesta (pamätáme si to napríklad v mape s farbou a počtom okrajových vrcholov). Vždy keď sme v nejakom vrchole s farbou `x`, pozrieme sa, koľko okrajových vrcholov s farbou `x` je v jeho podstrome. To sú všetky cesty, čo môžu vychádzať z neho smerom dole. Toto nám už stačí na 3. podúlohu, kde sme na ceste, teda v DFS-ku idú všetky cesty iba hore. (aj keď sa dala riešiť aj oveľa jednoduchšie)

Ale vo všeobecnom strome budeme mať ešte cesty, ktoré nejdú iba hore. Každú takúto cestu zarátame v jej najvyššom vrchole. Takže keď spracovávame nejaký vrchol, môžeme zobrať nejakú farbu `x` okrajových vrcholov v jednom podstrome a poslať z nich cestu do okrajových vrcholov tej istej farby v ostatných podstromoch.

Ako si udržiavať takéto informácie v DFS-ku?

V každom vrchole máme v mape pre všetky farby ich príslušné počty okrajových vrcholov. Z ich otca sa dá dostať do tých istých vrcholov, okrem tých, ktoré majú rovnakú farbu ako ich otec, lebo ich otec zablokuje. Potrebujeme teda spojiť mapy pre synov do jedného pre otca a zmeniť hodnotu pre jeho farbu na 1.

Na toto použijeme metódu, ktorá sa volá "small to large merging". Jej myšlienka je jednoduchá, vždy keď spájame dve mapy, jednu z nich musíme celú presypať do druhej. Presypeme teda tu menšiu. (to, ku ktorému vrcholu patrí ktorá mapa vieme ľahko zmeniť pomocou metódy `swap`). Ak sa na tento proces pozrieme z pohľadu prvku, tak ak je v nejakej mape a presype sa do inej, veľkosť mapy, do ktorej sa práve presypal, sa aspoň zdvojnásobí. Finálna veľkosť mapy je najviac n, teda presypaní každého prvku bude najviac $O(log(n))$.

Každé prehodenie prvku trvá v mape tiež $O(log(n))$, teda dokopy bude časová zložitosť $O(nlog^2(n))$, prípadne $O(nlog(n))$ ak použijeme hash mapu. Pamäte nám stačí $O(n)$, lebo každý vrchol je naraz len v jednej z máp. Toto riešenie stačilo na plný počet bodov.

Optimálne riešenie

Dá sa to ale ešte rýchlejšie. Na cesty sme sa pozerali v ich najvyššom bode, ale vrátme sa na chvíľu na počítanie podľa okrajových bodov.

Graf si znova zakoreníme v ľubovoľnom vrchole a spustíme z neho prehľadávanie do hĺbky. Cesty si ale tentokrát zorientujeme tak, aby začiatok danej cesty bol v DFS prechode navštívený skôr ako jej koniec. Teda keď v DFS-ku prechádzame nejaký vrchol, zaujíma nás, koľko možných začiatkov tejto farby je prístupných z vrcholov, ktoré sme už navštívili.

Keď prejdeme cez vrchol farby `x` do jeho podstromu, vrchol zablokuje všetky začiatky, ktoré boli prístupné, teda jeho hodnotu v zozname nastavíme na 1. Keď z jeho podstromu vychádzame, zablokujú sa všetky, čo sme videli v podstrome, no odblokujú sa tie, čo sme pred tým zahodili, a pribudne 1 za tento vrchol, ktorý je teraz tiež dostupný a už sme ho spracovali v DFS-ku.

Odpoveď počítame tak, že vždy pred tým ako, pôjdeme do podstromu a zablokujeme niektorú farbu, spočítame, koľko ciest ide z tohto vrchola do odblokovaných navštívených vrcholov (to sú cesty do všetkých vrcholov, ktoré ideme teraz zablokovať).

Počty začiatkov si môžeme pamätať v zozname, kde sa v každom vrchole mení len jedna hodnota, teda toto riešenie dosahuje časovú zložitosť $O(n)$ a pamäťovú tiež $O(n)$.

Merlin

7. Kryptosekuritné problémy

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

Bruteforce

Prvé pozorovanie, ktoré môžeme spraviť je, že všetkých permutácií, ktorými mohol byť disk zašifrovaný, je strašne veľa. Určite teda nemôžeme vyskúšať všetky permutácie. Čo ale môžeme spraviť je, pre každú pozíciu v T vyskúšať, či sa na nej môže začínať P.

To môžeme spraviť napríklad tak, že postupne prechádzame písmenkami a konštruujeme permutáciu, pre ktorú sa bude P nachádzať v T (počínajúc daným indexom). Časová zložitosť tohto algoritmu bude $O(|T||P|)$, čo stačí na prejdenie prvej sady.

Kanonický tvar

Jedno slovo sa nám môže zašifrovať na veľa iných. Preto by sme chceli spomedzi všetkých týchto slov nejaké vybrať a prehlásiť ho za **kanonický tvar** daného slova. Môžeme si všimnúť, že ak majú dve slová A a B rovnaký kanonický tvar K, tak sa jedno mohlo zašifrovať na druhé.

Zo slova A vieme dostať slovo K nejakou permutáciou. Rovnako aj zo slova B vieme dostať K nejakou permutáciou. Inverz tejto permutácie je tiež permutácia, preto aj z K vieme nejakou permutáciou dostať B. Keď tieto dve permutácie spolu zložíme, dostaneme ďalšiu permutáciu, ktorá z A spraví B.

Ak naopak slová A a B nemajú rovnaký kanonický tvar, tak sa nemôže jedno zašifrovať na druhé. To platí preto, lebo ak by sa z A dalo vytvoriť nejakou permutáciou B, tak by množina všetkých slov, na ktoré sa dá A zašifrovať, bola rovnaká ako množina slov, na ktoré sa dá zašifrovať B. Keďže sú ale tieto množiny rovnaké, tak by A a B museli mať rovnaký kanonický tvar, čo je spor.

Ako si vyberieme tento kanonický tvar? Dobre napríklad funguje lexikograficky najmenší reťazec. Ten vieme dostať tak, že postupne prechádzame dané slovo a pamätáme si najmenšie nepoužité písmenko. Ak natrafíme v slove na nové písmenko, tak mu v permutácií priradíme najmenšie nepoužité písmenko a zvýšime ho o 1.

Optimálne riešenie

V prípade, že by disk nebol nijakým spôsobom zašifrovaný, tak by táto úloha bola celkom štandardná. Stačí len nájsť vzorku v texte, čo dokážeme pomocou KMP alebo rolling hashov (Rabin-Karp). Ak by sa nám teda podarilo nejak previesť slová na ich kanonické tvary, tak by sa z úlohy stala úloha riešiteľná jedným z týchto algoritmov.

Z týchto algoritmov si vyberieme rolling hash. Previesť reťazec P do kanonického tvaru a spočítať jeho hash je triviálne. Zaujímavejšie je, ako počítať postupne hashe úsekov reťazca T.

Budeme klasicky počítať hodnoty polynómu $T_i \cdot p^{k-1} + T_{i+1} \cdot p^{k-2} \cdots + T_{i+k-1} \cdot p^0 \mod q$ pre každý úsek dĺžky $|P|$. Keďže ale chceme počítať hodnotu hashu pre kanonickú reprezentáciu substringu $T[i : i + k]$, tak si ešte potrebujeme udržiavať permutáciu, pomocou ktorej z tohto substringu dostaneme jeho kanonický tvar.

Keď sa toto okno posunie doprava nastanú dve veci. Ako prvé treba odčítať prvý člen polynómu, zvýšiť všetky exponenty o 1 a nakoniec pripočítať člen reprezentujúci pribudnuté písmeno. Keďže sa zmenilo prvé písmeno nového reťazca, tak sa mohla aj zmeniť permutácia, ktorou z tohto reťazca dostaneme jeho kaninocký tvar.

Táto permutácia je určená pozíciami prvých výskytov jednotlivých písmen. Preto si budeme navyše udržiavať pre každé písmeno frontu s indexmi v aktuálnom okne, na ktorých sa nachádza dané písmeno. Podľa prvých prvkov z týchto front potom usporiadame permutáciu 1 až 26, čím dostaneme hľadanú permutáciu.

Aby sme teraz vedeli zmeniť písmená, ktoré sú koeficientami v polynóme touto novou permutáciou, rozdelíme si tento polynóm na súčet 26 polynómov. Každý z týchto polynómov bude obsahovať rovnaké koeficienty (teda to budú členy prislúchajúce nejakému písmenu).

Keď sa hodnota nejakého písmena zmení v permutácií z a na b, tak stačí polynóm tohto písmena vynásobiť hodnotou $a^-1 \cdot b$. To platí preto, lebo všetky koeficienty majú hodnotu a, teda vynásobením touto hodnotou sa zmenia všetky zmenia na b. Keď potom chceme zistiť, či nastala zhoda, stačí hodnoty týchto polynómov sčítať a porovnať s hashom reťazca P.

Keďže abeceda má konštantnú veľkosť, tak substringy, ktoré hashujeme, vieme posúvať v konštantnom čase. Časová zložitosť je preto $O(|T| + |P|)$. Pamäťová zložitosť je tiež $O(|T| + |P|)$.

Jakub Konc

8. Akademické zbrane

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

Tretia podúloha

Pozrime sa najprv na tretiu podúlohu – teda verziu úlohy, kde a_i je vždy 0. Simulujme si jednu `q` query a pozerajme sa na najvyšší bit výsledku. Ak tento bit nie je nastavený v žiadnom b_i v intervale, určite to musí byť 0. Ak je zapnutý v jednom b_i, určite to chceme takto ponechať, inak by sme si výsledok určite zhoršili. Ten najzaujímavejší prípad ale nastáva, ak je zapnutý vo viacerých. Tu totiž môžeme pre jedno vybrať samotné b_i a pre druhé také číslo, ktoré má tento bit vypnutý, ale všetky nižšie zapnuté.

Teda napríklad keď v našom intervale nájdeme $b_1 = 10 = 1010_2$ a $b_2 = 9 = 1001_2$, tak si pre prvú pozíciu môžeme vybrať samotné $b_1 = 1010_2$ a pre druhú 0111_2. Ich OR potom bude 1111_2, čo je evidentne optimálne.

Teda už máme nejaké riešenie tretej podúlohy – pre každú `q` query si postupne prechádzame bitmi odpovede od najvyššieho, nastavujeme ich podľa toho, či niekde existuje príslušný bit v b_i a keď také dokonca existujú viaceré, tak zvyšok bitov doplníme jednotkami. `u` query vyriešime triviálne – proste si prepíšeme dané b_i.

Toto riešenie má ale časovú zložitosť $O(qn \log A)$, kde A je najväčšie číslo v poli b_i. Toto je ale príliš pomalé na vyriešenie tretej sady.

Zrýchlenie riešenia

Môžeme si všimnúť dve možné miesta na optimalizáciu – zahodiť zo zložitosti logaritmus alebo dokonca zahodiť celé n.

Ak chceme zahodiť logaritmus, môžeme si všimnúť, že celé čo robíme je bitový OR všetkých b_i v intervale. To vieme urobiť triviálne lineárne. A teda okrem toho zisťujeme najvyšší bit, ktorý je zapnutý aspoň v dvoch číslach. Toto sa tiež dá efektívne urobiť troškou bitovej mágie. Už máme premennú O, ktorá si udržiava OR všetkých zatiaľ spracovaných čísel. Zadefinujme si okrem toho D. To vždy pri spracovaní prvku b_i updateneme na $D \,\mathrm{OR}(O \,\mathrm{AND}\, b_i)$. Teda do neho vždy pridáme bity, ktoré sme už niekedy predtým videli ale aj sú nastavené

v aktuálnom čísle, čo je presne čo chceme. Teda nakoniec si v D nájdeme najvyšší zapnutý bit a všetky nižšie bity v odpovedi zapneme.

Takto sme teda dosiahli zložitosť $O(qn)$, čo je ale stále príliš pomalé. Ďalšia optimalizácia by mala ale byť každému skúsenému KSPákovi jasná – máme nejaké updatey a nejaké query na intervale (s asociatívnou kombinačnou funkciou) , takže na to proste hodíme intervaláč[1]. Takto dosiahneme zložitosť $O(n+q \log n)$, ktorá získa dva body.

Späť k pôvodnej úlohe

No a teraz sa ešte potrebujeme zbaviť obmedzenia $a_i = 0$. Na to použijeme jedno veľmi pekné pozorovanie – pozrime sa na spoločný prefix čísel a_i a b_i v binárke. Vieme, že každé číslo v intervale $[a_i, b_i]$ má tiež tento prefix. Teda akúkoľvek hodnotu si vyberieme, tento prefix nedokážeme zmeniť. Nazvime si tieto bity *fixnuté*. No a tu nadchádza ďalšie pozorovanie – ingorujúc prípad $a_i = b_i$, ktorý vieme triviálne vyriešiť, sa tieto čísla musia na prvej cifre po prefixe líšiť. To ale znamená, že v tomto intervale je aj toto b_i, aj číslo, ktoré má na tejto líšiacej pozícií nulu a nižšie jednotky. Tiež ak je na niektorej nižšej pozícií jednotka, vieme ju vynulovať a nižšie dať jednotky.

No a tak sa nám rysuje riešenie: najprv si pre každé i spočítame najdlhší zhodný prefix a_i a b_i. Následne si hodnoty b_i rozdelíme na tento prefix a zvyšok. Tieto dve časti si uložíme do intervaláča. Pri vykonávaní query si najprv vyORujeme tieto *fixné* prefixy. A následne už len pokračujeme v riešení prípadu $a_i = 0$, akurát začíname s O nastaveným na OR *fixných* prefixov a pracujeme len na zvyškoch.

Ešte by sme si mohli myslieť, že sa nám v zložitosti opäť objaví $\log A$ pri hľadaní najdlhšieho zhodného prefixu, avšak tomu sa vieme jednoducho vyhnúť tak, že a_i a b_i vyXORujeme a nájdeme najvyšší nastavený bit, čo moderné CPU zvládnu v konštantnom čase.

Toto riešenie má zložitosť $O(n + q \log n)$ a získa plné body. Pamäťová zložitosť je $O(n)$.

[1] https://www.ksp.sk/kucharka/intervalovy_strom/