Korešpondenčný seminár z programovania
XXXVII. ročník, 2019/20
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola letnej časti

1. Polovica párna

Marcel a Sabinka
(max. 12 b za popis, 8 b za program)

Poďme si najprv povedať, kedy a prečo vieme, resp. nevieme na danú vakcínu nájsť vírus. Prečo by sa vlastne taký vírus nemal dať urobiť? Veď vieme predsa na obe strany dať (takmer) ľubovolné čísla, nie? V sampli ale poľahky nájdeme nejaké prípady, ktoré spája to, že sa na daný vírus nedá nájsť vakcína. Čo majú spoločné?

Aby sme na to prišli, tak musíme použiť trošku matematiky. Totiž, nech sčítame akýkoľvek počet párnych čísel, výsledok bude vždy párny. Pri nepárnych číslach záleží na počte sčítavaných čísel. Ak sčítame nepárny počet nepárnych čísel, výsledok bude nepárny (napr. $3+3+3=9$), ale ak sčítame párny počet nepárnych čísel, tak výsledok bude párny ($3+3=6$).

Z toho vyplýva, že ak dostaneme úlohu vypísať nepárny počet párnych čísel, a nepárny počet nepárnych čísel tak, aby sa ich súčet rovnal, tak sa to nedá. Pretože, kým súčet párnych bude vždy párny, súčet nepárnych bude nepárny (lebo ich je nepárny počet). Takto sme našli prípad, kedy sa vakcína na vírus nedá nájsť. Čo ale ostatné prípady? Ak máme vypísať párny počet párnych a párny počet nepárnych čísel, tak sú súčty oboch častí párne, a nie je nič, čo by nám bránilo vytvoriť dve skupiny s rovnakým súčtom.

Teraz, keď už vieme, kedy sa to dá, a kedy nie, je riešenie úlohy jednoduché. Ak sa to nedá, teda ak $n/2$ je nepárne, tak vypíšeme "nie".

Vo všetkých ostatných prípadoch vypíšeme "ano", a vypíšeme takú postupnosť čísel, ktorá spĺňa podmienky zo zadania. Napríklad začneme postupne vypisovať párne čísla, presne toľko, koľko treba ($n/2$), a počítame si ich súčet. Potom vypíšeme o jedno nepárne menej ($n/2-1$), ako treba (zase si počítame súčet), a na koniec vypíšeme také číslo, koľko je rozdiel súčtov (prečo bude nepárne sme si povedali v prvom odstavci).

Vzorový kód si nepočíta súčty, ale robí to trochu trikovejšie, a to tak, že ak je prvé párne 2, a prvé nepárne 1, tak s každou dvojicou (jedným párnym, jednám nepárnym) sa zvýši rozdiel medzi súčtami o 1.

2. Atypické zasadnutie

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

Riešenie hrubou silou

Prvé riešenie, ktoré si ukážeme, využíva jednoduchý prístup, kedy si pre každý možný posun stola spočítame, koľko ľudí bude sedieť na správnom mieste. Toto vieme spraviť jednoducho, stačí ak ku každému odborníkovi n_i pripočítame otočenie stola j. Ak sa následne $n_i + j$ rovná i, vieme, že odborník sedí na správnom mieste. Úlohou je taktiež otočiť stôl o čo najmenej pozícií. Preto v prípade, že máme k otočení s rovnakým a zároveň najväčším počtom správne usadených odborníkov, musíme vybrať najmenšie otočenie do ľubovolnej strany. To vieme spraviť jednoducho. Vieme, že ak sme stôl otočili o viac ako polovicu miest, otočenie do druhej strany je efektívne. Stôl nám tak stačí otáčať iba do jednej zo strán. Stôl tak otočíme o najviac $\lfloor \frac{n}{2} \rfloor$ miest.

Časová zložitosť tohto riešenia je kvadratická. Pre každú pozíciu pri stole sme museli prejsť všetkých sediacich pri stole. Keďže pozícií je n, rovnako ako sediacich, časová zložitosť je $O(n^2)$. Pamäťová zložitosť tohto riešenia je $O(n)$ keďže sme si potrebovali pamätať všetky pozície zo vstupu.

Vzorové riešenie

Pri riešení hrubou silou sme si postupne pre každú pozíciu pri stole vypočítali, koľko ludí sedí na správnom mieste. Tento krok vieme zjednodušiť. Každý odborník pri stole buď sedí na svojom mieste, alebo vieme otočiť stôl práve dvoma spôsobmi tak, aby sedel na svojom mieste. Ako sme si avšak ukázali pri riešení hrubou silou, stôl nám stačí v skutočnosti otáčať iba do jednej strany. Nepotrebujeme preto pre každého odborníka uvažovať každú možnú pozíciu otočenia stola. Pre každého odborníka nám stačí vypočítať, o koľko treba otočíť stôl, ak má daný odborník sedieť na správnom mieste. Vytvoríme si tak pole veľkosti n, kedy pre každé otočenie stolu dostaneme počet odborníkov sediacich na správnom mieste. Následne nám stačí iba nájsť také otočenie, ktoré má najväčší počet odborníkov na správnom mieste a zároveň otáčame stôl o čo najmenej miest do ktorejkoľvek strany.

Časová zložitosť tohto riešenia je lineárna vzhľadom na počet odborníkov n. Pre každého odborníka sme v konštantnom čase vypočítali otočenie stola tak, aby sedel na správnom mieste. Následne sme prešli všetky možné otočenia. Keďže odborníkov aj možných otočení je n, časová zložitosť je tak $O(n)$. Pamäťová zložitosť je taktiež lineárna vzhľadom na počet obdorníkov n. Počiatočné pozície odborníkov si pamätať nemusíme, stačí nám pamätať si pole s počtom odborníkov pre jednotlivé otočenie stolu. Keďže možných otočení je n, pamäťová zložitosť je $O(n)$.

Jano

3. Na dušu liek

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

Aby sme neboli zmätení z označení, vyjasnime si to hneď na začiatku. Zvislá os bude x-ová, vodorovná bude y-ová. Políčko na súradniciach (x, y) je teda v riadku x a stĺpci y.

Hrubá sila

Najpriamočiarejšie riešenie, je riešenie hrubou silou. Skúšame postupne všetkých $(r+1)\cdot(s+1)$ možných súradníc vysielača. Pre každú možnosť prejdeme cez všetkých $r \cdot s$ domov aby sme spočítali celkovú depresiu.

Časová zložitosť takéhoto riešenia je $O(r^2 s^2)$. Pamäťová zložitosť je $O(rs)$ (pre každý dom si pamätáme počet obyvateľov).

Rozdelíme si to

Celkovú depresiu pre vysielač na súradniciach (x, y) si vyjadríme takto:

$$\sum_{i=1}^{r}\sum_{j=1}^{s} c_{ij}\left((8x-8i+4)^2+(8y-8j+4)^2\right)$$

Stačí si uvedomiť, že toto môžeme rozdeliť na dve časti:

$$\left(\sum_{i=1}^{r}\sum_{j=1}^{s} c_{ij}(8x-8i+4)^2\right)+\left(\sum_{i=1}^{r}\sum_{j=1}^{s} c_{ij}(8y-8j+4)^2\right)$$

Na x-ovú časť depresie, ktorá závisí iba od x a nezávisí od y a na y-ovú časť, ktorá závisí iba od y. Vďaka tomu môžme hľadať najlepšie x a y nezávisle od seba. Inak povedané, nemusíme skúsiť všetky dvojice (x, y), namiesto toho najprv nájdeme najlepšie x (ktoré minimalizuje x-ovú časť depresie), a potom najlepšie y.

Takto sme si zlepšili časovú zložitosť na $O(rs^2 + r^2 s)$.

Vzorové riešenie

Čo sa týka x-ovej časti depresie, nerozlišujeme, v akom je dom stĺpci. Záleží nám len na čísle riadka. Všetky domy v tom istom riadku môžme sčítať dokopy. Keď si predpočítame súčty riadkov, budeme môcť počítať x-ovú časť depresie rýchlejšie. Využijeme teda prefixové súčty[1]. Označme si R_i súčet počtov obyvateľov domov v riadku i. x-ová časť depresie teda je:

$$\sum_{i=1}^{r} R_i(8x-8i+4)^2$$

Pre jedno x zrátame x-ovú časť depresie jedným cyklom v čase $O(r)$. Musíme vyskúšať r možností, hľadanie najlepšieho x nám teda tentoraz zaberie $O(r^2)$

Samozrejme, rovnaký trik spravíme aj pri počítaní y-ovej časti depresie. Predpočítame si teda aj súčty stĺpcov.

Časová a pamäťová zložitosť

Časová zložitosť takéhoto riešenia je $O(r^2+s^2)$. Pamäťová, ak si budeme šikovne počítať súčty riadkov a stĺpcov už pri načítavaní vstupu, je $O(r+s)$ (nemusíme si držať celé dvojrozmerné pole domov v pamäti).

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

Marek

4. Dezinfekcia

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

Ak si to zhrnieme, v úlohe nám ide o to, aby sme odstraňovali skaly, ktoré susedia s nádržou, kým nádrž nezaberá aspoň požadovanú plochu. Komplikácie s dezinfkeciou vieme jednoducho vyriešiť tak, že skaly susediace s nejakou dezinfekciou si označíme ako zakázané políčka, s ktorými nemôžeme nič robiť. Tým pádom sa nám neskôr nemôže stať, že by sme vypustili dezinfkeciu do pitnej vody.

Pomalé riešenie

Ako teda odstraňovať skaly? Ako zistiť, ktoré skaly môžeme odstraňovať? V podzemí sa má nachádzať práve jedna súvislá nádrž. To znamená, že každá skala, ktorej odstránením zväčšíme nádrž, sa musí nachádzať hneď vedľa niektorého políčka nádrže.

Prvoplánovým riešením by mohlo byť niečo také, že by sme dookola prehľadávali všetky políčka mapy a pri každom políčku skaly by sme zisťovali, či sa nachádza vedľa nádrže, a v pozitívnom prípade by sme skalu odstránili. To by sme mohli opakovať, kým by nádrž nenabudla dostatočné rozmery.

Toto riešenie by určite fungovalo, keďže vždy by sme pridávali iba políčka susediace s nádržou a postup by sme opakovali až kým by sme nedostali dostatočne veľkú nádrž.

V najhoršom prípade by sme pri požadovanej ploche s museli s-krát prehľadať celú mapu o veǩosti mn. Z toho nám vychádza časova zložitosť $O(smn)$.

Čo sa týka pamäte, vystačili by sme si s pamätaním si iba políčok mapy, čo nám dáva $O(mn)$.

Vzorové riešenie

Zrejme tušíte, že opakované prehľadávanie celej mapy neznie ako správna cesta. Skaly na odstránenie potrebujeme "hľadať" o čosi sofistikovanejšie. Pripomínam, že nádrž musí byť práve jedna, čo vieme do reči informatiky preložiť aj ako "všetky políčka nádrže musia tvoriť jeden súvislý graf". Súvislý graf je graf, ktorý sa skladá z práve jedného komponentu, čiže medzi ľubovoľnými dvoma vrcholmi v ňom existuje cesta.

Hm... Keby sme si existujúcu nádrž predstavili ako jeden podgraf, tak všetci jeho susedia by boli skaly, ktoré môžeme odstrániť v danom momente. Ak odstránime takúto susednú skalu, daný vrchol splynie s nádržou a teraz jeho susedia sú susedmi celej nádrže (podgrafu). A tak ďalej by sme pridávali postupne po jednej susednej skale, až kým nemáme dostatočnú nádrž.

Tu vieme teraz použiť prehľadávanie do hĺbky (DFS[2]), alebo do šírky (BFS[3]). Povedzme, že už máme spočítanú veľkosť pôvodnej nádrže. Môžeme začať prehľadávať mapu (graf) od ľubovoľného políčka existujúcej nádrže. Ak počas prehľadávania vidíme políčko nádrže, sme šťastní. Ak nájdeme políčko skaly (ktorá nie je blokovaná kvôli dezinfekcii), môžeme ju odstrániť a tým zväčšiť nádrž o toto jedno políčko. Keď už máme nádrž dostatočnej veľkosti, skončíme prehľadávanie.

Prečo políčka existujúcej nádrže nerátame? Tieto si musíme zrátať ešte pred prehľadávaním. Ak by sme nevedeli, aká veľka bola pôvodná nádrž, nevedeli by sme, o koľko ju potrebujeme zväčšiť. Potom by sa mohlo stať, že počas prehľadávania by sme prehľadali priveľa skál ešte predtým, než by sme prehľadali existujúcu nádrž, čím by sme vytvorili zbytočne veľkú nádrž.

Toto riešenie bude fungovať, pretože každá odstránená skala určite susedila s nádržou, čiže nemôže vzniknúť druhá nádrž, a okolo dezinfekcie máme označené zakázané políčka, čiže sa nám ani nemôže stať, že otrávime obyvyteľov KMP. Tiež určite nebude priveľká, ako už bolo objasnené v predchádzajúcom odseku.

Časová zložitosť BFS je $O(V+E)$, kde V je počet vrcholov grafu a E je počet jeho hrán. V našom prípade je V počet políčok na mape mn a $E \leq 4V$, pretože každý vrchol má v štvorčekovej mape nanajvýš štyroch susedov. To nám dáva časovú zložitosť $O(mn)$.

Pri pamäťovej zložitosti sa nám nič zásadné nezmenilo. Záleží od implementácie, ale vo všeobecnosti použijeme iba zanedbateľné množstvo pamäte pre BFS/DFS. Takže čo sa týka pamäte, zostávame na $O(nm)$, keďže si pamätáme celú mapu.

MichalS

5. Éra krátkych skratiek

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

Zabudnime zatiaľ na obmedzujúcu podmienku v zadaní, že z každého slova musíme do skratky vybrať aspoň jedno písmeno. Keďže medzery sa v skratke nenachádzajú, bez tohto obmedzenia sa nás úloha pýta, koľkými spôsobmi možno dostať skratku – reťazec S ako podreťazec reťazca T. Táto úloha sa dá riešiť jednoducho dynamickým programovaním.

[2] https://www.ksp.sk/kucharka/dfs/
[3] https://www.ksp.sk/kucharka/bfs/

Dynamické programovanie označuje techniku, kedy si problém vieme rozdeliť na menšie, výsledky pre menšie problémy si zapamätať a potom ich rovno používať, nepočítať ich znova. Naše podproblémy budú zodpovedať tú istú otázku, ale len pre menšie prefixy S a T. Konkrétne: označíme si $dp[i][j]$ počet spôsobov, ako vybrať z prvých i znakov T podpostupnosť zhodnú s prvými j znakmi S. Tieto hodnoty budeme počítať od menších hodnôt i a j k väčším.

Ako vypočítať hodnotu $dp[i][j]$, ak poznáme hodnoty pre menšie i a j Chceme teda získať prefix S dĺžky j ako podpostupnosť prefixu T dĺžky i.

Ak je $j = 0$, potom zrejme existuje práve jedna možnosť, ako vybrať prázdnu podpostupnosť z niečoho, a to práve tá, že nevyberiem žiadny znak.

Ak je $i = 0$ a $j \neq 0$, potom nie je možné z prázdneho prefixu T neprázdnu podpostupnosť.

Ak sa j-ty znak S zhoduje s i-tym znakom T [4], potom tento znak mohol byť z T vybraný a ostáva nám vybrať prvých $j-1$ znakov S z o 1 kratšieho prefixu T. Už vieme, koľkými spôsobmi to ide: $dp[i-1][j-1]$. Inou možnosťou je, že sme tento znak z T nevybrali (nejaký rovnaký sme v T vybrať museli, ale skôr). To znamená, že sa pokúšame vybrať prvých j znakov S ako podpostupnosť z o 1 kratšieho prefixu T, čo ide $dp[i-1][j]$ spôsobmi. Každá možnosť konštrukcie podpostupnosti spadá do práve jedného z týchto prípadov, a teda bude započítaná práve raz.

Ak sa j-ty znak S nezhoduje s i-tym znakom T, musia všetky spôsoby, ako vyrobiť žiadaný prefix, vyzerať tak, že používajú iba prvých $i-1$ znakov T.

Vďaka pamätaniu si predchádzajúcich hodnôt dp vieme každú ďalšiu hodnotu dp spočítať v konštantnom čase, teda celková časová zložitosť je úmerná počtu dvojíc i a j, teda $O(|S| \cdot |T|)$.

Ako by sme vedeli upraviť tento algoritmus, aby počítal len možnosti, kedy z každého slova zoberieme aspoň jedno písmeno?

Parametre i, j dynamického programovania vlastne zodpovedajú akýmsi stavom. Stav je napríklad, že mám prefix S dĺžky j a prefix T dĺžky i. V našej pôvodnej úlohe tiež môžeme nájsť nejaké stavy. Konkrétne je stavom to, že máme prefix S dĺžky j, prefix T dĺžky i a buď sme už z aktuálneho slova (z toho, ktorému patrí i-ty znak T) vybrali nejaké písmeno, alebo nie. Týmto sa stavový priestor zväčší iba dvojnásobne, takže to časovú zložitosť nepokazí.

Nech teda $dp[i][j][N]$ značí stav, kedy máme prefix S dĺžky j a prefix T dĺžky i, pričom z aktuálneho slova sme ešte nepoužili žiaden znak a $dp[i][j][P]$ značí, že sme už nejaký znak slova použili. N a P sú len symboly pre lepšie pochopenie. V implementácii môžeme použiť hodnoty 0 a 1 alebo mať dve polia, dp_N a dp_P.

Ak je i-ty znak T písmeno, do stavu $dp[i][j][N]$ sa dá dostať jedine zo stavu $dp[i-1][j][N]$. Ak by sme totiž použili daný znak, dostali by sme sa do stavu s P. To však nenastalo, a teda máme o 1 kratší prefix.

Do stavu $dp[i][j][P]$ sme sa mohli dostať viacerými spôsobmi: - i-ty znak T sme nevybrali, museli sme vybrať nejaký znak predtým, čiže sme prišli zo stavu $dp[i-1][j][P]$, - i-ty znak T sme vybrali (samozrejme, len ak je zhodný s j-tym znakom S) a už predtým sme vybrali nejaký iný znak slova, teda sme prišli zo stavu $dp[i-1][j-1][P]$, - i-ty znak T sme vybrali a bol to prvý vybraný znak slova, žiaden predošlý sme nevybrali, prišli sme zo stavu $dp[i-1][j-1][N]$.

Ostáva poriešiť hranice medzi slovami. Tie sa jednoducho riešia vtedy, ak je aktuálny (i-ty) znak T medzera. Vtedy začína nové slovo, takže počet spôsobov, ako mať vybraný nejaký znak nového (aktuálneho) slova ($dp[i][j][P]$) je nula. Počet spôsobov, ako nemať vybraný znak nového slova ($dp[i][j][N]$) je rovný $dp[i-1][j][P]$, teda hodnota pre o 1 kratší prefix T s použitím nejakého znaku posledného slova. Všimnime si, že nezapočítavame $dp[i-1][j][N]$, pretože to zodpovedá situácii, kedy sme z predošlého slova nič nevybrali.

Počet možností pre skúmaný stav získame ako súčet počtov možností pre stavy, z ktorých sa do aktuálneho vieme dostať.

Odpoveďou je potom odpoveď pre celý reťazec S a celý reťazec T, pričom aj z posledného slova sme museli nejaké písmeno vybrať. Je to teda hodnota $dp[|T|][|S|][P]$.

Časová zložitosť riešenia je stále $O(|S| \cdot |T|)$, pretože máme len dvakrát viac hodnôt ako v zjednodušenom prípade a každú hodnotu vieme vypočítať v konštantnom čase.

Pamäťová zložitosť je rovnaká, ale vieme dosiahnuť aj zložitosť $O(|S|)$, pretože pri výpočte hodnôt $dp[i]$ nám stačí poznať hodnoty $dp[i-1]$, teda stačí si pamätať dva stĺpce, predošlý a aktuálny, tabuľky dp.

Ralbo

6. Mandarínkové kráľovstvo

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

Pozrime sa na to, aké podmienky musí vybraná postupnosť zamestnancov spĺňať, aby sa najväčšia mandarínka dostala z ľubovoľnej začiatočnej pozície na poslednú. Pre jednoduchosť sa pozrime na prvú mandarínku

[4] Znaky indexujeme od 1, pretože napr. prefix dĺžky 2 končí druhým znakom, ale ten je na indexe 1 v zero-based stringu.

a ako ona bude cestovať v rámci vybranej množiny zamestnancov.

Vždy, keď mandarínka prejde rukami nejakého zamestnanca, tým že je najväčšia sa ocitne na konci intervalu, ktorý má tento zamestnanec na starosti. Preto precestuje nejakou podpostupnosťou zamestancov až na koniec. Každý ďalší zamestnanec v tejto postupnosti sa prekrýva v pracovnom intervale s intervalom predchádzajúceho zamestnanca. Zároveň na to, aby druhý zamestnanec mandarínku vôbec zodvihol, musí jeho interval končiť až po konci predchádzajúceho intervalu. Posledný zamestnanec, ktorého rukami mandarínka prejde ju dá na posledné miesto. Všimnime si tiež, že táto postupnosť, ktorou prejde najväčšia mandarínka, je istým spôsobom minimálna. To znamená, že keby iba táto časť zamestancov prišla do práce, tak ľubovoľná mandarínka z ľubovoľnej pozície by sa dostala na koniec.

Toto sa dá dokázať tak, že sa paralelne pozrieme na dva prípady: Na to, keď je mandarínka na začiatku na prvej pozícii a na to, keď je na pozícii P. Ak mandarínka z pozície P neskončila na konci na správnej pozícii, tak musela skončiť skôr. Teda mandarínka z pozície 1 ju musela predbehnúť. Pozrime sa na moment, kedy ju predbehla. Zistíme, že v tom momente úradoval jeden konkrétny zamestnanec. Ten mandarínku začínajúcu na pozícii 1 presunul na koniec svojho intervalu, ale mandarínku začínajúcu na pozícii P nie. Toto je však spor, keďže mandarínka z pozície P je tiež najväčšia v tom svojom prípade.

Z tohto vyplýva, že hľadáme najkratšiu možnú postupnosť zamestnancov takú, aby sa prvá mandarínka vedela dostať na poslednú pozíciu. Musí teda spĺňať to, že v postupnosti sú zamestnanci, kde každý ďalší sa v intervale práce prekrýva s predchádzajúcim a kde aj konce intervalov v poradí rastú.

Na to aby sme našli túto podmnožinu intervalov môžeme použiť dynamické programovanie. A to takým spôsobom, že si pre každú pozíciu budeme pamätať, na koľko najmenej skokov medzi intervalmi sa najväčšia mandarínka vie dostať z prvej pozície na túto. Na začiatku sa mandarínka vie dostať na prvú pozíciu na 0 skokov a na ostatné pozície na nekonečne veľa skokov (inými slovami, nevie sa tam dostať). Potom prichádzajú postupne zamestnanci. Každý zamestnanec môže najväčšiu mandarínku na koniec svojho intervalu preniesť až potom, čo najväčšia mandarínka do tohto intervalu vstúpi. Teda na koniec svojho intervalu do poľa zapíše minimum čísel nájdených v intervale plus jedna.

Priamočiara implementácia tejto myšlienky je taká, že si tieto hodnoty ukladáme do poľa. Potom pre každého zamestnanca nájdeme minimum intervalu ktorý mu prislúcha. Následne, ak je toto číslo menšie ako to, ktoré tam máme aktuálne napísané, tak ho prepíšeme. Na nájdenie minima môžme potrebovať spraviť $O(n)$ krokov. Preto celkovo môže byť časová zložitosť až $O(mn)$.

Lepším riešením môže byť napríklad použiť intervalový strom[5]. Ten nám umožní zistiť minimum intervalu v čase $O(logn)$. Tým sa nám zlepší časová zložitosť na $O(mlogn+n)$. Pamäťová zložitosť bude kvôli zamestnancom a intervalovému stromu $O(n+m)$.

Dano

7. Istota čistoty

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

Riešenie hrubou silou je jednoduché. Pre každý koberec môžeme mať `set` a farbu každej lentilky vložíme do `set`u každého koberca, na ktorom sa nachádza. Dostaneme riešenie s časovou zložitosťou $O(nm \log m)$. Ak namiesto `set`u použijeme hešovaciu tabuľku, tak to zlepšíme na $O(nm)$. Pamäť bude $O(n+m)$, ak si zapamätáme iba koberce, lentilky a `set`y.

Vzorové riešenie

Predstavme si, že by vždy ležal menší koberec na väčšom. Uvedomme si, že toto nijak nezmení odpoveď, pretože farba z lentilky presiakne cez všetky koberce pod ňou bez ohľadu na ich poradie. Môžeme teda predpokladať, že koberce skutočne ležia týmto spôsobom.

Keďže sa koberce nepretínajú stranami, tak nám vzniklo niekoľko kôpok kobercov. Každá z týchto kôpok je v podstate strom. Jeho koreňom je spodný, najväčší koberec.

Poďme si rozmyslieť, ako spraviť, aby sme každú lentilku nemuseli zapisovať do každého koberca, ktorý ofarbí, ale iba do jedného. Zapíšeme ju iba do najvrchnejšieho. To nám stačí, pretože pod ním sú už iba väčšie koberce. Keď zapíšeme všetky lentilky, tak nám bude stačit prejsť každý strom a každému vrcholu, čiže kobercu, priradiť farby tak, že zjednotíme farby všetkých jeho synov. Potlačíme teda farby z listov, kam sme ich zapísali, dohora.

Vieme to spraviť rýchlo? Pre každý koberec môžeme mať `set`, do ktorého budeme dávať jeho farby. Keby sme do otca vkladali farby zo synov len tak ledajako, mohlo by sa nám stať, že budeme jednu farbu kopírovať príliš veľa krát. Použijeme teda trik používaný napríklad v dátovej štruktúre `union-find`, kde spájame dve množiny tak, že kopírujeme prvky z menšej tabuľky do väčšej. Predstavme si spájanie jedného zo synov s otcom. Ak

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

je viac farieb v otcovi, prekopírujeme farby zo syna do otca. Ak je viac farieb v synovi, prekopírujeme farby z otcovského `setu` do synovho a prehlásime ho za nový otcov `set`. Takto bude každá lentilka prekopírovaná nanajvýš $\log m$-krát. Na tomto mieste sa vieme zbaviť jedného logaritmu tým, že namiesto `setu` použijeme hešovaciu tabuľku. Táto časť teda bude mať zložitosť $O(m \log m)$.

Ostáva nám vyriešiť iba to, ako zapísať každú lentilku iba do najvrchnejšieho koberca. Toto je celkom známa úloha. Budeme mať intervalový strom. Z obdĺžnikov zoberieme začiatočnú a konečnú zvslú úsečku a o každej si zapamätáme, či je začiatočná, alebo konečná. Tieto úsečky si usporiadame podľa x-ovej súradnice spolu s lentilkami. Usporiadané úsečky pozametáme intervaláčom. Vždy, keď nájdeme začiatočnú úsečku, v intervalovom strome si zapamätáme že daný obdĺžnik je na celom intervale, ktorý pokrýva daná úsečka aktuálne najvyššie. Ináč povedané, na intervale danej úsečky pridáme na `stack` korešpondujúci obdĺžnik. Keď dojdeme ku koncu obdĺžnika, odoberieme daný obdĺžnik zo `stacku` na celom intervale tejto koncovej úsečky. Keď nájdeme lentilku, pozrieme sa do intervalového stromu, ktorý obdĺžnik je najvyššie na danom mieste a lentilku do neho zapíšeme. Keďže súradnice sú veľké, budeme pre intervalový strom potrebovať spraviť kompresiu súradníc, alebo ho stavať dynamicky. Táto časť bude mať zložitosť $O((n+m)\log(n+m))$, čo bude aj výsledná časová zložitosť programu. Pamäťová zložitosť bude $O(n+m)$.

8. Aminokyseliny

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

Stavanie z jednotlivých nukleidov

V prvej sade za dva body máme iba nukleidy ($n = 0$) a tak nemusíme implementovť žiadny algoritmus na vyhľadávania podreťazcov. Stačí nám jednoduché dynamické programovanie: pre každý (súvislý) podreťazec si spočítame aká je minimálna cena za ktorý ho vieme dostať.

Cenu pre podreťazec začínajúci na pozícii i a končiaci na pozícii j ($i < j$) získame iba tak, že sme na začiatok, alebo na koniec prilepili nukleid. Vyskúšame obe možnosti, a keď podreťazce spravcovávame od najkratších, tak získame takto riešenie v čase aj pamäti $O(|T|^2)$.

Krátke sekvencie

V druhej sade už máme nejaké sekvencie, ale je ich málo a sú krátke. Vieme teda rýchlo zistiť, či na nejakú pozíciu sekvencia pasuje. Môžeme upraviť dynamické programovanie pre prvú sadu: okrem pozretia sa na reťazce o prvé/posledné písmenko kratšie, si pre všetkých n sekvencií pozrieme, či mohol podreťazec vzniknúť pridaním sekvencie na koniec/začiatok. Toto nám pre každý podreťazec zaberie $O(n \cdot \max |p_i|)$, čo bohato stačí na získanie ďalších dvoch bodov v druhej sade.

Lepšie riešenie

Prvé dve sady vyžadovali nie až tak trikové dynamické programovanie. Pre posledné dve sady je však pomalé a musíme ho zlepšiť.

Algoritmus nám spomaľujú dva hlavné faktory:

Po prvé, naivné hľadanie, či vieme sekvenciu pridať, alebo nie, je príliš pomalé. Aj keby sme si to pre každú pozíciu predpočítali, bolo by to $O(n|T| \max |p_i|)$ operácií, čo je priveľa. Vedeli by sme to zlepšiť algoritmom KMP[6], takto dostaneme zložitosť $O(n(|T| + \max |p_i|))$.

Druhý problém nastáva s dynamickým programovaním: reťazcov je príliš veľa, takže skúšať pre každý podreťazec všetkých n je pomalé. Pomôže nám nasledovné uvedomenie: sekvencie nie sú veľmi dlhé. Každý podreťazec môže vzniknúť buď pridaním jedného písmenka na koniec/začiatok, alebo prilepením sekvencie s dĺžkou od 1 do 100 (také sú limity na dĺžky sekvencií). Pre každú pozíciu si dostatočne rýchlym algoritmom vypočítame pre každú dĺžku, či existuje sekvencia, ktorá sedí a aká je najlacnejšia sekvencia pre pridanie zo začiatku/konca. Pre každý podreťazec (od najkratších, cenu nulových vieme) si spočítame najmenšiu cenu, za ktorú ho vieme dostať. Následne spočítame cenu pre dlhšie reťazce pridávaním po písmenku/sekvencie na koniec/začiatok. Pre každý podreťazec takto skúsime najviac $2+\max |p_i|$ pridaní, takže časová zložitosť dynamického programovanie je $O(|T|^2 \max |p_i|)$.

S predpočítaním dostaneme časovú zložitosť $O(|T|^2 \max |p_i| + n(|T| + \max |p_i|))$ a pamäťovú zložitosť $O(\sum p_i + |T|^2 + |T| \max |p_i|)$, čo stačí na šesť bodov.

Vzorové riešenie

Aj posledné riešenie, ktoré sme tu doteraz videli, je príliš pomalé pre n okolo stotisíc. Konkrétne, pomalé

[6] https://www.ksp.sk/kucharka/kmp/

je hľadanie všetkých výskytov sekvencií v reťazci T. Našťastie existuje zovšeobecnenie KMP pre vyhľadávanie viac podreťazcov, a to sa volá Aho-Corasick. (Tutoriál napríklad tu[7])

Tento algoritmus už nájde všetky výskyty v čase $O(|T| + \sum p_i)$ a keď ním nahradíme n volaní KMP, dostaneme vzorové riešenie za osem bodov.

Iné riešenie

Existujú riešenia založené na hešovaní. Keďže sekvencie sú krátke, pre každú dĺžku si vieme pamätať samostaný set s hešmi. Následne to, či existuje reťazec dĺžky l, ktorý končí na pozícii i, si vieme zistiť spočítaním vhodného rolling hashu a nazretím do setu.

Toto riešenie tiež vie získať osem bodov. Dôvod, prečo je ako vzorové uvedený Aho-Corasick, je len ten, že na hash ide teoreticky nájsť vstup, na ktorom bude veľa kolízií a tak nepôjde/bude pomalé.

[7] https://codeforces.com/blog/entry/14854