Korespondenční Seminář z Programování

ZAČÁTEČNICKÁ KATEGORIE

32. ročník | **KSP-Z** | **Duben 2020**

Řešení třetí série začátečnické kategorie 32. ročníku KSP

32-Z3-1 Tiskařský stroj

První část úlohy je správně nalézt dvojice. Celý text projdeme ve smyčce znak po znaku a v každém kroku si budeme pamatovat současný a předchozí znak. Pokud ani jeden z nich není mezera, tak tyto dva znaky dohromady tvoří dvojici.

Také si musíme u každé dvojice pamatovat počet jejích výskytů a v každém kroku smyčky jej pro nalezenou dvojici o jedna zvýšit. Potřebujeme tedy nějakou datovou strukturu, pomocí které rychle k nějaké dvojici najít číslo počtu dosud nalezených výskytů.

První možností je použít *slovník*, anglicky *dictionary*. Slovník si pamatuje páry (*klíč*, *hodnota*) a když se ho zeptáme na nějaký klíč, tak nám umí rychle vrátit odpovídající hodnotu. V našem případě bychom jako klíč použili dvouznakový řetězec reprezentující dvojici písmen a jako hodnotu číslo počtu výskytů této dvojice.

Druhou možností je použít obyčejné pole. Ze zadání máme zaručeno, že na vstupu se můžou objevit pouze malá písmena anglické abecedy, kterých je 26. Existuje tedy jen $26 \cdot 26 = 676$ různých dvojic písmen. Vytvoříme si tedy pole čísel o 676 prvcích, kde každé číslo odpovídá počtu výskytů nějaké dvojice. Dvojici převedeme na index v poli tak, že se na obě písmena budeme dívat jako na čísla od 0 do 25, kde `a` odpovídá 0, `b` odpovídá 1 a tak dále. Index spočítáme vynásobením hodnoty prvního písmene dvojice dvaceti šesti a následným přičtením hodnoty druhého písmene. (Také si dvojici písmenek můžeme představovat jako dvojciferné číslo v 26kové soustavě.)

Nyní máme počty výskytů každé dvojice v textu a zbývá najít k nejčastějších. K tomu stačí dvojice setřídit sestupně podle počtu výskytů a vypsat prvních k. Může se stát, že existuje více dvojic se stejným počtem výskytů, a poté nemusí být jednoznačné, které dvojice jsou v k nejčastějších. Například text může obsahovat třikrát dvojici `ab`, dvakrát dvojici `cd` a dvakrát `ef` a ptáme se na dvě nejčastější dvojice. Dle zadání si v takovém případě můžeme vybrat libovolně, tedy odpovědi `ab`, `cd` i `ab`, `ef` jsou správně. V praxi můžeme z dvojic se stejným počtem výskytů vybrat třeba ty, co jsou dříve v lexikografickém pořadí, nebo můžeme dvojice vypisovat prostě v pořadí, v jakém nám je vrátil třídicí algoritmus.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z3-1.py`

Kuba Pelc

32-Z3-2 Sářina omalovánka

Úlohu můžeme vyřešit velmi přímočaře kontrolou podmínek pro každý čtvereček zvlášť. Obrázek si načteme do dvojrozměrného pole. Jednou ho projdeme a bokem do druhého pole si schováme pozice a barvy počátečních čtverečků. Pak můžeme jít čtvereček po čtverečku přes celý obrázek, vždy projdeme všechny počáteční pozice, najdeme nejbližší a podle toho vybarvíme. Kód by toho měl vysvětlit více než pár slov:

```
obrazek = # pole řádků tvořených polem písmen

def vzdalenost(x1: int, y1: int,
               x2: int, y2: int) -> int:
    """
    Spočítá manhattanskou vzdálenost mezi
    zadanými body.

    :param x1: Souřadnice X prvního bodu
    :param y1: Souřadnice Y prvního bodu
    :param x2: Souřadnice X druhého bodu
    :param y2: Souřadnice Y druhého bodu
    :returns: Manhattanská vzdálenost bodů
    """

    return abs(x1 - x2) + abs(y1 - y2)

# Najdeme všechny počáteční čtverečky
pocatecni = []
for y,radek in enumerate(obrazek):
    for x, ctverecek in enumerate(radek):
        if ctverecek != '.':
            pocatecni.append((x,y,ctverecek))

# Pro každý čtvereček v obrázku...
for y in range(len(obrazek)):
    for x in range(len(obrazek[y])):
        # ...vybereme novou barvu...
        barva = '.'
        min_vzd = float('inf')
        # projitím všech počátečních čtverečků
        for px, py, pbarva in pocatecni:
            d = vzdalenost(px, py, x, y)
            # Dva poč. čtverečky stejně daleko
            if d == min_vzd and barva != pbarva:
                barva = '.'
            # nebo je nějaký blíže než ostatní
            elif d < min_vzd:
                barva = pbarva
                min_vzd = d
        obrazek[y][x] = barva

# Nakonec vytiskneme výsledek
for radek in obrazek:
    print(''.join(radek))
```

Kolik kroků bude tento výpočet trvat? Pro každé políčko obrázku uděláme tolik kontrol vzdáleností, kolik je počátečních čtverečků (říkejme tomuto číslu P). Takže složitost je $\mathcal{O}(ABP)$.

***

Pojďme si udělat krátkou technickou odbočku. Všimněte si funkce `vzdalenost` v kódu výše. V samotné definici obsahuje informace o tom, že argumenty jsou čísla a že návratová hodnota je také číslo. Toto je volitelné rozšíření Pytho-

nu 3 nazývané *type hints*, česky typové anotace. Samotnou funkčnost programu přítomnost typů nijak neovlivňuje. Hodí se ale na dokumentaci a jako nápověda vývojovým prostředím, kde kód píšete. Ty pak dokáží lépe napovídat.

Druhou netradiční odlišností oproti klasickým zdrojákům k úlohám, které od nás vidíte, je přítomnost dokumentačního komentáře. Ten popisuje, co funkce dělá, a vývojová prostředí ho zobrazují, když funkci napovídají. Dokumentační komentář může být i pouze obyčejný textový popis, co funkce dělá bez struktury. Pokud je ale funkce složitější a komentář delší, hodí se na první řádek napsat krátce a stručně shrnutí. Níže potom více rozepsat detaily o tom, jak přesně funkce funguje. Navíc můžete ještě přiložit strojově zpracovatelný popis jednotlivých argumentů a návratové hodnoty. Formátů, jak toto zapisovat, je více. Ten použitý výše je doporučován v PEP 287[1] a používá ho například IDE *PyCharm* od *JetBrains* nebo generátor dokumentace z kódu *Sphinx*.

Jak typové anotace tak dokumentační komentáře pomáhají v orientaci v kódu. To se může hodit když program vyroste a už ho není možné celý přečíst za pár minut a nebo na něm pracuje více vývojářů. Ale i když se ke kódu vrátíte po roce, tak to může podstatně usnadnit orientaci. A teď už zpátky k úloze.

⋆⋆⋆

Výše jsme si ukázali algoritmus, kterým dokážeme řešení spočítat se složitostí $\mathcal{O}(ABP)$. Dokážeme se ještě zbavit závislosti na P?

Můžeme využít konceptu z prohledávání do šířky. Pokud jste o tomto algoritmu ještě neslyšeli, podívejte se do naší kuchařky o grafech.[2] Fungovat to bude tak, že se z každého počátečního čtverečku začneme rozšiřovat a barvit čtverečky v okolí. Od všech počátečních čtverečků stejně rychle. Můžete si to představit tak, že na obraz na místa počátečních čtverečků ve stejný okamžik nalijeme barvu, která se začne roztékat.

My v programu neumíme dělat více věcí paralelně jako roztékající se barva, ale to nás nemusí zastavit. Můžeme si do fronty zařadit všechny hraniční čtverečky barevných skvrn (na začátku jsou to počáteční body) a postupně se na všechny podívat a zkusit je. Tím vyrobíme novou hranici, kterou si opět řadíme do fronty a opakujeme, dokud je kam se rozšiřovat, dokud něco ve frontě je. Vzorový program můžete nalézt přiložený ...

Všimněte si, že každý čtvereček navštívíme pouze jednou. Jakmile ho jednou navštívíme, přestane být hraniční a už se k němu nikdy nevrátíme. Uděláme přesně tolik kroků, jaká je velikost obrázku. Složitost je nyní $\mathcal{O}(AB)$. Závislosti na P jsme se opravdu dokázali zbavit.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z3-2.py`

Vašek Šraier

32-Z3-3 Akční ceny

Dostali jsme dvě posloupnosti n čísel a máme najít jejich nejdelší společný úsek. Nejprve hledejme společné úseky, které v obou posloupnostech začínají na stejném místě:

2 3 5 7 3 5 7 6 3 3
9 3 5 1 3 5 7 3 5 3

To jde snadno: procházíme obě posloupnosti současně zleva doprava a hledáme první prvek, který se shoduje. Jakmile ho najdeme, počítáme, kolik dalších prvků se také shoduje. Tak najdeme jeden společný úsek a až skončí, stejně budeme hledat úseky další. Celkem to potrvá lineárně dlouho s délkou posloupnosti, tedy $\mathcal{O}(n)$.

Jenže skutečný nejdelší společný úsek 3 5 7 3 5 jsme najít nemohli, protože jeho výskyty v obou posloupnostech „neleží pod sebou“. Vyzkoušíme proto ještě všechna možná vzájemná posunutí obou posloupností vůči sobě. Mezi nimi bude i to posunutí, ve kterém hledaný úsek vyjde pod sebou:

2 3 5 7 3 5 7 6 3 3
9 3 5 1 3 5 7 3 5 3

Jak rychle tento algoritmus poběží? Pro dvě posloupnosti délky n stačí uvážit $2n-1$ posunutí ($n-1$ na jednu stranu, $n-1$ na druhou a nulové posunutí). Pro každé z nich v čase $\mathcal{O}(n)$ najdeme nejdelší úsek zarovnaný pod sebou. Takže celkově strávíme čas $\mathcal{O}(n^2)$.

Dodejme ještě, že existují rychlejší algoritmy – dokonce s lineární časovou složitostí. Ty ale vyžadují pokročilejší teorii, nejspíš o mnoho levelů nad úrovní KSP-Z. Kdybyste byli zvědaví, zkuste si přečíst něco o datové struktuře zvané suffixové pole.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z3-3.py`

Martin „Medvěd“ Mareš

32-Z3-4 Dálnice

Spočítat počet jízdních pruhů půjde nejlépe, když si jízdní pruhy zkusíme naplánovat (už v zadání jste si mohli všimnout, že rozhodně nestačí najít nejužší místo mapky). Tak pojďme Kevinovi s jejich plánováním pomoci.

Budeme se pokoušet plánovat jeden jízdní pruh po druhém, a to tak, abychom právě plánovaným pruhem co nejméně omezili místo pro další pruhy. Jak takovou věc provést? Můžeme zkusit právě plánovaný jízdní pruh „připlácnout“ co nejblíže k levému okraji koridoru. Tím nám vznikne užší koridor (mezi právě naplánovaným jízdním pruhem a pravým okrajem koridoru), ve kterém můžeme opakovat stejný postup do té doby, než již žádný pruh naplánovat nepůjde.

Jak „připlácnout“ jízdní pruh co nejblíže k levému okraji? Začneme od nejlevějšího volného políčka na prvním řádku mapy a pokračujme těsně vedle levého okraje koridoru. Na řádku postupujme co nejvíce doleva a když to již nejde doleva, tak sestupme dolů. Takto ale můžeme dojít do slepé uličky, co potom?

Zadání slibovalo, že strana koridoru je jenom zubatá, ale nikde na mapě nejsou žádné „ostrůvky“ (zkrátka že každá řádka mapy je tvořena nejdříve několika plnými políčky, tak prázdným místem a pak opět plnými políčky). Jediná slepá ulička, do které můžeme vstoupit, je tak vždy horizontální, například tato (`o` vyznačuje již naplánovaný jízdní pruh):

[1] `https://www.python.org/dev/peps/pep-0287/`
[2] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

```
###o.
###o.
#ooo.
#####
```

V takovém případě nám jen stačí popojít směrem doprava na první políčko takové, že má pod sebou volno. Pokud takové políčko neexistuje, není již žádná cesta, jak se dostat na nižší řádek, a můžeme skončit.

Plánování jednoho jízdního pruhu tak můžeme shrnout do toho, že odstartujeme z prvního volného políčka v prvním řádku mapy a na každém řádku mapy:

- Postoupíme, co nejvíce doleva to lze.
- Postoupíme směrem doprava na první políčko takové, že má pod sebou volno. Pokud takové neexistuje, ukončíme plánování tohoto pruhu.
- Posuneme se na další řádek a opakujeme postup.

Při postupování mapou si také budeme značit již použitá políčka, a tak můžeme po nalezení prvního jízdního pruhu stejným způsobem nalézt další. Jízdních pruhů bude maximálně S (počet sloupců mapy), takže nám stačí výše zmíněný postup spustit S-krát.

A kolik času to zabere? Načtení mapy na vstupu zabere čas $\mathcal{O}(RS)$ a poté S-krát spustíme hledání jízdního pruhu. Jedno hledání může zabrat až čas $\mathcal{O}(RS)$, což by vedlo k celkovému času $\mathcal{O}(RS^2)$, ale ne každé hledání se podívá na všechna políčka mapy. Každé políčko v mapě bude označeno za použité nejvýše jednou a v každém řádku se každé hledání podívá nejvýše na jedno políčko, které neoznačí (bude to políčko, za které se nemůže posunout více doleva). Když to spočítáme přes všechna hledání jízdního pruhu, tak nám vyjde čas všech hledání dohromady (a tím pádem i celého algoritmu) $\mathcal{O}(RS)$.

Důkaz funkčnosti algoritmu

Chtělo by asi zdůvodnit, proč takový postup povede k maximálnímu možnému počtu jízdních pruhů. Podívejme se na plánování prvního pruhu – drželi jsme se co nejvíce vlevo tak, že vlevo od nás nezůstávala žádná nepoužitá políčka. Jediná chvíle, kdy jsme se vraceli, byla ta, kdy jsme „couvali“ ven ze slepých uliček – ale tato políčka ve slepých uličkách žádný jiný jízdní pruh použít nemůže. Vlevo od právě naplánovaného jízdního pruhu tak nezůstala žádná políčka využitelná jinými pruhy a levou hranici koridoru jsme tak zmenšili nejméně, jak to jenom šlo.

Pokud vám důkaz z předchozího odstavce nestačí, tak formálněji bychom to mohli dokázat důkazovou technikou sporem – budeme předpokládat, že by mohlo existovat lepší řešení s více jízdními pruhy, ale dokážeme si, že je nejlépe tak dobré jako to naše, a tím dojdeme ke sporu. Uvažme tedy pro spor jiné řešení s více jízdními pruhy. Jakýkoliv jiný jízdní pruh, který by zasahoval do políček použitých naším plánovaným pruhem, by náš pruh zablokoval.

Můžeme tedy z onoho lepšího řešení odstranit jeho první pruh, přidat místo něj náš první pruh a počet jízdních pruhů se nezmění. Navíc jsme určitě nevyužili žádná políčka ležící napravo od prvního jízdního pruhu v hypotetickém ideálním řešení (protože v našem řešení máme první pruh nejvíce vlevo, jak ho lze umístit) a určitě jsme se tak neprotnuli s druhými jízdními pruhy v obou řešeních. Můžeme tak na ně aplikovat stejný postup (a pak obdobně na všechny další pruhy). Tím dojdeme k tomu, že hypotetické ideální řešení umíme celé převést na námi nalezené řešení, kterému se již žádný pruh navíc nevešel. Tím jsme ukázali, že v hypotetickém ideálním řešení již nemůže být prostor na další jízdní pruh, což je spor s tím, že by mohlo být lepší.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z3-4.py`

Jirka Setnička

32-Z3-5 Čtyři kamarádi

V úloze chceme najít takové místo rozloučení kamarádů, že maximum z délek nejkratších cest do jejich domečků bude co nejmenší. K tomu potřebujeme nejprve pro každé políčko nalézt délku nejkratší cesty do domečku každého z kamarádů.

Na město můžeme nahlížet jako na graf. Každé políčko je vrchol a každá dvě sousední volná políčka jsou spojena hranou. Může se nám hodit nějaký grafový algoritmus, konkrétně k řešení této úlohy použijeme *prohledávání do šířky*, které je popsáno v naší kuchařce o grafech[3] a také jsme jej už potkali v úloze 32-Z2-6. Lze jej použít pro spočítání délky nejkratší cesty z počátku do každého dosažitelného vrcholu.

Pro každý vrchol si budeme pamatovat, zda jsme ho už navštívili a jaká je vzdálenost (délka nejkratší cesty) z tohoto vrcholu do počátečního. Na začátku označíme počáteční vrchol za navštívený, jeho vzdálenost nastavíme na nulu a vložíme jej do fronty.

Dokud je fronta neprázdná, opakujeme: z fronty odebereme vrchol, podíváme se na všechny jeho sousední vrcholy, které ještě nejsou označené za navštívené, přidáme je do fronty a zároveň jim nastavíme vzdálenost na hodnotu o jedna vyšší, než je u aktuálně zpracovávaného vrcholu.

Nalezené vzdálenosti budou skutečně délky nejkratších cest. Počáteční vrchol je jediný se vzdáleností nula. Vzdálenost jedna mohou mít pouze sousedé vrcholu se vzdáleností nula a ty všechny nalezneme a označíme touto hodnotou, vzdálenost dva mohou mít jen sousedé vzdálenosti jedna, ty také správně ohodnotíme a tak dále.

Tento postup provedeme pro každého ze čtyř kamarádů. Pokaždé jako počáteční vrchol zvolíme domeček daného kamaráda a nalezené vzdálenosti si zapamatujeme.

Nyní projdeme všechna políčka, pro každé spočítáme maximum ze čtyř vzdáleností a vybereme jako místo rozloučení takové políčko, kde je toto maximum co nejmenší. Budeme ovšem uvažovat pouze políčka, která byla navštívena ve všech čtyřech prohledáváních do šířky.

Je totiž třeba dát pozor na to, že prohledávání do šířky navštíví pouze vrcholy, které jsou dosažitelné z počátečního. Mohlo by se stát, že jeden domeček bude od ostatních oddělen zdmi. Poté úloha nemá řešení a žádné vyhovující místo rozloučení v předchozím kroku nenalezneme.

Kuba Pelc

32-Z3-6 Světelný hlavolam

Máme-li jen jedno nebo dvě tlačítka, musíme spoléhat na to, že už jsou rozsvícená. Jinak s nimi nic nenaděláme. Podobně

[3] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

pokud budou tři, musí být buď všechna rozsvícená, nebo zhasnutá.

Budou-li tlačítka čtyři, rozeberme, co s nimi umíme dělat podle toho, kolik jich ještě nesvítí:

- 4: Zmáčkneme libovolná tři a tím dostaneme případ, kdy nesvítí jedno...
- 1: Stiskneme naše jediné nesvítící tlačítko a dvě ze svítících a už nesvítí dvě.
- 2: Stiskneme ta dvě, která svítí a jedno z těch, co nesvítí a najednou nesvítí tři.
- 3: Máme vyhráno, stačí stisknout ta tři, která nesvítí.

S $n = 4$ si tak umíme poradit vždy, jenže tím i se všemi vyššími n. Pokud bude nesvítících tlačítek víc než 4, můžeme jednoduše po trojicích rozsvěcet zhasnutá tlačítka dokud nebudou zhasnutá 4 nebo méně, pak volbou čtyř tlačítek zahrnujících všechna zhasnutá už můžeme k úloze přistupovat jako k případu $n = 4$.

Jak je vidět i na příkladu `01011010` ze zadání, v některých případech lze učinit až o tři kroky méně, ale celkem snadno jsme se dozvěděli, že (kromě případů $n \leq 3$) si poradíme s libovolnou kombinací rozsvícených tlačítek.

Tento postup jde jistě naprogramovat s lineární časovou složitostí.

Na závěr dodejme, že existuje i systematičtější způsob, jak řešit úlohy tohoto druhu. Najdeme kombinaci kroků, která způsobí změnu stavu právě jednoho tlačítka: u této úlohy můžeme pro tlačítka 1234 zmáčknout 123, pak 134, a nakonec 124. Tím se změní stav tlačítka 1 a „pomocná“ tlačítka 234 se vrátí do původního stavu. A jakmile dokážeme změnit jedno tlačítko, můžeme tento postup postupně aplikovat na všechna nesvítící tlačítka. Jen je potřeba si uvědomit, že pro $n < 4$ nemáme dost pomocných tlačítek, takže musíme rozebrat malé případy ručně.

Martin Koreček

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`.