Přirozeně nám do toho zapadá prázdný úsek s 0 prvky a nulovým součtem, který mezi koncové také patří.

Ze všech koncových úseků si ovšem stačí pamatovat ten, který má největší součet (jen ten může ovlivnit celkové maximum). Takže si budeme udržovat nějakou proměnnou M se maximálním součtem koncového úseku. Pokaždé, když přibude nový prvek, přičteme ho k této proměnné a pokud vyjde $M < 0$, pak M vynulujeme, protože prázdný úsek je výhodnější.

Ejhle, vyšel nám úplně stejný algoritmus :)

Martin „Medvěd“ Mareš

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Diskusní fórum: *https://ksp.mff.cuni.cz/forum/*

Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – jeho SHA1 fingerprint je: `E9:DB:EE:C6:62:BC:14:DE:09:E4:E8:97:DC:36:0E:87:B3:50:B0:01`.

Korespondenční Seminář z Programování

ZAČÁTEČNICKÁ KATEGORIE

31. ročník | **KSP-Z** | **Duben 2019**

Řešení třetí série začátečnické kategorie 31. ročníku KSP

31-Z3-1 Tvůrčí krize

Najprv sa zamyslíme nad tým, kedy riešenie existuje. V celej úlohe nás z každého vstupného riadku zaujíma iba posledných K znakov. Označme si takto orezané riadky zo vstupu, tvorené iba poslednými K znakmi ako *sufixy*. Riešenie bude určite existovať ak sa medzi *sufixmi* nachádza aspoň M rovnakých prvkov. Čo znamená, že rýmy v básni budú na párnych aj nepárnych riadkoch rovnaké. Riešenie môže existovať aj keď rýmy v básni na párnych a nepárnych riadkoch budú rôzne. V tomto prípade musí byť medzi *sufixmi* aspoň $\lceil \frac{M}{2} \rceil$ rovnakých prvkov pre nepárne riadky básne a aspoň $\lfloor \frac{M}{2} \rfloor$ rovnakých prvkov pre párne riadky básne. Pochopiteľne, ak M je nepárne, tak nepárnych riadkov bude o jeden viac ako párnych.

Ostáva nám vyriešiť ako reprezentovať *sufixy*. V Pythone môžeme k tomu použiť slovník, v C++ napr. `std::map`. Ako kľúče si budeme ukladať jednotlivé sufixy a ku každému sufixu ako hodnotu si uchováme zoznam príslušných riadkov zo vstupu.

Ako alternatívne riešenie na reprezentáciu *sufixov* je možné využiť dátovú štruktúru *trie*, o ktorej sa môžete dočítať v našej kuchárke Hledání v textu.[1] V trii si pre každý vrchol reprezentujúci koniec slova budeme pamätať opäť zoznam riadkov zo vstupu, ktoré odpovedajú danému sufixu.

Po spracovaní *sufixov* zo vstupu stačí zistiť, či existujú nejaké sufixy, ktoré spĺňajú podmienky existencie riešenia. Následne vypíšeme príslušné riadky, ktoré odpovedajú sufixom a to v správnom poradí striedavého rýmu ABAB...

Pali Rohár

31-Z3-2 Zámek obrazovky

Kruhy s písmenky ze zadání a jejich spojnice popisují věc, které se v informatice říká *graf* s *vrcholy* (kruhy s písmenky) pospojované *hranami*. Abychom se v tom pocvičili, tak budeme toto názvosloví v řešení používat.

Naším úkolem bylo začít od zadaného počátečního vrcholu a najít v grafu posloupnost vrcholů pospojovanou hranami takovou, že písmenka z vrcholů budou postupně dávat zadané slovo. Schválně jsme nepoužili výraz *cesta*, jelikož v našem případě (jak je vidět i na ukázce v zadání) se mohou vrcholy a dokonce i hrany opakovat, což nesplňuje definici *cesty v grafu*. Takovou posloupnost vrcholů správně nazýváme *sled* (ale to tu uvádíme jenom jako zajímavost pro rozšíření slovní zásoby o grafech).

A jak takový sled najít? Víme, že když začneme z počátečního vrcholu s prvním písmenkem, tak můžeme pokračovat jenom po hranách, které vedou do vrcholů se stejným písmenkem, jako je druhé písmenko zadaného slova. Tak si nějaký z nich vyberme a zkusíme to dál – z druhého vrcholu si vybereme nějakou hranu, která vede do vrcholu se třetím písmenkem a tak dál. Skončit toto procházení můžeme ve dvou případech: Buď se nám povede dojít až na konec slova (a tím jsme objevili řešení), nebo se dostaneme do slepé uličky – do místa, odkud nepokračuje žádná zatím nepoužitá hrana, po které bychom mohli pokračovat do vrcholu se žádaným písmenkem. V takovém případě se zkusíme vrátit o krok zpět a podíváme se, jestli z minulého vrcholu nešlo pokračovat i někam jinam.

Právě popisovaná technika se nazývá *prohledávání do hloubky* a spočívá v tom, že postupujeme jedním směrem stále dál a dál, dokud to lze, a vracíme se zpět, pokud to nelze.

Nejsnáze se dá implementovat pomocí *rekurze* – vyrobíme si funkci, která jako parametr bude brát vrchol a slovo, které má od tohoto vrcholu najít. Funkce pak pro zadaný vrchol projde všechny hrany vycházející z tohoto vrcholu a pokud je na opačném konci hrany vrchol se správným písmenkem, tak se na něj zavolá (se zbytkem slova). Pokud se povede této funkci umístit všechna písmenka, tak pak může zpětně vracet seznam vrcholů, přes který prošla (jenom pozor na to, že bude pozpátku).

Řešení pomocí rekurze v Pythonu stačí na většinu vstupů, ale poslední už je moc velký a přesahuje limit zanoření volání funkcí, které Python dovoluje. Ale každá rekurze se dá snadno změnit na řešení se *zásobníkem*. V podstatě nám stačí pořídit si pole, ve kterém na začátku bude pouze úvodní stav – počáteční vrchol a celé slovo, které máme najít. Pak v každém kroku odebereme ze zásobníku poslední prvek a provedeme to samé, co rekurzivní funkce v předchozím případě – jenom namísto rekurzivního zavolání funkce přidáme vrcholy k pokračování na konec zásobníku.

Tím, že přidáváme na stejný konec pole, ze kterého odebíráme, tak se nejprve „zanoříme do hloubky“ (podobně jako rekurzivní funkce) a teprve při neúspěchu se vracíme do předchozích vrcholů zkoušet jiné hrany. Toto řešení v Pythonu by již nemělo mít na našich vstupech žádný problém získat plný počet bodů.

Pokud bychom však vstupy vytvořili trochu záludněji s velkým množstvím dlouhých slepých uliček, do kterých půjde vstoupit z mnoha směrů, tak by výše zmíněné řešení mohlo v těchto slepých uličkách dlouho uváznout. Potíž je v tom, že přes stejný vrchol bychom mohli zkusit projít mnohokrát se stejným písmenem – a tím, že za tímto vrcholem bude dlouhá slepá ulička, tak každým tímto pokusem strávíme mnoho času, i když pokaždé dostaneme stejný výsledek.

Uděláme tedy poslední trik a to ten, že si budeme pro každý vrchol pamatovat, že jsem ho už pro nějaký dotaz začali prohledávat (že jsme ho již pro danou část slova přidali do zásobníku). Pokud bychom se ho rozhodli přidat znovu, tak to neuděláme, protože každé takové prohledání by dopadlo stejně a jenom by nás to zdrželo. Toto vylepšení sice na naše testovací vstupy není potřeba (tak zákeřní jsme nebyli), ale podobný trik se do bucoucna může velmi hodit.

[1] `http://ksp.mff.cuni.cz/viz/kucharky/hledani-v-textu`

Každý vrchol v tomto případě projdeme nejvýše K-krát (kde K je délka slova), takže časová složitost se v tomto případě dá odhadnout na $\mathcal{O}(K(N+M))$.

Jirka Setnička

31-Z3-3 Stáda hrochů

Stejně jako v předchozí úloze se i tady trochu pocvičíme v práci s grafy a grafovou terminologií. Pokud jste se s grafy ještě nesetkali, nebojte se, nekoušou :-). *Graf* je v informatice struktura, která se skládá z *vrcholů* (nějakých objektů, v našem případě hrochů) pospojovaných *hranami* (vztahy mezi objekty, v našem případě hrana představuje informaci, že se dva hroši z jiného stáda potkali). Hodí se představovat si ho nakreslený, s vrcholy jako body a hranami jako čárami mezi nimi.

Zadání nás vybízí k nabarvení vrcholů grafu dvěma barvami tak, aby žádné dva vrcholy spojené hranou (o takových říkáme, že jsou *sousední*) neměly stejnou barvu, případně zjištění, že to nejde.

Vcelku přirozeně nás může napadnout následující algoritmus: Na začátku si vybereme libovolný vrchol a dáme mu třeba černou barvu. Pak se podíváme na všechny jeho sousedy a nastavíme jim bílou barvu. Následně vezmeme všechny jejich ještě neobarvené sousedy a nastavíme jim zase černou barvu a tak dále, dokud nacházíme ještě nějaké neobarvené vrcholy.

Tento algoritmus má v podstatě správnou myšlenku, ale ještě je v něm potřeba dořešit několik detailů. V první řadě nebude fungovat například pro graf bez hran, protože nabarví nějaký vrchol na černo a hned skončí. Obecně algoritmus nebude fungovat na grafech, které nejsou *souvislé* – nejde se mezi každými dvěma vrcholy dostat po nějaké cestě z hran. Pomůžeme si tak, že ho budeme spouštět opakovaně – dokud bude existovat nějaký neobarvený vrchol, nabarvíme ho na černo a pak z něj spustíme náš algoritmus. Tak vždy obarvíme jednu *komponentu souvislosti* našeho grafu – tak říkáme souvislým částem, na které se náš graf rozpadá.

Druhý problém je, že zatím neřešíme, jak poznat, kdy graf dvěma barvami obarvit nejde. I to snadno spravíme, například tak, že po doběhnutí algoritmu znovu projdeme všechny hrany a ověříme, že jejich krajní vrcholy mají opačnou barvu. Pokud ano, je vše v pořádku, pokud ne, odpovíme, že náš algoritmus graf obarvit neumí. Znamená to ale, že graf obarvit nelze? Až na první vrchol v každé komponentě barvíme vrcholy vždy tak, jak je to nutné – jinými slovy, pro každý vrchol víme, že nemůže mít barvu jinou, než jakou mu právě dáváme. A jelikož jsou komponenty nezávislé a barvy symetrické, znamená to, že pokud obarvení nalezené algoritmem není správné, pak graf obarvit nejde.

Jak náš algoritmus naimplementujeme? Nejjednodušší je pořídit si rekurzivní funkci, která dostane již obarvený vrchol a všem jeho neobarveným sousedům nastaví barvu na opačnou, než má aktuální vrchol, a pak se na každý z nich rekurzivně zavolá. Pokud přitom navíc bude u již obarvených sousedů kontrolovat, zda je jejich barva správná, můžeme se zbavit výše zmíněného druhého průchodu všech hran, a kontrolovat neexistenci obarvení už během barvení.

Rekurzivní řešení může být často snažší na pochopení, ale někdy není žádoucí, protože za rekurzi často platíme drobným zpomalením a některé jazyky (například Python) mají omezenou maximální hloubku rekurze. Pojďme se tedy rekurze zbavit. Místo, abychom se na nově obarvené vrcholy volali, pořídíme si pole, do kterého si budeme ukládat vrcholy, které jsme nabarvili, ale ještě musíme nabarvit jejich sousedy. Na začátku bude v poli jen počáteční černý vrchol. V každém kroku odebereme libovolný vrchol z pole (třeba ten poslední, ten totiž umíme odebírat v konstantním čase) nabarvíme/zkontrolujeme jeho sousedy a ty předtím neobarvené do pole přidáme (opět na konec). Tak budeme pokračovat, dokud pole není prázdné.

Tak komponentu i s N_i vrcholy a M_i hranami zpracujeme v čase $\mathcal{O}(N_i + M_i)$, protože pro každý vrchol i hranu vykonáme konstantně práce. Budeme-li ještě nenavštívené komponenty hledat chytře, bude celková časová i paměťová složitost $\mathcal{O}(N_1 + M_1 + N_2 + M_2 + \ldots) = \mathcal{O}(N + M)$, kde N je počet vrcholů a M počet hran grafu.

Říša Hladůk

31-Z3-4 Pohyb termitů

Tato úloha spočívala v tom rychle najít všechny dvojice termitů, které jsou ve vzdálenosti nejvýše K. Dřevorubeckým řešením je spočítat vzdálenost mezi všemi dvojicemi termitů a vypsat ty, jež jsou menší nebo rovno K.

Takové řešení se dá zkonstruovat snadno pomocí dvou cyklů zanořených v sobě, jenom je potřeba dát pozor na to, abychom každou dvojici termitů vypsali jenom jednou (což můžeme udělat třeba tak, že vnitřní cyklus bude procházet pouze termity s vyšším číslem, než má termit vybraný vnějším cyklem). Časová složitost tohoto řešení je však $\mathcal{O}(N^2)$, což pro větší testovací vstupy už nedostačuje. Jak to udělat lépe?

V dřevorubeckém řešení jsme kontrolovali i dvojice termitů, které od sebe byly velmi daleko – co kdybychom si zkusili nějak odhadnout, kteří termiti mají šanci na to být blízko sebe? Zkusme si termity rozdělit do přihrádek velikých $K \times K$ vyskládaných vedle sebe a zamyslet se, v jakých přihrádkách se může nacházet termit vzdálený od vybraného termita nejvýše K.

Takový termit může být buď ve stejné přihrádce (ale ne všichni termiti ze stejné přihrádky musí být blízko sebe, třeba protější rohy přihrádky jsou $\sqrt{2}K$ daleko od sebe) nebo v nějaké ze sousedících přihrádek (v osmi směrech). V žádné jiné přihrádce už být nemůže – i kdyby zkoumaný termit ležel přímo na okraji přihrádky, tak vzdálenost K vystačí maximálně na okraj sousední přihrádky a jakýkoliv termit v jiné přihrádce už bude určitě vzdálen více, než je K.

Rozdělíme si tedy termity při načtení rozdělit do přihrádek (třeba tím, že si spočteme velikost přihrádky v x a y ose a těmito velikostmi vydělíme souřadnice termita) a pak tyto přihrádky postupně projdeme a zpracujeme. Protože výskyt termitů může být docela řídký, tak je vhodné vytvářet jen přihrádky, ve kterých nějaký termit je – v Pythonu na to můžeme šikovně využít slovník, který budeme indexovat souřadnicemi přihrádek. Tím pádem bude počet přihrádek maximálně rovný počtu termitů na vstupu.

Pro každou přihrádku nám pak stačí zkontrolovat termity v ní navzájem mezi sebou a pak také s termity v osmi okolních přihrádkách (pokud existují). Pro každou přihrádku máme konstantní počet sousedů a přihrádek je nejvýše N, takže provedeme nejvýše $\mathcal{O}(N)$ porovnání přihrádek, kde vždy porovnáme každý bod s každým.

I na tento postup mohou existovat vstupy, kde bude potřeba udělat řádově N^2 kontrol dvojic termitů – například pokud skoro všichni termiti spadnou do stejné přihrádky. V zadání jsme vám ale slíbili, že u všech vstupů bude očekávaný počet dvojic blízkých termitů řádově N. V rámci jedné přihrádky o hraně K může být pouze velmi omezený počet termitů tak, aby všichni byli ve vzdálenosti větší než K od ostatních (zkuste se zamyslet, kolik nejvíc jich dokážete rozmístit), a každý další termit v přihrádce již přidá alespoň jednu dvojici blízkých termitů. Při takto zadaných omezeních nám tedy výše popsaný postup s rozdělením na přihrádky stačí a nebyl problém za něj získat plný počet bodů.

Jirka Setnička

31-Z3-5 Scrabblová

V úloze hledáme takový souvislý podřetězec délky k v řetězci, který má co největší součet hodnot jeho písmen. Ten můžeme nalézt třeba vyzkoušením všech možností. Podřetězec může začínat na řádově n pozicích ve vstupním řetězci. Pro každou z těchto počátečních pozic projdeme následujících k písmen a sečteme jejich hodnoty, čímž zjistíme, jak dobrý je podřetězec začínající na této pozici.

Při zkoušení pozic si v proměnných pamatujeme, na které pozici jsme viděli zatím nejlepší podřetězec a jaká byla jeho celková hodnota. Pokud je podřetězec na aktuální pozici lepší, zapíšeme tuto pozici do proměnné zatím nejlepší pozice a jeho délku do délky zatím nejlepšího podřetězce. Na konci běhu algoritmu budou tyto proměnné obsahovat informaci o celkově nejlepším podřetězci.

Tento přístup bude ovšem poměrně pomalý. Pro řádově n počátečních pozic sečteme k čísel, časová složitost tedy bude $\mathcal{O}(nk)$, což je pro k podobně velké jako n kvadraticky pomalé.

Pozorování: Jak se od sebe liší dva podřetězce, které začínají na sousedních pozicích? Většinu písmen budou mít společnou, konkrétně $k-1$ písmen, vyjímky jsou první písmeno levého podřetězce a poslední pravého. Dále si všimněme, že toto platí pro každé dva sousední podřetězce. Toho můžeme využít pro vytvoření rychlejšího algoritmu.

Vyjdeme z předchozího pomalého řešení. V prvním kroku spočítáme v $\mathcal{O}(k)$ hodnotu podřetězce začínajícího prvním písmenem řetězce. Místo abychom pro následující podřetězec počítali znovu všech k písmen, využijeme už spočítaný výsledek pro aktuální podřetězec, pouze od něj odečteme jeho první písmeno a přičteme k němu poslední písmeno následujícího podřetězce. Využijeme tím toho, že zbylých $k-1$ písmen se nezměnilo. To samé provedeme i pro každý následující podřetězec.

Jak to ale bude rychle? Všimněme si, že k hodnotě každého písmena ze vstupu přistoupíme nejvýše dvakrát: jednou, abychom ji přičetli, podruhé, abychom ji odečetli. Pro k písmen z levého okraje vstupního řetězce a k z pravého okraje dokonce provedeme jen jeden přístup. Pro každé z n písmen vstupu provedeme nejvýše konstantně mnoho operací, celková časová složitost tedy bude lineární.

Na závěr poznamenejme, že lepší než lineární algoritmus pro tento problém existovat nemůže, protože vždy musíme projít celý lineárně dlouhý vstup, jinak bychom mohli nějaký potenciálně nejlepší podřetězec úplně minout.

Kuba Pelc

31-Z3-6 Vyzvednutí výhry

Nabízí se následující poměrně intuitivní algoritmus: začneme úsekem posloupnosti (tak budeme říkat souvislé podposloupnosti), který bude obsahovat jenom první prvek. Úsek budeme rozšiřovat směrem doprava prvek po prvku. Přitom budeme počítat, jaký je aktuální součet prvků v úseku. Také si budeme udržovat průběžné maximum ze součtů, které jsme zatím potkali. A jakmile součet klesne pod nulu, na aktuální úsek zapomeneme a začneme znovu s prázdným.

V Pythonu by to vypadalo takto:

```
x = [ ... ]   # Vstup
s = 0         # Průběžný součet
z = 0         # Aktuální začátek úseku
max = 0       # Dosavadní maximum
maxz = maxk = 0   # Poloha max. úseku
for k in range(len(x)):   # Aktuální konec
    s = s + x[k]
    if s > max:
        max, maxz, maxk = s, z, k
    if s < 0:
        s = 0
        z = k+1
print(max, maxz, maxk)
```

Tento algoritmus evidentně běží v lineárním čase, ale vůbec není jasné, jestli doopravdy funguje. Algoritmus totiž nezkouší všechny úseky, některé drze přeskakuje. Musíme ukázat, že žádný z vynechaných úseků nemůže být maximální.

Nejprve ukážeme, že pokud jsme našli úsek $[z,k]$ se začátkem z a koncem k, který už má záporný součet, nemá smysl zkoušet úseky s pozdějšími konci. To proto, že žádný takový úsek nemůže mít největší součet. Úsek $[x,k']$ pro $k' > k$ totiž můžeme rozložit na úseky $[x,k]$ a $[k+1,k']$. A jelikož součet úseku $[x,k]$ je záporný, pak součet $[k+1,k']$ musí být ještě větší než součet $[x,k']$. Tím pádem přeskočením úseku $[x,k']$ nic nezkazíme.

A teď si uvědomíme, že pokud jsme nějaký úsek ukončili se záporným součtem, nemá smysl zkoušet jiné úseky, které začínají uvnitř něj. Opět nahlédneme, že takové úseky nemohou mít maximální součet. Nechť úsek $[z,k]$ měl záporný součet a $[z',k']$ je nějaký úsek, který začíná uvnitř něj (tedy $z < z' \leq k$). Všimneme si, že úsek $[z,z'-1]$ nemůže mít záporný součet (jinak bychom totiž začátek z už dávno vzdali). Takže pokud úsek $[z,z'-1]$ přilepíme před $[z',k']$, vznikne úsek $[z,k']$ s ještě větším součtem.

Dokázali jsme tedy, že všechny úseky, které náš algoritmus přeskočil, nejsou maximální. Proto jsme museli maximální úsek najít.

Další, prosím ...

Na úlohu se můžeme dívat i jinak. Vymyslíme tzv. *inkrementální algoritmus*. Tak se říká algoritmům, které čtou vstup postupně a v každém okamžiku znají výstup pro zatím přečtenou část vstupu. V našem případě tedy čteme posloupnost prvek po prvku a stále si udržujeme, jaký úsek má největší součet.

Přidáme-li k posloupnosti další prvek, jaké nové úseky se objevily? Každý takový úsek vznikl rozšířením nějakého koncového úseku původní posloupnosti (to je úsek odněkud až do konce posloupnosti). Přibudou tedy nové koncové úseky, které jsou rozšířením předchozích koncových úseků.