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

Vzorové riešenia 1. kola zimnej časti

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

1. Dvojičky

Vašou úlohou bolo zistiť vek najstaršieho možného páru dvojičiek, a najväčší možný počet dvojičiek na oslave. K správnemu vyriešeniu úlohy si bolo potrebné uvedomiť, že aj fotograf môže byť dvojička, a dokonca môže byť aj súčasťou najstaršieho páru dvojičiek.

Pri hľadaní maximálneho veku páru dvojičiek, sa môže stať buď, že najstarší pár dvojičiek je na fotke, alebo ho tvorí najstarší jednotlivec z fotky a fotograf. Z tohto nám teda vyplýva, že vek najstaršieho možného páru dvojičiek bude maximum z vekov na fotke, bez ohľadu na to, či je tam tento vek dvakrát, alebo len raz.

Celkový počet dvojičiek na oslave ale nemusí byť rovný počtu dvojičiek na fotke. Môže byť o jedna väčší. Tento prípad nastane vždy, keď je na fotke aspoň jeden jednotlivec. Potom fotograf a tento jednotlivec tvoria spoločne dvojičky.

Keď už vieme, čo máme urobiť, pozrime sa na to, ako to urobiť.

Riešenie v čase $O(n)$ a pamäti $O(maximalny_vek)$

Chceli by sme vedieť zistiť počet ľudí daného veku. To vieme urobiť tak, že si najprv vytvoríme pole dĺžky $maximalny_vek$, ktoré celé vynulujeme. Jednotlivé hodnoty na indexoch tohoto poľa budú predstavovať počty ľudí daného veku. Následne prechádzame veky na fotke a zväčšujeme hodnotu na indexe daného veku o jedna. Na konci v celom poli spočítame počet indexov (počet vekov) s hodnotou 2 (počet dvojičiek), a zistíme, či existuje index s hodnotou 1(nejaký jednotlivec na zdvojičkovanie s fotografom). Takto zistíme najväčší možný počet dvojičiek. Maximálny vek dvojičiek nájdeme ako najväčší index v poli s nenulovou hodnotou. Úlohu sme teda vyriešili v časovej zložitosti $O(n + maximalny_vek)$, a pamäťovej zložitosti $O(maximalny_vek)$.

Riešenie v čase $O(nlog(n))$ a pamäti $O(n)$

Najprv si veky z fotografie utriedime[1]. Následne si prejdeme pole a hľadáme miesta, kde majú dva za sebou nasledujúce prvky v poli rovnakú hodnotu (dvojičky). Ak existuje nejaký jednotlivec, teda $pocet_dvojiciek \cdot 2 \neq n$, môže tento človek urobiť dvojičku s fotografom, čiže zvýšime počet dvojičiek o 1. Maximálny možný vek dvojičiek je posledný prvok v poli, keďže pole máme utriedené od najmenšieho po najväčšie.

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

2. Všetkých nás nezastavia!

Najprv sa zamyslime nad tým, ako vyriešiť prípad $n = 2$, $m = 4$, teda pre dvoch ľudí a štyroch mimozemšťanov. Označme si hmotnosti mimozemšťanov $a \leq b \leq c \leq d$. Akými spôsobmi ich môžeme rozdeliť medzi dvoch ľudí? Sú iba tri rôzne možnosti: $a + b$ a $c + d$; $a + c$ a $b + d$; alebo $a + d$ a $b + c$. Všimnite si, že všetky ostatné rozdelenia sú zhodné s jedným z týchto rozdelení.

Rozdelenie $a + b$ a $c + d$ zjavne nie je najlepšie, keďže $c + d$ je väčšie alebo rovné ako každý iný súčet. Stačí nám teda porovnať iba rozdelenie $a + c$ a $b + d$ s rozdelením $a + d$ a $b + c$. V prvej možnosti vidíme, že človek s $b + d$ má najťažší náklad, keďže $b + d \geq a + c$, lebo $b \geq a$ a zároveň $d \geq c$. Ak ale túto hmotnosť porovnáme s tretím rozdelením, zistíme, že tretie rozdelenie je výhodnejšie, keďže $b + d \geq a + d$ (lebo $b \geq a$) a zároveň $b + d \geq b + c$ (lebo $d \geq c$). Najvýhodnejšie rozdelenie týchto štyroch mimozemšťanov teda je $a + d$ a $b + c$.

Teraz skúsme vyriešiť prípad $n = 3, m = 6$; označme si hmotnosti mimozemšťanov $a \leq b \leq c \leq d \leq e \leq f$. Podobným uvažovaním ako vyššie vieme ukázať, že prvý človek musí niesť hmotnosť $a + f$. Zvyšných mimozemšťanov potom musíme rozdeliť rovnakým postupom ako pre $n = 2, m = 4$, teda na $b + e$ a $c + d$.

Takto vieme pokračovať a dokázať, že pre ľubovoľné zadanie $m = 2n$ musíme mimozemšťanov rozdeliť na najťažšieho s najľahším, druhého najťažšieho s druhým najťažším, tretieho najťažšieho s tretím najľahším, atď.

Ako môžeme toto riešenie zovšeobecniť na ľubovoľné $m \leq 2n$? Mohli by sme uvažovať nad tým, koľko ľudí bude mať koľko voľných rúk, a priradiť im najťažších mimozemšťanov, a zvyšok vyriešiť ako vyššie. Pri

[1] funkcia `sort` vo väčšine programovacích jazykov

tomto riešení sa ale dá ľahko zabudnúť na prípad $m < n$. A ako ste sa mnohí sami presvedčili, program ktorý správne rieši iba $m \geq n$, mohol dostať len 2 body. Oveľa jednoduchšie je predstaviť si, že v každej voľnej ruke je mimozemšťan s hmotnosťou 0. Všimnite si, že celkovú hmotnosť, ktorú nesie daný človek to neovplyvní, a môžeme potom využiť vyššie popísané riešenie pre $m = 2n$.

Výsledný program teda najprv načíta hmotnosti m mimozemšťanov, doplní ich o $2n - m$ mimozemšťanov s hmotnosťou 0, utriedi podľa hmotnosti, a vypíše pre každého človeka doteraz nepriradeného najťažšieho a najľahšieho mimozemšťana.

Dano

3. Obojstranná čokotuba

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

Iba jedna strana

Pozrime sa, ako by sa úloha zmenila, keby sme mohli brať čokolády iba zľava. Skúsme všetky možnosti a pre každú overme, či vyhovuje. Najskôr teda overíme, či nám stačí zobrať iba prvú čokoládu. Potom skúsime zobrať prvú a druhú, potom skúsime prvú, druhú a tretiu... Z vyhovujúcich možností zoberieme tú najkratšiu. Ako ale zistiť, či možnosť vyhovuje?

Pre každú testovanú možnosť si zrátame, koľko čokolád ktorej farby obsahuje. Majme pole p veľkosti f so samými nulami. Číslo p_i nám hovorí, koľko čokolád farby i momentálne máme. Ak zoberieme čokoládu farby u, tak ku p_u pripočítame 1. Keď si teda napočítame čokolády v testovanom úseku, môžeme prejsť všetky požiadavky a pre každú požiadavku overiť, či je splnená. Takéto riešenie jednoduchšej úlohy bude fungovať v čase $O(n(n+m+f))$, pretože pre každú z n možností prejdeme nanajvýš n políčok, vynulujeme pole p veľkosti f a prejdeme všetkých m požiadaviek.

Iba jedna strana, rýchlejšie

V pôvodnom riešení sme veľa vecí robili zbytočne. Napríklad, ak sme už poznali počty čokolád v úseku $[1, x]$ a chceli sme zistiť počty čokolád v úseku $[1, x+1]$, nemuseli sme nulovať celé pole p. Úplne nám stačí pridať do poľa p čokoládu na pozícií $x+1$. Všetky ostatné čokolády totiž budú tie isté, ako v starom úseku. Ku $p_{c_{x+1}}$ teda iba pripočítame 1. Teraz, keď nezahadzujeme informácie zo skúmania predchádzajúcej možnosti, vieme každú novú možnosť spracovať v $O(m)$. Možností je stále n, takže časová zložitosť je $O(f+nm)$. Stále totiž pre každú možnosť prechádzame cez všetky požiadavky. Pole p však nastavujeme už iba raz.

Iba jedna strana, ešte rýchlejšie

Asi je celkom očividné, že slabou časťou nášho doterajšieho riešenia je to, že pre každú možnosť prechádzame všetky požiadavky, aby sme overili, či sú splnené. Ako to zlepšiť?

Vieme, že keď sme ešte nezobrali žiadnu čokoládu, tak nie je splnená žiadna požiadavka. Majme teda počítadlo *nesplnene*, ktoré bude hovoriť, koľko máme ešte nesplnených požiadaviek. Na začiatku, je samozrejme nastavené na m. Postupne pridávame čokolády, až niekedy nastane zaujímavá udalosť. Pridaním čokolády farby w sa nám mohol zvýšiť počet týchto čokolád na minimálnu hranicu požadovanú Samom a Janom. To znamená, že požiadavka na čokolády farby w doteraz nebola splnená, ale teraz už splnená je. Môžeme teda od *nesplnene* odčítať 1.

Keď niekedy *nesplnene* dosiahne hodnotu 0, vieme, že už sú všetky požiadavky splnené. Nemusíme teda už ani skúšať brať ďalšie čokolády. Určite by sme tým už nenašli lepšie riešenie, ako máme teraz.

Ako ale zistiť, že nejaká požiadavka doteraz nebola splnená a teraz už je? Na začiatku si spravíme pole t veľkosti f, ktoré nám pre každú farbu povie, koľko jej je žiadanej. To vieme spraviť už počas načítavania požiadaviek. Potom s týmito hodnotami vieme porovnávať hodnoty z poľa p. Ak doteraz platilo, že $p_i < t_i$ a zobratím nejakej čokolády farby i začalo platiť, že $p_i = t_i$, tak požiadavka na farbu i je už splnená. Časová zložitosť teda bude $O(n+m+f)$.

Dve strany, hrubá sila

Jednoduchým riešením pôvodnej úlohy je vyskúšať všetky možnosti pre zobratie prefixu a suffixu a každú takúto možnosť overiť. Podľa toho, ako dobre spravíme overovanie môže mať toto riešenie časovú zložitosť zhruba niekde medzi $O(n^2)$ a $O(n^3)$. Rôzne spôsoby overovania sme si popísali vyššie.

Dve strany, vzorové riešenie

Poďme sa ale pozrieť, ako môžeme riešenie úlohy, kde berieme čokolády iba zľava rozšíriť na obojstrannú verziu.

V jednoduchšej úlohe sme spravili pozorovanie, že ak vieme splniť požiadavky najľavejšími x čokoládami, tak už nemusíme skúšať brať ďalšie, pretože by sme tým určite nezískali lepšie riešenie. Ako toto pozorovanie využiť? Nuž, skúsme nezobrať žiadne čokolády sprava. Zľava teraz zoberieme iba toľko čokolád, koľko nám stačí na splnenie všetkých požiadaviek. Máme teda nejaké riešenie. Možno je najlepšie, možno nie, no určite spĺňa všetky požiadavky. No dobre, zľava už určite nechceme zobrať viac čokolád.

Čo ale sprava? Čo sa môže stať, keď zoberieme naviac ešte jednu čokoládu sprava? Ak to bola čokoláda, ktorej farba sa nijak nevyskytuje v požiadavkách, tak nič neriešime. Zhoršili sme si síce doterajšie riešenie, ale to nás nijak netrápi, pretože výsledok lepšieho riešenia si pamätáme ako doteraz najlepšie nájdené. Čo sa ale stane, ak sa farba najpravejšej čokolády nachádza niekde v požiadavkách na vyváženú stravu? Znamená to, že možno môžeme na ľavú stranu vrátiť čokoládu, ktorú sme odtiaľ zobrali ako poslednú. Môžeme ju vrátiť, ak má rovnakú farbu, ako tá, ktorú sme teraz zobrali sprava. No a aby sme sa z ľavej strany dostali k čokoláde, ktorú sme tam teraz vrátili, museli sme možno prejsť cez nejaké čokolády, ktoré sme vôbec nechceli. Môžeme tam teda vrátiť aj tie. A ľaľa, možno sme práve našli nové najlepšie riešenie.

Pozrime sa na to naopak. Máme riešenie, ktoré berie čokolády iba zľava. Vieme, že všetky požiadavky sú splnené. Skúsme vrátiť čokoládu, ktorú sme zobrali ako poslednú. Tým možno prestala byť splnená nejaká požiadavka. To znamená, že teraz budeme brať čokolády sprava, až kým neopravíme túto pokazenú požiadavku. Tento postup teda začne s riešením, ktoré berie iba čokolády zľava a opakuje dva jednoduché kroky:

1. Vráť na ľavú stranu čokoládu, ktorú si odtiaľ zobral ako poslednú
2. Ber čokolády sprava, kým opäť nie sú splnené všetky požiadavky

Môžeme si všimnúť, že niekedy nám vrátenie čokolády na ľavú stranu žiadnu požiadavku nepokazí. Vtedy teda nemusíme brať žiadne ďalšie čokolády sprava.

Akú to má celé časovú zložitosť? Najskôr nájdeme nejaké riešenie, ktoré berie čokolády iba zľava. To už vieme v $O(n+m+f)$. Potom vždy vrátime jednu čokoládu na ľavú stranu a zoberieme niekoľko, možno aj 0, čokolád sprava. Každú čokoládu teda najviac raz zoberieme a najviac raz vrátime. Bude to teda celé v $O(n+m+f)$. Vieme ale, že m ani f nikdy nebudú väčšie, ako n. Môžeme teda povedať, že časová zložitosť je $O(n)$.

Pamäťová vyzerá ako $O(n+m+f)$. Požiadavky si ale nemusíme pamätať ako zoznam, takže sa dostávame na $O(n+f)$ a stále platí, že $f \leq n$, takže aj o pamäti môžeme povedať, že je $O(n)$.

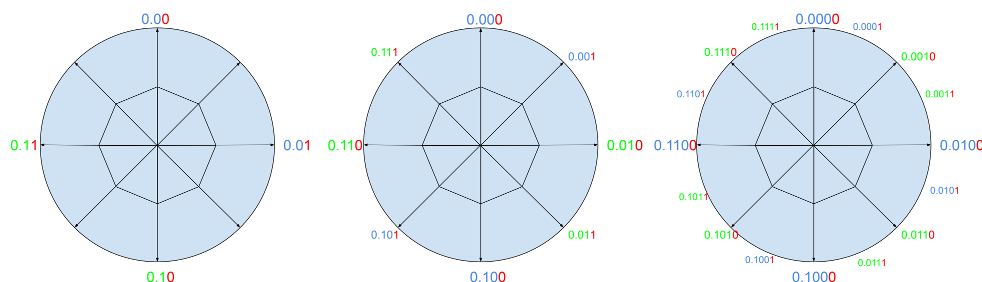
Dávid

4. Jeden smer

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

Prečo binárny zápis a ako to funguje

Binárny zápis riešenia sme zvolili pre to, že na to, aby sme zistili “uhol”, potom nepotrebujeme žiadne výpočty. Stačí nám zistiť, ako daný názov svetovej strany vznikol. Totiž keď zistíme v ktorej polovici je hľadaná strana, vieme jedno desatinné miesto. Keď zistíme štvrtinu, vieme dve. Keď osminu, vieme tri, a tak ďalej. Riešenie sa teda dá ľahko postupne spresňovať a ľahko sa dá vypísať aj ako dlhý reťazec bez nejakých problémov s presnosťou alebo upravovaním zlomkov. Nasleduje ešte jeden obrázok, na ktorom vidno ako toto “spresňovanie” funguje. Všimnite si, že zelené/modré časti vedľa seba sú vždy rovnaké a na červených sa striedajú 0 a 1.



Naivné riešenie

Až 4 body získame za riešenie, ktoré bude postupne vytvárať všetky svetové strany, a s nimi aj uhly, ktoré ku nim prislúchajú. Následne nám stačí sa pozrieť do tohoto zoznamu a vypísať ten správny uhol, bez zbytočných núl na konci. Keď budeme tieto svetové strany vytvárať po úrovniach, ďalšiu úroveň vyrobíme tak, že vždy zoberieme uhol strany vľavo od nej a pripíšeme na koniec 1 (viď obrázok vyššie). Keď toto spravíme, treba ešte

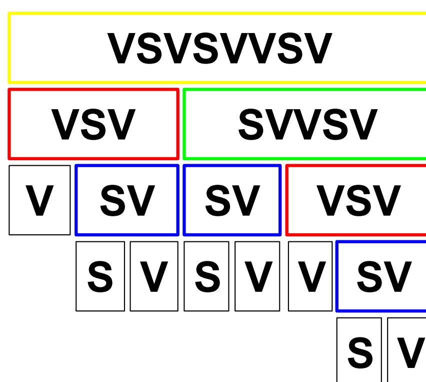
všetkým starým stranám pripísať na koniec 0, aby sa nám aj ďalej zachoval rovnaký počet desatinných miest. Zložitosť takéhoto riešenia je $O(2^u)$ kde u je počet úrovní ktoré chceme vyrobiť. Aby náš odhad bol závislí len od veľkosti vstupu, môžeme ho zvýšiť na $O(2^n)$ kde n je dĺžka vstupu, lebo vieme, že viac úrovní potrebovať určite nebudeme.

Vzorové riešenie

Prvá vec, čo si určite každý všimne je, že už z písmen ktoré máme na vstupe, vieme ľahko určiť štvrtinu, v ktorej je hľadaný smer. Ak sú to napríklad písmená S a V, vieme, že smer je niekde medzi severom a východom. Uhol ktorý hľadáme v binárnej sústave teda začína 0.00 (okrem prípadu kedy je to východ samotný a smer je už 0.01).

Ďalší krok, ktorý by sme mali spraviť je, zistiť či sa nachádzame medzi S a SV alebo medzi SV a V. Keď to zistíme, budeme vedieť, že ďalšia cifra je 0 v prvom prípade resp. 1 v druhom prípade. Najprv však musíme pochopiť, ako toto skladanie strán prebieha. Keď si napíšeme názov nejakej strany, môžeme si ho "uzátvorkovať" tak, ako vznikal až po úroveň, kedy sú v samostatných zátvorkách jednotlivé písmená.

Aby to bolo prehľadnejšie, budeme to demonštrovať na obrázku, na strane $VSVSVVSV$. Každý rámček predstavuje jednu "zátvorku" a pod ním sú naznačené "zátvorky" v ňom, teda dve svetové strany, ktorých zložením vznikol. Farebné odlíšenie nám ešte ukazuje koľko zložení treba na vytvorenie danej srany. Názvy ktoré vznikli bez skladania - základné strany (čierna), strany ktoré vznikli jedným zložením (modrá), dve zloženia (červená), tri (modrá) a štyri (žltá).



Na tomto obrázku si môžeme všimnúť napríklad to, že v jednej farbe sa vždy nachádza to isté (až na čiernu). Čo by sa stalo na ďalšej úrovni? To je veľmi jednoduché. Ďalšia úroveň vznikne tak, že pred tento názov prilepíme to, čo je vľavo alebo to, čo je vpravo od tejto strany. Teraz si stačí uvedomiť, že vlastne tento názov samotný vznikol ako spojenie toho vľavo, a toho vpravo, teda VSV a $SVVSV$. Keďže tieto dve veci už máme v našom obrázku (v druhom riadku), nič "nové" nám nepribudne.

Späť ku našej úlohe. Chceme zistiť, či sa nachádzame medzi S a SV aleo medzi V a SV. Pri ďalšom pohľade na obrázok zistíme, že S sa tam vždy nachádza iba ako priama súčasť SV, ale V sa tam nachádza aj ako súčasť iných strán (napr VSV). Keď sa nad tým ešte chvíľu zamyslíme, ľahko zistíme prečo je to tak. Ak sa nachádzame medzi SV a V a každá strana vzniká spojením okolitých dvoch, nemá sa tam ako dostať S samotné. Teda ak máme viac písmen V ako písmen S, vieme že okrem SV nám tam ostávajú práve písmená V. Máme teda ďalšiu cifru nášho dvojkového desatinného zápisu. Nakoľko sme išli do pravej polovice (pri pohľade zo stredu), je to 1 a teda máme 0.001.

Teraz nám ostáva celý proces opakovať. Vieme že sme medzi SV a V, a chceme zistiť, či vpravo alebo vľavo od VSV. Vieme teda, že celý názov je zložený zo SV a V a chceme zistiť, či je zložený z VSV a SV alebo VSV a V. Na to sa nám oplatí vedieť, koľko krát je tam SV a koľko V. Keď si na začiatku spočítame že $\#_S = 3$ a $\#_V = 5$ (mriežkou označujeme počet výskytov písmena) vidíme, že v ďalšom kroku $\#_{SV} = \#_S$ a $\#_V = \#_V - \#_S$. Teda minuli sme toľko V koľko je S aby sme ich pospájali do SV. Teraz opäť zistíme, čoho je viac a vieme ďalšiu cifru. Máme teda $\#_{SV} = 3$ a $\#_V = 2$, sme bližšie ku SV, teda medzi VSV a SV (uhol 0.0010). Teraz položíme $\#_{VSV} = \#_V = 2$ a $\#_{SV} = \#_{SV} - \#_V = 1$. Sme teda bližšie ku VSV a máme uhol 0.00101. V ďalom kroku nám ostane $\#_{SVVSV} = 1$ a $\#_{VSV} = 1$, teda sme priamo medzi týmito dvoma stranami a finálny uhol je 0.001011 (posledná cifra nám vždy vyjde 1).

Podobnosť tohto postupu s algoritmom na zistenie najväčšieho spoločného deliteľa sama o sebe naznačuje jednoduchosť implementácie.

Časová zložitosť takéhoto riešenia je $O(n)$ od dĺžky slova na vstupe.

Iné riešenie

Z predošlého riešenia si môžeme všimnúť, že už počet písmen S a V, resp. Ich pomer, určuje jednoznačne uhol a je v jednom smere rastúci. Ak teda vieme desatinné dvojkové číslo previesť na názov strany (stačí aj počet písmen S a V), môžeme výsledný uhol binárne vyhľadávať.

MarekC

5. Interpunkčné párovanie

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

Keďže z reťazca bola podľa zadania odstránená práve jedna zátvorka, chceme zistiť, aká. Nájsť chýbajúcu zátvorku vieme tak, že jednoducho prejdeme celý reťazec a spočítame výskyty jednotlivých zátvoriek. Typ zátvorky, ktorého početnosť v reťazci je o 1 menšia ako početnosť jeho párového typu, chýba. Okay, inými slovami, ak sme zistili, že je v reťazci, napríklad, $n\times$ `(` a $(n-1)\times$ `)`, vieme, že chýba `)`.

Pomalé riešenie a overenie správnosti uzátvorkovania reťazca

Dobre, máme chýbajúcu zátvorku. Čo s ňou? Skúsme ju vložiť na všetky možné pozície a spočítajme, pri koľkých týchto pokusoch vznikne správne uzátvorkovaný reťazec. Ako ale vieme zistiť, či je reťazec správne uzátvorkovaný?

Použijeme zásobníkový automat. Každá otváracia zátvorka, napr. `(`, musí mať niekde vpravo od seba svoju zatváraciu, `)`. A to nie len-tak niekde. Teda, zamerajme sa skôr na to, čo musí platiť v intervale medzi týmto párom zátvoriek.

Musí sa tu nachádzať správne uzátvorkovaný (korektný) podreťazec alebo viac takýchto podreťazcov vedľa seba. Napríklad: `( {[]()} [] )`. Medzi vonkajšími `()` sa nachádzajú dva korektné podreťazce: `{[]()}` a `[]`. Z tohto sa nám už črtá akési rekurzívne pravidlo, že každý korektný podreťazec, ohraničený párom zátvoriek (v správnom smere), obsahuje 0 alebo viac korektných podreťazcov vedľa seba. Napríklad `[]` je korektný podreťazec, ktorý obsahuje 0 podreťazcov v sebe; a `{[]()}` obsahuje `[]()`, atď.

Ako teda vieme použiť zásobníkový automat? Povedzme, že prechádzame reťazec po znakoch zľava. Ak nájdeme otváraciu zátvorku, pridáme si ju do zásobníka a ideme ďalej. Ak nájdeme zatváraciu zátvorku, musí to byť práve párová zátvorka k zátvorke na vrchu zásobnika. Ak to platí, otváraciu zátvorku zo zásobnika vyberieme. Znamená to, že uzátvára zátvorku, ktorá sa otvorila ako posledná a nebola ešte uzavretá. Nemôže sa stať, že sa otvorí, napr. `(` a hneď za ňou sa zatvorí `]`. Vtedy by bol reťazec nekorektný.

Napríklad: `( {[]()} [] )` - na začiatku sme našli prvú `(`, potom sme skontrolovali niekoľko ďalších párov, ktoré boli kompletné (otvorili sa a zatvorili bez toho, aby sa "pretínali", napr. `{[}]`, a teraz sme našli poslednú `)`, ktorá je zatváracia k tej prvej, čiže "uzátvára zátvorku, ktorá sa otvorila ako posledná z tých, čo ešte neboli uzavreté". Z toho je jasné, že nás ten stred (podreťazec medzi krajnými zátvorkami) nezaujíma.

Takáto kontrola prejde celý reťazec iba raz. Vyberať z a vkladať do zásobnika vieme v konštantnom čase. Preto vieme povedať, že časová zložitosť jedného overenia reťazcu dĺžky n je $O(n)$.

Pamätať si potrebujeme iba zásobník, ktorý plníme len zo vstupného reťazca, preto jeho veľkosť určite nepresiahne veľkosť vstupného reťazca, čo je rovnako $O(n)$.

Správnosť stringu v tomto riešení ale overujeme pre každú možnú pozíciu chýbajúcej zátvorky, ktorých je $n+1$. To nám dokopy dáva časovú zložitosť $O(n^2)$.

Dá sa to rýchlejšie? Pýtal by som sa, ak by sa nedalo?

Niečo lepšie

Naozaj musíme skúšať doplniť chýbajúcu zátvorku na všetky miesta v reťazci? Vieme nejako zistiť kam najskôr a kam najneskôr môžeme doplniť chýbajúcu zátvorku? Áno.

Skúsme zásobníkovým automatom overiť správnosť reťazca zo vstupu aj keď vieme, že je pokazený. Akú ďalšiu informáciu si z toho ale vieme vziať?

Na vstupe je toto: `[(([]())]{}`. Pustíme na to zásobníkový automat (popísaný vyššie). Ten sa zasekne na tretej zátvorke od konca, keďže na vrchu zásobníka je `(`, ale na vstupe sa nachádza `]`. Má zmysel doplniť chýbajúcu zátvorku niekam za túto `]`?

Uvedomme si: Ak doplníme chýbajúcu zátvorku hocikam za miesto zaseknutia, pri spustení automatu pre overenie sa automat zasekne znova na rovnakom mieste a k novo doplnenej zátvorke sa ani nedostane, lebo chybná časť reťazca zostala nezmenená. Týmto sme vlastne našli hranicu, pokiaľ má zmysel skúšať doplniť chýbajúcu zátvorku. Tak isto vieme spustiť automat aj z opačnej strany a zasekne sa na `(` prvej z ľava.

Tým pádom nám zostalo `[(_(_[_]_(_)_)_]{}`, kde _ znázorňuje miesto, kam má zmysel skúsiť doplniť zátvorku. Teraz môžeme na každé _ skúsiť doplniť zátvorku a overiť korektnosť reťazca.

Je to určite zaujímavá myšlienka, no pri takomto použití je toto len jemná optimalizácia. Časová zložitosť tohto vylepšenia je aj tak stále $O(n^2)$, takže sme veľmi potenciál týchto "hraníc" nevyužili. Ako teda túto informáciu využiť naplno?

Vzorové riešenie

Dokážeme zistiť, či zátvorku môžeme doplniť na dané miesto, bez toho, aby sme museli overiť korektnosť reťazca? Hm... Áno!

Dobre, zásobníkovým automatom sme zistili, kde najskôr a kde najnekôr, mohla vzniknúť chyba. Chyba znamená, že nejaká zátvorka sa otvorila a nezatvorila. Je to jasné? Pri prechode automatom sme narazili na miesto, kde sme chceli zavrieť nejakú zátvorku, ale zistili sme, že ešte iná nebola zavretá, a tým pádom sa to celé pokazilo. (Ak chýba otváracia zátvorka, vieme rovnakú úvahu aplikovať na reťazec z opačnej strany)

Ako vieme tento pokazený reťazec opraviť? Zavrieme túto nezatvorenú zátvorku. To sa zdá byť celkom jednoduché a toto zdanie nie je ďaleko od pravdy.

Ako by sme "zavreli" zátvorku? (Spomeňme si na štvrtý odsek tohto vzoráku) Doplníme zatváraciu zátvorku niekam vpravo od prvej otvorenej v nájdenom intervale (prvá zátvorka bude určite práve opačná k chýbajúcej) tak, aby sa v ich vnútri nenachádzalo nič alebo nejaký korektný podreťazec. Ak by sme do vnútra zátvoriek uzaverli nekorektný podreťazec, celý reťazec by bol nekorektný a preto by daná pozícia doplnenej zátvorky nepripadala do úvahy.

Napríklad, zostal nám interval `[()` a chýba nám v reťazci `]`. Môžeme ju doplniť takto: `[]()` alebo takto: `[()]`, ale nie takto: `[(])`. Ak by bol interval `[[()]`, máme 4 možnosti, kam dať chýbajúcu `]`: `[][()]`, `[[]()]`, `[[()]]` a `[[()]]`.

Všimnime si, že poslednú a predposlednú možnosť rátame ako dve, aj keď v konečnom výsledku vyzerajú rovnako. To preto, lebo nás zaujíma, kam dáme chýbajúcu zátvorku, nie koľko rôznych reťazcov môže vzniknúť. Chápeme, že všetky zátvorky sú unikátne a je teda rozdiel, či doplníme chýbajúcu zátvorku na posledné alebo predposledné miesto.

Je tak zrejmé, že zátvorku chceme dpolniť, buď hneď za jej opačnú (ak tá nie je vo vnútri iných zátvoriek, čo znamená, že je tento podreťazec korektný a pridaním ďalšej zátvorky by sa určite pokazil), alebo medzi nimi nechať nejaký "uzavretý" korektný reťazec/reťazce.

"Uzavretý" korektný reťazec znamená, že ho obkolesujú zátvorky iného typu, ako chýbajúca. To znamená, že doň nemôžeme nič vložiť, lebo by sme ho pokazili.

Poslednou časťou skladačky je tak identifikovanie "uzavretých" korektných podreťazcov v skúmanom intervale. Vieme ľahko zistiť, že takýto podreťazec začína tým, že prečítame otváraciu zátvorku iného typu, akého je chýbajúca. Ako potom zistiť, pokiaľ až tento uzavretý podreťazec siaha? Vieme zase použiť zásobníkový automat... Vieme, že podreťazec je korektný. Tým pádom končí, keď sa zásobník automatu vyprázdni.

Takto dokážeme jedným prechodom intervalu nájsť, a teda aj spočítať, miesta, kam je možné doplniť chýbajúcu zátvorku.

Zložitosti

Pre nájdenie hraníc intervalu potrebujeme pustiť zásobníkové automaty z oboch strán. Vkladanie a vyberanie zo zásobníka má konštantnú časovú zložitosť, takže časová zložitosť nájdenia hraníc pri n znakoch je $O(n)$. Nájdenie a spočítanie vhodných miest trvá jeden prechod nájdeného intervalu, čo je tiež $O(n)$. Preto aj výsledná časová zložitosť je $O(n)$.

Potrebujeme si zjavne pamätať celý reťazec, aby sme ho mohli prejsť z oboch strán. Okrem toho si niekedy pamätáme obsah nejakého zásobníka, ktorý plníme len zo vstupného reťazca, preto jeho veľkosť určite nepresiahne veľkosť vstupného reťazca. Preto hovoríme $O(n)$.

Keďže máme t reťazcov o dĺžkach $n_1, n_2 \dots n_t$, vieme časovú zložitosť odhadnúť pomocou sumy: $O(\sum_{i=1}^{t} n_i)$ a pamäťovú pomocou maxima: $O(MAX(N))$, kde $N = \{n_1, n_2 \dots n_t\}$.

Samo

6. Čakove deti

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

Skúsme najprv vyriešiť jednoduchšiu verziu úlohy. Pokúsme sa len overiť, či sa zo všeobecných práud (ďalej ich budeme volať axiómy) dá odvodiť výrok 0.

Naše riešenie sa bude inšipovať myšlienkami pri prehľadávaní grafov. Budeme si udržovať množinu už dokázaných výrokov a takisto frontu výrokov, ktoré postupne budeme spracovávať (už dokázaných). Ku každému pravidlu si budeme navyše pamätať počet predpokladov (výrokov, ktoré treba mať dokázané aby sme vedeli dokázať nové tvrdenie) ktorésúuž dokázané.

Začneme tým, že označíme všetky axiómy za pravdivé a všetky si ich uložíme do našej fronty. Budeme postupne spracovávať výroky z fronty. Keď budeme spracovávať výrok v, ktorý sa nachádza ako predpoklad v odvodzovacích pravidlách $a_1, ..., a_k$. Pre každý z týchto výrokov zvýšime počítadlo dokázaných predpokladov. A pokiaľ sme už dokázali všetky predpoklady môžeme prehlásiť za pravdivý aj jeho dôsledok (výrok ktorý sme dokázali týmto odvodzovacím pravidlom). A zároveň tento novo dokázaný výrok pridáme do fronty.

Až sa nám raz vyprázdni fronta budeme mať dokázané všetky dokázateľné výroky a teda jednoducho overíme, či medzi ne patrí aj výrok 0.

Všimnime si, že celý algoritmus by mal časovú zložitosť $O(n + \sum_{i=0}^{m-1} k_i)$. Kde n je počet výrokov a k_i počet predpokladov v i-tom odvodzovacom pravidle. Každý výrok totiž len raz pridáme do fronty a každý predpoklad si teda len raz označíme za splnený.

Jednoduché zaklincovanie

Ak chceme vyriešiť pôvodnú úlohu, stačí vyskúšať všetky možné pamäte Čaka, vyfiltrovať, ktoré odvodzovacie pravidlá si vie zapamätať a vždy spustiť vyššie spomínaný algoritmus. Ba čo dokonca viac, my tú správnu najmenšiu vyhovujúcu pamäť vieme binárne vyhľadať. Vtedy časová zložitosť nášho riešenia pri správnej implementácii vedie ku $O((n + \sum_{i=0}^{m-1} k_i) log(max k_i))$

Vzorák

Vzorák namiesto spúšťania prehľadávania zakaždým od znova bude prehľadávať postupne, najprv pomocou najmenších odvodzovacích pravidiel a potom pomocou väčších. Zabezpečíme to tým, že namiesto jednej fronty ich budeme používať $max k_i$. Algoritmus bude podobný ako pôvodný. Axiómy pridáme do fronty číslo 0. Teraz budeme postupne spracovávať frontu ako vyššie. Avšak ak dokážeme všetky predpoklady niektorého odvodzovacieho pravidla, tak namiesto pridania dôsledku do rovnakej fronty pridáme ho do fronty s poradovým číslom podľa počtu potrebovaných predpokladov. Keď sa nám vyprázdni aktuálna fronta skontrolujeme, či sme už dokázali výrok 0. Ak áno, môžeme vypísať poradové číso ukončenej fronty. Ak nie, tak začneme spracovávať frontu s poradovým číslom o jedna väčším.

Celé to bude mať časovú zložitosť $O(n + \sum_{i=0}^{m-1} k_i)$. Keďže podobne ako vyššie, každý výrok bude v niektorej fronte maximálne raz (ak už je výrok pravdivý nepridávame ho do fronty aj keď sme ho odvodili znova). Zbytok algoritmu je rovanký ako vyššie. Pamäťová zložitosť bude $O(n + \sum_{i=0}^{m-1} k_i)$, keďže si musíme pamätať všetky odvodzovacie pravidlá a k nim to, koľko z predpokladov už bolo odvodených, a takisto fronty, v ktorých bude dokopy najviac n prvkov (každý výrok maximálne raz).

7. Keď už je to po ceste...

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

Ako vždy, hrubá sila

Chceme vediet počet usporiadaných trojíc bodov (A,B,C) pre ktoré platí, že súčet Manhattanovských vzdialeností medzi A,B a B,C je rovný Manhattanovskej vzdialenosti medzi A a C.

Tak ich poďme všetky vyskúšať - v troch vnorených cykloch prejdeme všetky body, overíme či rovnosť sedí, a ak áno, pripočítame odpoveď.

Počet rôznych trojíc bodov, ktoré vyskúšame, je $O(n^3)$. Nemusíme si pritom pamätať nič okrem samotných bodov a zopár pomocných premenných, máme teda pamäť $O(n)$.

Toto by nám malo stačiť na prvú sadu, kde $n \leq 200$.

Poznámka o súradniciach

Doleuvedené riešenia budú záležať od veľkosti súradníc. Pre zjednodušenie zložitostí si teda môžeme uvedomiť, že ozajstné súradnice nás až tak nezaujímajú, iba ich relatívne poradie. Môžeme si teda x-ové aj y-ové súradnice na vstupe osobitne prečíslovať od 0 po $n - 1$ v $O(nlogn)$, a ďalšia práca záležiaca na súradniciach bodov už bude $O(n)$. Keď sú súradnice menšie ako n, ponecháme ich tak a použijeme notáciu $O(s)$.

Postavíme Manhattan

Kedy platí požadovaná rovnosť pre body (A,B,C)? Kedy je cesta od A do B a B do C rovnako dlhá, ako priama cesta z A do C? Optimálna cesta z A do B, kde BÚNV. A je naľavo hore od C, je nejaká postupnosť krokov dĺžky jedna dole a vpravo, pričom nikdy neprekročíme súradnice bodu C. Keďže táto postupnosť krokov je ľubovoľná (nezáleží v akom poradí tieto kroky spravíme, len aby bol správny počet krokov dole a správny počet vpravo), môžeme pritom navštíviť ľubovoľný bod B ktorý je v obdĺžniku s bodmi A a C ako rohmi.

Pre každú dvojicu bodov by sme teda chceli vedieť odpovedať na otázku: Koľko bodov leží v obdĺžniku určenom týmito dvoma bodmi?

Toto je presne tá otázka, na ktorú vie efektívne odpovedať 2D prefixový súčet, o ktorom si môžete prečítať v kuchárke[2].

Spravíme si teda mriežku núl, na súradnice bodov pripočítame jedna, a spravíme na nej 2D prefixový súčet.

Teraz jednoducho prejdeme všetky dvojice bodov, a pre každú k odpovedi pripočítame počet bodov v obdĺžniku medzi nimi, mínus dva (keďže v obdĺžniku sa nachádzajú aj naše rohové body). Keďže každý platný bod B s rohmi A a C patrí do trojice (A,B,C) aj (C,B,A), nakoniec náš výsledok ešte prenásobíme dvomi.

Naša mriežka musí mať riadok/stĺpec pre každú možnú súradnicu. Naša mriežka teda má veľkosť $O(n^2)$,

Keďže vytvárame mriežku veľkosti n^2, a prechádzame všetky dvojice bodov (pre ktoré spravíme $O(1)$ operáciu v 2D prefixej mriežke), naša časová zložitosť je $O(n^2).

Pamäťová je priamočiaro $O(n^2)$.

Opačná otázka

Limitujúci faktor v našom riešení je, že musíme vyskúšať všetky dvojice bodov (a spočítať koľko bodov je v obdĺžniku medzi nimi).

Vieme túto otázku otočiť naopak, a zistiť pre každý bod, koľko dvojíc bodov tvorí obdĺžnik, v ktorom leží?

Samozrejme že to bola básnická otázka, inak by nebola vo vzoráku, však?[3]

Poďme sa nad ňou teda zamyslieť. Obdĺžnik ktorý obsahuje nejaký bod B musí mať jeden roh nad ním (alebo zarovno), a druhý roh pod ním (alebo zarovno). Nápodobne, musí mať jeden roh naľavo od neho (alebo zarovno), a druhý roh napravo (alebo zarovno).

Ak skúsime tieto štyri podmienky zlúčiť do dvoch bodov, získame len dve možnosti: buď je jeden roh naľavo hore a druhý napravo dole, alebo je jeden roh napravo hore a druhý naľavo dole. Všetky takéto dvojice bodov sú rohmi obdĺžnika, ktorý náš bod obsahuje.

Prejdime teda všetky body, a pre každý vynásobme počet bodov naľavo hore s bodmi napravo dole, a počet bodov napravo hore s bodmi naľavo dole, a tieto dve čísla sčítajme. Úspešne sme započítali všetky obdĺžniky, ktoré obsahujú náš bod! Nanešťastie, niektoré sme započítali viackrát. Sú to práve tie, v ktorých oba body majú jednu zo súradníc rovnakú ako náš bod. Napríklad bod ktorý je priamo nad našim zvoleným, sme započítali v obdĺžnikoch s druhým rohom naľavo dole aj napravo dole, pričom v oboch prípadoch sme pripočítali body ktoré sú priamo pod ním. Musíme ich teraz raz odpočítať. Od našeho výsledku teda ešte odpočítame počet bodov priamo nad našim bodom krát počet bodov priamo pod ním, a nápodobne s bodmi priamo naľavo a napravo.

Všetky tieto otázky sú na počty bodov v obdĺžnikoch, čiže ich hravo zodpovieme 2D prefixovými súčtami v konštantnom čase.

Stále vytvárame mriežku so všetkými súradnicami, čiže $O(s^2)$.

Každý bod teraz spracujeme len raz, pričom sa popýtame na konštantne veľa obdĺžnikov v našich 2D prefixoch, a teda aj časová zložitosť je rovnaká.

Vzorák

Keď však súradnice narastú, máme už problém aj s $O(s^2)$ aj s $O(n^2)$ mriežkou. 2D prefixy nám už nestačia.

Zamyslime sa teda trochu nad informáciou ktorú potrebujeme, aby sme pripočítali k odpovedi cesty prechádzajúce cez daný bod. Potrebujeme vedieť počet bodov presne nad ním a pod ním, počet bodov presne naľavo a napravo od neho, a počet bodov vľavo/vpravo a hore/dole od neho. Ak si teda túto informáciu nevieme získať pre všetky body naraz (postavením 2D prefixov), možno si ju vieme postupne prepočítavať od jedného bodu k druhému? Tento prístup nás v tomto prípade dovedie k riešeniu.

Poďme skúsiť prechádzať bodmi zhora-dole, zľava-doprava, a udržovať si všetku potrebnú informáciu pre bod, ktorý práve spracúvame. Na začiatku, pred spracovaním prvého bodu, sú akoby všetky bod pod nami. Vytvorme si teda pole `dole`, ktoré na súradnici x bude udržovať počet bodov pod bodom, ktorý práve spracúvame, a s x-ovou súradnicou x. Pred tým ako začneme spracovávať si doň zapíšeme všetky body. Nápodobne si vytvorme pole `hore`, ktoré bude udržiavať počet bodov nad bodom ktorý práve spracúvame, ktoré je najprv plné núl.

Postupne ako budeme prechádzať bodmi zhora-dole, si ich budeme odpočítavať z poľa `dole` a pripočítavať do poľa `hore`. Toto nám dá skoro všetku informáciu čo potrebujeme - vieme spočítať počet bodov napr. naľavo dole od práve spracúvaného bodu (x, y) ako súčet poľa `dole` od súradnice 0 po x, a napr. napravo hore ako súčet poľa `hore` od x po maximum zo vstupu. Počet bodov priamo nad/pod našim bodom je priamo `hore[x]`, resp. `dole[x]`.

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

[3] aj to je básnická otázka

Jediné čo nám ostáva je vyriešiť správanie bodov s rovnakou y súradnicou. Jedna možnosť je napríklad spracovávať všetky body s rovnakou výškou naraz - pridať ich `hore`, spočítať požadované obdĺžniky, odpočítať počet bodov priamo vľavo a vpravo (keďže spracovávame všetky v jednom prechode, udržujeme si túto informáciu), a nakoniec ich všetky jednotne odpočítať z `dole`.

Aby sme si polepšili s časovou zložitosťou, naše polia `dole` a `hore` budú musieť vedieť sčítavať čísla v intervale, a meniť hodnoty na jednotlivých súradniciach, v lepšom čase ako $O(n)$ resp. $O(s)$. Použijeme teda nejakú vhodnú dátovú štruktúru, ako Intervalový strom[4] alebo Fínsky strom[5].

Budeme si pamätať naše body a polia zhruba také veľké, akú máme najväčšiu súradnicu – ak použijeme kompresiu, je to n, čiže pamäťová zložitosť je $O(n)$.

Body musíme usporiadať, prípadne spraviť kompresiu súradníc, čo nám zaberie $O(n \log n)$. Následne všetky body prejdeme, a každý konštantne veľa krát pridáme/odpočítame z intervalového stromu, a zakaždým spravíme konštantne veľa súčtov na tomto strome. Všetky tieto operácie zaberú $O(\log n)$, čiže dokopy taktiež $O(n \log n)$.

Baklažán

8. Yvettin Ementál Neschopnosti

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

Začnime jednoduchým pozorovaním: ak pre daný zoznam slov existujú nejaké dvojičky, určite existujú aj také dvojičky, ktoré začínajú rôznymi slovami, aj končia rôznymi slovami (túto vlastnosť budú spĺňať napríklad najkratšie dvojičky – ak by mali rovnaké prvé slovo, mohli by sme toto slovo vynechať a dostali by sme kratšie dvojičky, rovnako, ak by mali rovnaké posledné slovo). Na vyriešenie úlohy nám teda stačí zistiť, či existujú dvojičky, ktorých prvé aj posledné slová sú rôzne. Všetky algoritmy v tomto vzoráku budú hľadať práve takéto dvojičky.

Dvojičky by sme mohli konštruovať nasledujúcim postupom:

1. Nájdeme v zozname dve slová u, v také, že u je prefixom v. Tieto dve slová si zapíšeme do dvoch riadkov (každé do jedného) tak, aby ich začiatky boli zarovnané pod seba. Platí teda, že ak sú dve písmená napísané nad sebou, sú to rovnaké písmená.
2. Kým platí, že naše riadky sú rôzne dlhé, robíme nasledujúci krok:
   2.1. Nájdeme v zozname nejaké vhodné slovo w a dopíšeme ho na koniec kratšieho riadka. Slovo w musí byť také, aby aj po jeho dopísaní platilo, že písmená, ktoré sú pod sebou, sú rovnaké.
3. Keď dospejeme k tomu, že oba riadky sú rovnako dlhé, našli sme dvojičky (slová, ktoré sme zapísali do prvého riadku, tvoria prvú z nich a slová, ktoré sme zapísali do druhého, tvoria druhú).

Pre zoznam slov {`emental`, `mentalne`, `moj`, `moje`, `neschopnosti`, `schopnosti`} by teda prvé tri kroky mohli vyzerať takto:

```
moj
moje

mojemental
moje

mojemental
mojementalne
```

Tento postup ešte nie je deterministický algoritmus – nie je jasné, ako máme zvoliť počiatočné slová u, v, (ak by sme mali viac možností), ani ako máme v druhom kroku voliť slovo w.

Platí však, že ak dvojičky existujú, pri vhodných voľbách u, v a w ich naším postupom vieme zostrojiť. Naopak, ak neexistujú, potom potom sa nám to pri žiadných voľbách u, v a w nepodarí (buď sa zasekneme na tom, že v zozname nie je vhodné slovo, alebo sa nikdy nedostaneme z druhého kroku). Na vyriešenie našej úlohy nám teda stačí zistiť, či existujú vhodné voľby slov zo zoznamu, ktoré by viedli k nájdeniu dvojičiek.

Všetky možnosti volieb u, v a w, vyskúšať nevieme, lebo ich môže byť nekonečne veľa (napríklad pre zoznam slov {`c`, `ca`, `aa`, `ab`, `ba` `bb`} sa náš postup v druhom kroku zacyklí, pričom si vždy môže vybrať z dvoch možných w). Namiesto toho založíme naše riešenie na nasledujúcich pozorovaniach.

To, či je v nejakom momente nášho postupu ešte možné doplniť naše dva riadky na dvojičky, závisí iba na rozdieli našich riadkov, teda na postupnosti písmen, o ktorú je dlhší riadok dlhší. Tento rozdiel si nazvime *stav*. Ak sme napríklad zatiaľ napísali

[4] https://www.ksp.sk/kucharka/intervalovy_strom/
[5] https://en.wikipedia.org/wiki/Fenwick_tree

```
mojemental
moje
```

povieme, že sme v stave „`mental`". Môžeme si všimnúť, že každý stav, od ktorého sa môžeme dostať, je sufixom nejakého slova zo zoznamu. To znamená, že rôznych možných stavov je $O(n)$.

Na celú úlohu sa môžeme pozerať ako na grafový problém: vrcholy sú stavy a hrana zo stavu A do stavu B vedie práve vtedy, keď v zozname existuje slovo w, ktorého napísaním by sme sa dostali zo stavu A do stavu B. Tie stavy, do ktorých sa dá dostať prvým krokom nášho postupu, nazvime *začiatočné*. Naším cieľom je zistiť, či sa z nejakého začiatočného stavu dá dostať do stavu „" (prázdny reťazec), keďže tento stav zodpovedá tomu, že naše dva riadky sú rovnako dlhé.

Prvé riešenie

Na grafe stavov urobíme prehľadávanie do šírky[6]. Graf si nebudeme konštruovať explicitne, ale podľa potreby za behu. Budeme si udržiavať množinu stavov, do ktorých sa dá dostať, ako množinu reťazcov. Okrem toho budeme mať aj frontu stavov, do ktorých sa dá dostať a ešte sme sa z nich nešírili.

Pre dvojicu slov x, y takú, že x je prefix y, budeme notáciou $y - x$ značiť slovo, ktoré dostaneme, keď zo slova y zmažeme prefix x.

Na začiatku sa pre každú dvojicu slov u, v zo zoznamu pozrieme, či u nie je prefixom v. Ak je, pridáme do množiny dosiahnuteľných stavov, aj do fronty, slovo $v - u$. Týmto sme našli všetky začiatočné stavy.

Následne vyberáme stavy z fronty a z každého z nich sa šírime. Keď vyberieme z fronty nejaký stav s, prejdeme celý zoznam slov a hľadáme v ňom také, ktorým je s prefixom, prípadne ktoré sú prefixom s.

- Keď nájdeme nejaké slovo w, ktorého je s prefixom, znamená to, že zo stavu s sa dá ísť do stavu $w - s$.
- Keď nájdeme nejaké slovo w, ktoré je prefixom s, znamená to, že z s sa dá ísť do $s - w$.

V oboch prípadoch sa pozrieme, či tento nový stav (teda $w - s$, alebo $s - w$) už máme v množine objavených stavov. Ak nie, pridáme ho tam a pridáme ho aj do fronty nerozšírených stavov.

Takto pokračujeme, až kým sa nám nevyprázdni fronta, alebo kým neobjavíme stav „" (prázdny reťazec). Dvojičky existujú práve vtedy, ak sme počas prehľadávania objavili stav „".

Tento algoritmus prehľadá graf stavov korektne, keďže ide iba o jemne exotickú implementáciu prehľadávania do šírky.

Zložitosť

Pri hľadaní začiatočných stavov porovnávame každé slovo v zo zoznamu s každým možným slovom u. Slová v zozname sú dokopy dlhé n, pre jedno v teda toto porovnávanie teda zaberie $O(n)$ času. Pri porovnávaní nájdeme pre jedno v nanajvýš k vhodných slov u, pre ktoré vznikne začiatočný stav $v - u$. Každý z týchto začiatočných stavov je dlhý $O(n)$ a treba ho pridať (ak tam ešte nie je) do množiny dosiahnuteľných stavov, ktorá obsahuje $O(n)$ reťazcov dlhých $O(n)$. Ak množinu stavov implementujeme pomocou vyváženého vyhľadávacieho stromu (ako napríklad `std::set<std::string>` v C++), jedno vyhľadávanie/vkladanie do nej bude vyžadovať $O(\log n)$ porovnaní trvajúcich $O(n)$. To znamená, že pridávanie jedného začiatočného stavu (resp. kontrola, že už bol objavený) trvá $O(n \log n)$. Pridanie všetkých začiatočných stavov objavených pri danom v teda bude trvať $O(kn \log n)$. Jedno v teda spacujeme v čase $O(kn \log n + n) = O(kn \log n)$. Všetky v nám budú trvať $O(k^2 n \log n)$.

Pri rozširovaní sa z nejakého stavu s je situácia podobná: musíme porovnať s s každým možným w, čo nám zaberie $O(n)$ času. Pritom nájdeme $O(k)$ hrán, pre každú z nich musíme overiť, či sa nejaký nový stav dlhý $O(n)$ nachádza v množine objavených. To zaberie dokopy $O(kn \log n)$. Na rozšírenie jedného stavu teda minieme $O(kn \log n + n) = O(kn \log n)$ času. Keďže možných stavov je $O(n)$, dokopy sa budeme rozširovať $O(n)$-krát, teda naše prehľadávanie do šírky bude trvať $O(kn^2 \log n)$.

Keďže $k \leq n$, časová zložitosť celého algoritmu je $O((kn^2 \log n) + (k^2 n \log n)) = O(kn^2 \log n)$. Ak by sme množinu stavov implementovali efektívnejšou dátovou štruktúrou, napríklad písmenkovým stromom[7], zložitosť by bola $O(kn^2)$ (keďže v písmenkovom strome vieme slová dĺžky n vyhľadávať/pridávať v $O(n)$). Trochu náročnejšou analýzou sa dá náš odhad zložitosti ešte spresniť na $\Theta\left(n^2 \min\left(k, \sqrt{n}\right)\right)$.

Lepšia reprezentácia stavov

Predošlé riešenie sa dá zrýchliť šikovnejšou reprezentáciou stavov. Vieme, že každý dosiahnuteľný stav je sufixom nejakého slova zo zoznamu. Dá sa teda popísať dvojicou čísel (i, l), kde i je index nejakého slova v zozname a l je dĺžka sufixu.

[6] https://www.ksp.sk/kucharka/bfs
[7] https://en.wikipedia.org/wiki/Trie

Všetky operácie, ktoré so stavmi v našom algoritme robíme, vieme efektívne robiť aj s takouto reprezentáciou. Pri hľadaní začiatočných stavov vždy vieme, z ktorých slov u, v jednotlivé stavy vznikajú. To znamená, že pre ne vieme v konštantnom čase vypočítať ich reprezentáciu. Pri šírení sa z nejakého stavu (i, l) si skonštruujeme príslušný reťazec (sufix i-teho slova dlhý l) a ten porovnávame so slovami v zozname. Vždy, keď nájdeme nejakú hranu, vieme aj v konštantnom čase vypočítať reprezentáciu stavu, do ktorého vedie (keďže vieme, z akého slova daný stav vznikne).

Na reprezentáciu množiny dosiahnuteľných stavov nám stačí pamätať si pre každé slovo t zo zoznamu jedno pole boolovských premenných rovnako dlhé ako t. Na l-tom indexe v tomto poli si budeme pamätať, či je stav zodpovedajúci sufixu slova t dĺžky l dosiahnuteľný. Keďže slová zo zoznamu sú dohromady dlhé n, celková dĺžka týchto boolovských polí bude tiež n. Overenie, či je nejaký stav v množine dosiahnuteľných, teraz vieme urobiť v čase $O(1)$, rovnako aj pridanie stavu do množiny dosiahnuteľných.

Všimnime si, že graf, ktorý teraz prehľadávame, je trochu iný, ako v predošlom riešení. Ak totiž dve slová zo zoznamu majú spoločný sufix, stav zodpovedajúci tomuto sufixu má až dve možné reprezentácie a my s ním pracujeme, akoby to boli dva rôzne vrcholy. Dá sa však ľahko ukázať, že cesta zo začiatočného stavu do stavu „" (prázdneho reťazca) existuje v našom grafe práve vtedy, keď existovala aj v pôvodnom. Navyše, pre náš upravený graf stále platí, že má $O(n)$ vrcholov.

Časovú zložitosť tohto riešenia odhadneme rovnakým spôsobom, ako pri predošlom riešení. Keďže sme výrazne zrýchlili operácie s množinou dosiahnuteľných stavov, časová zložitosť sa nám zlepší na $O(n^2)$.

Rýchlejšie hľadanie hrán

V doterajších riešeniach sme si hrany nášho grafu konštruovali počas prehľadávania, podľa potreby. Skúsme si ich teraz explicitne skonštruovať vopred. Pre každý sufix s každého slova zo zoznamu teda potrebujeme nájsť všetky slová w, také, že s je prefixom w, alebo w je prefixom s. Ak by sme to robili naivne, trvalo by nám to $O(n^2)$. Ide to však aj rýchlejšie, s pomocou Knuth-Morris-Prattovho algoritmu[8][9].

Pre každú dvojicu slov x, w zo zoznamu spustíme KMP s textom x a vzorkou w. Ak je w kratšie než x, možno nájdeme niekoľko výskytov w v x. Každý z týchto výskytov vlastne znamená, že slovo w je prefixom nejakého sufixu s slova x. Príslušné hrany pridáme do grafu.

KMP si počas svojho behu udržiava informáciu o tom, aký najdlhší sufix prečítanej časti textu je zároveň prefixom vzorky. To znamená, že po dočítaní celého slova x budeme vedieť, aký najdlhší sufix slova x je zároveň prefixom slova w. Príslušnú hranu pridáme do grafu. Okrem toho potrebujeme nájsť aj všetky ostatné sufixy x, ktoré sú prefixami w. Tie vieme nájsť v čase lineárnom od ich počtu, s využitím informácie, ktorú sme si predrátali v rámci KMP – stačí nám nasledovať spätné linky (to, čo v texte v Kuchárke voláme funkcia next()).

Takto sme našli všetky hrany zo sufixov slova x, ktoré využívajú slovo w, pričom nám to trvalo $O(|x| + |w|)$ času. Keď tento výpočet urobíme pre každú dvojicu x, w, budeme mať skonštruované všetky hrany v grafe. Dokopy spustíme KMP k^2-krát, pričom každé slovo zo zoznamu bude k-krát v roli slova x a k-krát v roli slova w. Každé slovo y teda do časovej zložitosti prispeje časom $O(k|y|)$. Keďže súčet dĺžok slov je n, celková časová zložitosť konštruovania grafu bude $O(kn)$.

Na hľadanie začiatočných vrcholov môžeme využiť náš čerstvo zostrojený graf. Pre každé slovo v zo zoznamu sa pozrieme na stav, ktorý zodpovedá jeho najdlhšiemu sufixu (teda celému slovu v). Z tohto stavu povedie niekoľko hrán, pričom jedna pôjde do stavu „" (prázdny reťazec). Všetky ostatné hradny vedú do začiatočných stavov. Rozmyslite si, že takto nájdeme všetky začiatočné stavy (dokonca každý dvakrát).

Keďže náš graf má $O(n)$ vrcholov a z každého vedie nanajvýš k hrán, prehľadávanie grafu bude trvať $O(kn)$, teda celý algoritmus nám beží v čase $O(kn)$. Ak by sme chceli odhadnúť zložitosť nášho algoritmu iba v závislosti od n, môžeme využiť, že $k \leq n$, teda náš algoritmus beží v čase $O(n^2)$. V skutočnosti sa však počet slov dá odhadnúť aj tesnejšie: keďže všetky slová musia byť rôzne, bude ich nanajvýš $O(\frac{n}{\log n})$, teda náš algoritmus beží v čase $O\left(\frac{n^2}{\log n}\right)$.

Počet hrán v grafe

Graf, ktorý sme v predošlom algoritme zostrojili, mal $O(n)$ vrcholov a mohol mať až rádovo $\frac{n^2}{\log n}$ hrán. Naozaj sa pritom dajú zostrojiť vstupy, pre ktoré bude mať graf $\Theta\left(\frac{n^2}{\log n}\right)$ hrán. Zdá sa teda, že naše riešenie sa už nedá zlepšiť.

V skutočnosti však všetky "zlé" vstupy, pre ktoré má graf veľa hrán, využívajú fakt, že ak má veľa slov rovnaký sufix a veľa iných slov má taký prefix, v grafe sa pospájajú "každé s každým". Ak prejdeme naspäť ku

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

[9] Ak tento algoritmus nepoznáte, odporúčam v tomto momente prerušiť čítanie tohto vzoráku a prečítať si o ňom niečo. Zvyšok tohto vzoráku predpokladá, že KMP poznáte.

grafu, kde každý reťazec mohol mať iba jeden vrchol (bez ohľadu na počet slov, ktorých je sufixom), najväčší možný počet hrán sa výrazne zníži.

Najprv potrebujeme zmeniť spôsob, ako reprezentujeme vrcholy grafu. Všetky slová zo zoznamu si napíšme odzadu. Nad takýmito odzadu napísanými slovami teraz postavme písmenkový strom. Každý vrchol písmenkového stromu zodpovedá nejakému reťazcu, z ktorého po otočení dostaneme sufix jedného, alebo viacerých slov zo zoznamu. Zároveň každý vrchol písmenkového stromu zodpovedá inému reťazcu. Vrcholy písmenkového stromu nám teda dobre reprezentujú stavy, pričom každý možný stav je reprezentovaný práve raz.

Koľko hrán môže mať náš graf teraz? Hrany si rozdeľme na dva typy:

- *vnútorné hrany* sú také, ktoré vznikli zo stavu s s použitím slova w kratšieho než s a vedú do stavu $s - w$.
- *vonkajšie hrany* sú také, ktoré vznikli zo stavu s s použitím slova w dlhšieho (alebo rovnako dlhého) ako s a vedú do stavu $w - s$.

Keď sa pozrieme na nejaký konkrétny stav s, vedie z neho toľko vnútorných hrán, koľko je v zozname slov, ktoré sú prefixom s. Všetky tieto slová sú určite rôzne (lebo slová v zozname sú rôzne) a tým pádom rôzne dlhé (lebo rôzne prefixy slova s musia byť rôzne dlhé). Keďže slová v zozname sú dokopy dlhé n, rôzne dlhých slov v ňom môže byť nanajvýš $O(\sqrt{n})$. To znamená, že z každého vrcholu vedie nanajvýš $O(\sqrt{n})$ vnútorných hrán. Zároveň platí, že ich z každého vrcholu vedie nanajvýš k. Celkový počet vnútorných hrán je teda $O(n \min(\sqrt{n}, k))$.

Vezmime si teraz nejaké číslo l a pozrime sa na všetky stavy dĺžky l (v hĺbke l v našom písmenkovom strome). Tieto stavy budeme volať *stavy l-tej úrovne*. Počet vonkajších hrán vedúcich zo stavov l-tej úrovne označme o_l. Na to, aby z nejakého stavu s z l-tej úrovne viedla nejaká vonkajšia hrana, musí v zozname existovať nejaké slovo w také, že s je prefixom w. Keďže každé slovo má nanajvýš jeden prefix dĺžky l, každé slovo zo zoznamu môže byť v roli w pre nanajvýš jeden zo stavov l-tej úrovne. Platí teda $o_l \leq k$. Ba čo viac, v roli w môžu byť iba slová dlhé aspoň l. Ak si počet slov zo zoznamu, dlhých aspoň l, označíme ako k_l, platí $o_l \leq k_l$.

Počet vonkajších hrán v našom grafe vieme spočítať ako

$$o_1 + o_2 + \cdots + o_n.$$

Vieme pritom, že platí

$$o_1 + o_2 + \cdots + o_n \leq k_1 + k_2 + \cdots + k_n.$$

Navyše platí

$$k_1 + k_2 + \cdots + k_n = n,$$

keďže každé slovo je zarátané v práve toľkých k_i, koľko má písmen. Vonkajších hrán je teda nanajvýš n. Dokopy má teda náš graf $O(n \min(\sqrt{n}, k))$ hrán.

Ešte rýchlejšie hľadanie hrán

Náš nový graf je redší, bude sa teda dať rýchlejšie prehľadať. Hľadanie hrán nám však stále trvá $\Theta(nk)$ času. To sa dá zrýchliť tým, že namiesto spúšťania KMP k^2-krát, použijeme Aho-Corasickovej algoritmus[10], ktorý bude stačiť spustiť k-krát.

Vo všetkých behoch Aho-Corasickovej algoritmu budeme používať rovnakú množinu vzoriek: všetky slová zo zoznamu. Predrátanie potrebných informácií pre vzorky (písmenkového stromu so spätnými linkami) nám teda stačí urobiť len raz. V tomto momente v našom algoritme vystupujú dva písmenkové stromy – jeden nad obrátenými slovami zo zoznamu, ktorého vrcholy sú zároveň vrcholmi grafu, ktorý budujeme, a druhý nad pôvodnými slovami zo zoznamu, využívaný v Aho-Corasickovej algoritme.

Pre každé slovo x zo zoznamu spustíme Aho-Corasickovej algoritmus s textom x a celým zoznamom slov ako vzorkami. Vždy, keď Aho-Corasickovej algoritmus nájde výskyt nejakého slova v slove x, pridáme do grafu príslušnú vnútornú hranu. Keď Aho-Corasickovej algoritmus dočíta celé x, budeme vedieť, aký najdlhší sufix slova x je prefixom jedného, alebo viacerých slov zo zoznamu. Tento sufix označme t. Nájdeme všetky slová zo zoznamu, ktorých prefixom je t (o chvíľu si ukážeme ako) a príslušné vonkajšie hrany pridáme do grafu. Následne nájdeme druhý najdlhší sufix x, ktorý je prefixom nejakých slov zo zoznamu (v konštantnom čase, nasledovaním spätnej linky v Aho-Corasickovej strome) a urobíme preň to isté. Takto pokračujeme, až kým nenastane jedna z dvoch možností:

[10] https://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_algorithm

1. Prešli sme všetky zaujímavé sufixy slova x. V tomto momente sú všetky hrany, ktoré vedú zo sufixov x, objavené.
2. Prišli sme na nejaký sufix s, ktorý je zároveň sufixom nejakého iného, už spracovaného slova. To znamená, že hrany vedúce zo stavu s, aj všetkých kratších sufixov slova x sú už objavené a spracovávanie slova x môžeme ukončiť.

Je dôležité, že v druhom prípade ihneď prestaneme pracovať. Inak by sme mohli vonkajšie hrany vedúce z jedného stavu objavovať veľakrát a vôbec by nám nepomohlo, že graf, ktorý konštruujeme, je riedky. Pri pridávaní vnútorných hrán tiež môžeme kontrolovať, či pridávaná hrana neviedie zo skôr spracovaného stavu, aby sme sa vyhli duplicitnému pridávaniu hrán (pri analýze zložitosti budeme predpokladať, že to kontrolujeme. Zložitosť by sa však nezhoršila, ani keby sme to pre vnútorné hrany nerobili).

Zostáva vymyslieť, ako pre zadaný reťazec t nájsť všetky slová zo zoznamu, ktorých je prefixom. Reťazcu t zodpovedá vrchol v Aho-Corasickovej strome. Všetkým slovám zo zoznamu, pre ktoré je t prefixom, zodpovedajú nejaké vrcholy Aho-Corasickovej stromu, ktoré sú v podstrome pod t. Stačí nám teda tento podstrom prehľadať (napríklad do hĺbky) a nájsť v ňom všetky vrcholy, ktoré zodpovedajú celému slovu zo zoznamu. Pritom si treba dať pozor na to, že podstrom pod t môže byť veľký, aj keby v ňom bolo iba jedno slovo. V takom prípade si ho nemôžeme dovoliť prehľadávať naivne, lebo by to trvalo veľmi dlho.

Preto urobíme takzvanú kompresiu ciest. Vrchol Aho-Corasickovej stromu budeme považovať za *zaujímavý*, ak má aspoň jednu z týchto dvoch vlastností:

- Končí v ňom nejaké slovo zo zoznamu
- Má aspoň dve deti

Nezaujímavé vrcholy sú teda také, v ktorých nekončí žiadne slovo a majú práve jedno dieťa. Pre každý nezaujímavý vrchol V si na začiatku algoritmu predpočítame, do akého prvého zaujímavého vrcholu U prídeme, ak z vrcholu V pôjdeme po strome nadol. Keďže z nezaujímavých vrcholov sa dá ísť nadol vždy práve jedným spôsobom, vrchol U je jednoznačne určený. Toto predrátanie vieme urobiť v čase $O(n)$, šikovným prehľadávaním do hĺbky. Následne vždy, keď pri prehľadávaní nejakého podstromu Aho-Corasickovej stromu prídeme do nejakého nezaujímavého vrcholu, skočíme rovno do najbližšieho zaujímavého vrcholu. Takto zaručíme, že čas prehľadávania podstromu bude lineárny od počtu nájdených slov, keďže prehľadáme nanajvýš toľko nezaujímavých vrcholov, ako zaujímavých a v minimálne polovici zaujímavých vrcholov končia slová (to vyplýva z toho, že strom má viac listov, než vetvení).

Zložitosť

Na začiatku si potrebujeme vybudovať oba naše písmenkové stromy a predpočítať si na nich všetko potrebné. To nám zaberie $O(n)$ času. Následne spúšťame Aho-Corasickovej algoritmus, ktorý musí postupne prečítať texty s celkovou dĺžkou n. Samotné čítanie mu teda zaberie $O(n)$ času. Okrem čítania slov ale robíme aj iné činnosti: pridávame vnútorné hrany a hľadáme a prídávame vonkajšie hrany. Všimnime si, že všetky činnosti, súvisiace s pridávaním hrán, mali časovú zložitosť lineárnu od počtu nových hrán, ktoré pridali do grafu. Keďže graf má $O\left(n \min\left(\sqrt{n}, k\right)\right)$ hrán, činnosti súvisiace s pridávaním hrán zaberú dohromady $O\left(n \min\left(\sqrt{n}, k\right)\right)$ času. Nakoniec ešte náš graf v čase $O\left(n \min\left(\sqrt{n}, k\right)\right)$ prehľadáme. Celková časová zložitosť nášho algoritmu je teda $O\left(n \min\left(\sqrt{n}, k\right)\right)$. Ak ju vyjadríme bez použitia k, dostaneme $O\left(n\sqrt{n}\right)$.