PRASK
VII. ročník, 2019/20
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola zimnej časti

1. Permoníci

vzorák napísal(a) Roman
(max. 100 b za riešenie)

K tejto úlohe sme pripravili videovzorák[1].

2. Rekreačný cyklista Kubko

vzorák napísal(a) Misko
(max. 100 b za riešenie)

Sada úloh 1 – Krátke cesty

Podúloha (a)

Táto podúloha sa dala vyriešiť pomerne jednnoducho a to tak, že sme začali v `A` a pozreli sa na všetkých susedov `A`. Potom sme sa pozreli na všetkých susedov týchto susedov. Z tých sme vybrali len tých, ktorý susedia s `D`. Takto sme prešli všetky možnosti ako vieme začať v `A`, a prejsť 3-krát. Preto sme určite našli všetky také cesty.

začiatok	1. sused	2. sused	susedí s `D`
`A`	`B`	`A`	nie
`A`	`B`	`D`	nie
`A`	`B`	`E`	áno
`A`	`E`	`A`	nie
`A`	`E`	`B`	áno
`A`	`E`	`C`	áno
`A`	`E`	`D`	nie

Cesty z `A` do `D`, s dĺžkou 3							
začiatok	`A`						
1. sused	`B`	`E`					
2. sused	`A`	`D`	`E`	`A`	`B`	`C`	`D`
susedí s `D`	nie	nie	áno	nie	áno	áno	nie

Existujú teda 3 také cesty a to: `A-B-E-D`, `A-E-B-D` a `A-E-C-D`.

Podúloha (b)

Túto podúlohu budeme riešiť pobodne ako predošlú. Pre každý začiatok sa zase pozrieme na všetkých susedov, potom na ich susedov, a potom na ich susedov, čo sú vlastne konce. Všetky cesty ktoré majú ako začiatok `A` sú:

začiatok	1. sused	2. sused	3. sused
`A`	`B`	`A`	`B`
`A`	`B`	`A`	`E`
`A`	`B`	`D`	`B`
`A`	`B`	`D`	`C`

[1] https://www.youtube.com/watch?v=LB2KaYRYRMY

začiatok	1. sused	2. sused	3. sused
A	B	D	E
A	B	E	A
A	B	E	B
A	B	E	C
A	B	E	D
A	E	A	B
A	E	A	E
A	E	B	A
A	E	B	D
A	E	B	E
A	E	C	D
A	E	C	E
A	E	D	B
A	E	D	C
A	E	D	E

Keďže všetkých ciest dĺžky 3 je pomerne veľa, môžeme si pomôcť a spočítať ich jednoduchým programom:

Listing programu (Python)

```
# pre jednoduchost oznacime miesta A az E cislami 0 az 4

# pre kazdu dvojicu zaciatku a konca pocet ciest medzi nimi, na zaciatku 0
cesty = [[0 for j in range(5)]for i in range(5)]

# pre kazde miesto zoznam jeho susedov
susedia = [
    [1, 4],
    [0, 3, 4],
    [3, 4],
    [1, 2, 4],
    [0, 1, 2, 3]
]

# pre kazdy zaciatok
for zaciatok in range(5):
    # pre kazdeho 1. suseda
    for sused1 in susedia[zaciatok]:
        # pre kazdeho 2. suseda
        for sused2 in susedia[sused1]:
            # je jeden koniec
            for koniec in susedia[sused2]:
                cesty[zaciatok][koniec] += 1


print(cesty)
```

Pre každý začiatok a koniec vieme nájsť počet ciest dĺžky 3 medzi nimi v tabuľke.

z/do	A	B	C	D	E
A	2	5	3	3	6
B	5	4	3	7	7
C	3	3	2	5	6
D	3	7	5	4	7
E	6	7	6	7	6

Podúloha (c)

Cestu dlhú 6, z `A` do `A`, ktorá má v strede `B`, môžeme rozdeliť na 2 cesty, a to `A-B` a `B-A`. Z predchádzajúcej podúlohy vieme, že počet ciest dĺžky 3 z `A` do `B` a aj z `B` do `A` je 5. Každú z ciest `A-B` môžeme spojiť s každou `B-A` a vytvoria jedinečnú cestu z `A` do `A` cez `B`. Preto je tento počet rovný súčinu počtu ciest `A-B` a `B-A`, teda $5 \cdot 5 = 25$.

Podúloha (d)

Opäť máme cesty dlhé 6, no tentokrát bez medzizastávky. Keďže počty ciest dĺžky 3 poznáme, aj tu by sa nám hodilo rozdeliť si cestu na dve, s dĺžkou 3. Ako medzizastávku si teda postupne vyberieme každé miesto. Potom vieme vypočítať počet ciest `A-C` ako súčet ciest `A-C` s medzizastávkou `A`, a ciest s medzizastávkou `B`, `C`, `D` a `E`. Tento súčet teda je $2 \cdot 3 + 5 \cdot 3 + 3 \cdot 2 + 3 \cdot 5 + 6 \cdot 6 = 78$.

Sada úloh 2 – Tabuľky

Podúloha (a)

Máme zistiť počet ciest z `A` do `E` s medzizastávkou v `C`. Počet ciest dĺžky 6 z `A` do `C` už poznáme z predošlej sady. Zvyšok cesty má dĺžku 9. Zo zadania teda poznáme aj počet takýchto ciest (`C-E`). Počet ciest `A` do `E` s medzizastávkou v `C`, vypočítame ako súčin týchto hodnôt, teda $78 \cdot 3\,182 = 248\,196$.

Podúloha (b)

Tu použijeme podobný princíp ako v predošlej sade. Pre nejaké dve miesta, napr. `A-B` sa dá počet ciest medzi nimi dĺžky 12 vypočítať tak, že využijeme medzizastávku vo vzdialenosti 9. Počet ciest dĺžky 9 poznáme zo zadania, ostane nám druhá časť cesty s dĺžkou 3. Počty takýchto ciest sme už zistili v predošlej sade. Pre jednu medzizastávku vieme vypočítať počet ciest keď vynásobíme počet ciest dĺžky 9 z `A` po medzistávku a počet ciest dĺžky 3 z medzizastávky do `B`. Keď potom tieto počty sčítame, dostane počet všetkých ciest medzi dvoma miestami.

Opäť si môžeme pomôcť jednoduchým programom, ktorý to zráta za nás.

Listing programu (Python)

```
# pocet miest
n = 5

# Tabulka pre cesty dlzky 3
T3 = [
    [2, 5, 3, 3, 6],
    [5, 4, 3, 7, 7],
    [3, 3, 2, 5, 6],
    [3, 7, 5, 4, 7],
    [6, 7, 6, 7, 6]
]

# Tabulka pre cesty dlzky 9
T9 = [
    [1972, 2681, 1993, 2647, 3182],
    [2681, 3546, 2647, 3601, 4255],
    [1993, 2647, 1972, 2681, 3182],
    [2647, 3601, 2681, 3546, 4255],
    [3182, 4255, 3182, 4255, 5038]
]

# Tabulke pre cesty dlzky 12
# zatial same 0
T12 = [[0 for j in range(n)] for i in range(n)]

# pre kazdy zaciatok
for z in range(n):
```

```
    # pre kazdy koniec
    for k in range(n):
        # pre kazdu medzizastavku
        for m in range(n):
            T12[z][k] += T9[z][m] * T3[m][k]

print(T12)
```

Riešenie opäť zapíšeme do tabuľky:

z/do	A	B	C	D	E
A	50 361	67 366	50 272	67 510	80 178
B	67 366	90 522	67 510	90 289	107 527
C	50 272	67 510	50 361	67 366	80 178
D	67 510	90 289	67 366	90 522	107 527
E	80 178	107 527	80 178	107 527	127 982

Podúloha (c)

Tu iba kúsok zovšeobecníme postup s predchádzajúcej podúlohy. Najprv si spravíme tabuľku 5×5, kde bude 1 ak medzi danými miestami je cesta, inak 0. Takáto tabuľka je totižto tabuľka počtov ciest dĺžky 1 pre každý začiatok a koniec. Ďalej využijeme medzizastávku, tentokrát už po prvom kroku. Tak vieme veľmi podobným algoritmom získať tabuľku pre cesty dĺžky $k + 1$.

Podúloha (d)

Tentokrát ide zase o zovšeobecnenie, namiesto tabuliek veľkosti 5×5 budeme mať tabuľky $n \times n$. Inak je postup rovnaký ako v predošlej podúlohe.

Časová zložitosť

Môžeme sa ešte zamyslieť ako dlho bude niečo takéto trvať. Skúsime teda spočítať akú bude mať takýto algoritmus časovú zložitosť. Ak ste o časovej zložitosti ešte nepočuli, môžete si o nej viac prečítať v kuchárke[2].

Aj z programu si môžeme všimnúť, že využívame 3 vnorené cykly, a v každom prechádzame cez všetky miesta (pre každý začiatok a pre každý koniec prejdeme každú medzizastávku). Časová zložitosť teda je $O(n^3)$.

Matice

Takéto tabuľky s ktorými pracujeme môžeme pomenovať matice. Naše spájanie tabuliek je ekvivalnentné s násobením matíc. To sa okrem tohto nášho spôsobu používa na riešenie sústav rovníc, alebo pri zobrazovaní rôznych útvarov napr. v počítačových hrách. Viac si o tom môžete prečítať napríklad tu[3].

Sada úloh 3 – Spájanie tabuliek

Podúloha (a)

Opäť sa jedná o ďalšie zovšeobecnenie predošlej podúlohy. Môžeme si všimnúť, že opať využijeme vpodstate rovnaký algoritmus. Tento krát na ceste dĺžky $k + l$ spravíme medzizastávku po k krokoch.

Podúloha (b)

Najprv si spravíme tabuľku $n \times n$ pre počty ciest dĺžky 1. Teda v tabuľke bude 1 ak dané dve miesta sú spojené cestou, inak tam bude 0.

Pomalé spájanie

Prvé riešenie, ktoré by nám mohlo napadnúť je, že môžeme postupne spojiť 100 takýchto tabuliek. Spojili by sme teda najprv túto tabuľku samú zo sebou, potom výslednú znovu s tou pôvodnou, … až kým by sme nedostali tabuľku pre cesty dĺžky 100. Potrebovali by sme na to 99 spojení.

[2] https://www.ksp.sk/kucharka/zlozitost0/
[3] https://sk.wikipedia.org/wiki/Matica_(matematika)

Rýchlejšie spájanie

Veľmi ľahko si vieme všimnúť, že takéto spájanie nie je optimálne. Po prvom spojení získame tabuľku pre cesty dĺžky 2. Ak by sme ju spojili so sebou samou, dostali by sme tabuľku pre 4. K nej by sme opäť mohli pridať tabuľku 2 a dostali by sme tabuľku pre dĺžku 6. Po 49 by sme mali tabuľku pre dĺžku 100. Dokopy by sme použili 50 spojení, započítavajúc aj to úplne prvé. Je však jasné, že ani pri dĺžke 2 by sme sa nemuseli zastaviť.

Najrýchlejšie spájanie

Zamyslime sa nad otázkou, pre akú najväčšiu dĺžku vieme mať tabuľku po $1, 2, ..., n$ spojeniach. Jedným spojením vieme dostať najväčšie dĺžku, ak spojíme tabuľky s čo najväčšou dĺžkou. Preto sa nám vždy oplatí spojiť tabuľku pre najväčšiu dĺžku samú zo sebou. Tým sa nám po jednom spojení najväčšia dĺžka zdvojnásobí, teda po prvom spojení bude 2, po druhom 4, po treťom 8, a vo všeobecnosti po n spojeniach bude 2^n. Vieme teda, že po n spojeniach určite nemôže mať tabuľku pre dĺžku väčšiu ako 2^n.

Spájanie pre dĺžku 100

Vieme, že budeme potrebovať viac ako 6 spojení, lebo $2^6 = 64$. Keďže 100 nie je mocninou 2, budeme musieť spraviť nejaké "neoptimálne" spojenia.

Spôsob ako získať tabuľku pre dĺžku 100: Najprv spojíme tabuľku pre dĺžku 1 s tou istou tabuľkou, a tým dostaneme tabuľku pre dĺžky 2. Potom spojíme tabuľky s dĺžkami 2 a 1, a tak dostaneme tabuľku pre dĺžky 3. Potom spojíme tabuľky s dĺžkami 2 a 3, a dostaneme dĺžky 5. Spojíme tabuľku s dĺžkami 5 s tou istou tabuľkou a dostaneme tabuľku pre dĺžky 10. Tú potom spojíme s tabuľkou pre dĺžky 5 a dostane tabuľku pre dĺžky 15. Tú spojíme znovu s tabuľkou pre 10, a dostaneme tabuľku pre 25. Tú spojíme samú zo sebou a dostaneme tabuľku pre dĺžky 50. No a tú keď spojíme samú zo sebou dostaneme tabuľku pre dĺžky 100.

Použili sme dokopy iba 8 spojení tabuliek. Existuje viacero rôznych riešní, ktoré využívajú 8 spojení, ale neexistuje žiadne, ktoré ich využíva menej.

Podúloha (c)

Opäť začneme s tabuľkou pre dĺžky 1. Budeme pokračovať ako v poslednej podúlohe, a vždy spojíme tabuľku pre najväčšiu dĺžku samú so sebou. Dostaneme teda tabuľky pre dĺžky ktoré sú mocninami čísla 2. Prestaneme, ak by sme mali dostať tabuľku väčšiu ako je d. Taká tabuľka nám určite nepomôže.

Každé číslo sa dá zapísať ako súčet nejakých mocnín 2, s tým že sa nebudú opakovať. Tento zápis nám reprezentuje zápis daného čísla v dvojkovej sústave. Stačí nám teda pospájať zodpovedajúce tabuľky, a získame tabuľku pre cesty dĺžky presne d.

Pre hodnotu $d = 100$ by to boli nasledovné tabuľky: 64, 32, 4.

Logaritmus

Toto slovo nie je nejaké zaklínadlo, ale je to názov matematickej funkcie. Táto funkcia je inverznou (opačnou) k umocňovaniu. Každý logaritmus má nejaký základ, my budeme tak, ako sa často v informatike používa používať ten so základom 2. Logaritmus nejakého čísla nám hovorí akou mocninou základu (v našom prípade 2) je dané číslo. Teda $\log(2^x) = x$.

Koľko spojení použije takéto riešenie?

Najprv sme spravili najviac $\log d$ spojení aby sme si predrátali tabuľky pre dĺžky mocnín 2. Potom pri spájaní sme každú spojili najviac raz, teda sme spravili najviac $\log d$ spojení. Celkovo teda takýto program využíva $O(\log d)$ spojení.

3. Ambiciózne múzeá

vzorák napísal(a) Marianka
(max. 100 b za riešenie)

Sada úloh 1 – Stratégie kupovania budov

Podúloha a)

Podúlohu si pri počítaní, koľko peňazí Samko zaplatí, kým vyzbiera všetky kuriozity sveta, vieme rozdeliť na 3 časti:

- koľko peňazí zaplatí za všetky kuriozity sveta

- koľko peňazí zaplatí za múzeá s vitrínami
- koľko peňazí zaplatí za presun kuriozít zo starého do nového múzea

Vieme, že na svete je iba 100 kuriozít, takže ak bude chcieť Samko nakúpiť všetky kuriozity sveta, minie za ne 100 peňazí.

Na to, aby niekde kuriozity uložil, si bude kupovať múzeá s vitrínami. Zo zadania vieme, že si Samko postupne kupoval múzeá s $10, 20, \ldots, 100$ vitrínami – začal s múzeom s 10 vitrínami a postupne si kupuje múzeá, kde je o 10 vitrín viac od predchádzajúceho múzea. Nakoniec skončí s múzeom so 100 vitrínami, lebo väčšie múzeum už nepotrebuje. Za múzeá preto zaplatí $10 + 20 + \cdots + 100 = 550$ peňazí.

Pri tom, ako si kupuje nové múzeá, musí kuriozity presúvať. Vždy presunie všetky kuriozity z predchádzajúceho múzea do nového, avšak z posledného múzea už kuriozity nepresúva. Preto zaplatí $10 + 20 + \cdots + 90 = 450$ peňazí. Dokopy Samko zaplatí $100 + 550 + 450 = 1100$ peňazí.

Takáto stratégia sa hodí pre menšie počty kuriozít, pretože Samko nekupuje zbytočne veľké múzeá s veľa vitrínami, ktoré nakoniec nevyužije naplno. Pre vysoké počty sa naopak nehodí, lebo by príliš často musel kupovať nové múzeá a presúvať kuriozity.

Podúloha b)

V podúlohe b) vieme postupovať rovnako. Keďže je všetkých kuriozít sveta 10 000 000, tak zaplatí 10 000 000 peňazí. Pri zväčšovaní múzeí vždy o 1 000 000 vitrín zaplatí $1\,000\,000 + 2\,000\,000 + \cdots + 10\,000\,000 = 55\,000\,000$ peňazí. Keď bude kuriozity presúvať, tak zaplatí $1\,000\,000 + 2\,000\,000 + \cdots + 9\,000\,000 = 45\,000\,000$ peňazí. Dokopy teda zaplatí $10\,000\,000 + 55\,000\,000 + 45\,000\,000 = 110\,000\,000$ peňazí.

Takáto stratégia sa hodí pre väčšie počty kuriozít, pretože si Samko kúpi veľké múzeum s veľa vitrínami a nemusí sa tak často sťahovať a presúvať kuriozity. Pri malých počtoch kuriozít si Samko kúpi príliš veľké múzeum, ktoré sa zaplní iba čiastočne, takže naň minie zbytočne veľa peňazí.

Podúloha c)

Neexistuje iba jedna lepšia stratégia ako Samkova. Stratégií je veľa a sú rôzne efektívne. Do vzorového riešenia, som vybrala jednu z najefektívnejších, s ktorými sa vieme stretnúť. Jej princíp spočíva v tom, že na začiatku si kúpime múzeum s 1 vitrínou a každé ďalšie múzeum bude mať dvakrát toľko vitrín ako predchádzajúce. Takže kupované múzeá by mali $1, 2, 4, 8, 16, \ldots$ vitrín, inak povedané $2^0, 2^1, 2^2, 2^3, 2^4, \ldots$ vitrín. Táto stratégia nefunguje iba striktne s číslom 2. Môžeme začať aj napríklad s múzeom s 5 vitrínami a každé ďalšie bude mať päťkrát viac vitrín ako predchádzajúce. Hlavná pointa stratégie je, že vždy počet vitrín násobíme tým istým číslom.

Poďme sa pozrieť, koľko Samko zaplatí pri tejto stratégií, ak je všetkých kuriozít sveta 100. Zase si úlohu rozdelíme na tri časti. Za všetky kuriozity sveta zaplatí 100 peňazí. Múzeá teraz bude kupovať s $1, 2, 4, \ldots, 128$ vitrínami – 128 lebo je to prvá mocnina dvojky, ktorá je aspoň 100, čiže nám stačí na uskladnenie všetkých kuriozít. Takže za kupovanie múzeí tentokrát zaplatí $1 + 2 + 4 + \cdots + 128 = 255$ peňazí. Za presun zaplatí $1 + 2 + 4 + \cdots + 64$ peňazí, dokopy 127 peňazí. Za všetko dokopy zaplatí $100 + 255 + 127 = 482$.

Porovnajme si túto stratégiu s ostatnými. Pri stratégii z podúlohy a) si pamätáme, že zaplatil 1100 peňazí. Ak by sme použili stratégiu z podúlohy b), tak by Samko zaplatil 100 peňazí za kuriozity, 1 000 000 peňazí za múzeum a nič za presun, pretože všetky kuriozity sa mu zmestia hneď do prvého múzea – dokopy 1 000 100 peňazí. V porovnaní násobiacou stratégia minie menej peňazí ako zvyšné dve.

Ak je všetkých kuriozít sveta 10 000 000, tak za ne zaplatí 10 000 000 peňazí. Za kupovanie múzeí zaplatí $1 + 2 + 4 + \cdots + 2^{24}$, pretože 2^{24} je prvá mocnina dvojky, ktorá je aspoň taká veľká ako 10 miliónov. Toto je v súčte $2^{25} - 1 = 33\,554\,431$. Za presun zaplatí $1 + 2 + 4 + \cdots + 2^{23}$, dokopy $2^{24} - 1 = 16\,777\,215$. Nakoniec za všetko spolu zaplatí $10\,000\,000 + 33\,554\,431 + 16\,777\,215 = 60\,331\,646$.

Poďme si túto stratégiu znova porovnať so zvyšnými dvoma. Ak by sme použili stratégiu z podúlohy a), tak by za kuriozity zaplatil 10 000 000, za múzeá by zaplatil 5 000 005 000 000 peňazí a za presuny 4 999 995 000 000, čo je dokopy viac ako 10^{13} peňazí. Pri stratégií z podúlohy b) už vieme, že zaplatí 110 000 000 peňazí. V porovnaní so zvyšnými dvoma zase minie násobiacou stratégia menej peňazí.

Prečo je táto stratégia výhodná pre akýkoľvek počet kuriozít? Hlavným dôvodom je to, že pri násobení vitrín máme zo začiatku múzeá s malými počtami vitrín, pre menšie čísla a postupne sa dostávame rýchlejšie na väčšie a väčšie počty vitrín pre veľké čísla. Toto nám veľmi dobre balansuje to, ako často potrebujeme kupovať nové múzeá a teda aj presúvanie kuriozít.

Sada úloh 2 – Odkladanie kuriozít

Samkovi je najjednoduchšie dávať kuriozity do vitrín v poradí, čiže od 1 po n – počet vitrín v múzeu.

Podúloha a)

Na prvú podúlohu nám stačí, aby si zapamätal jednu premennú, označme si ju k ako koniec. Táto premenná hovorí o tom, do ktorej vitríny pôjde nasledujúca kuriozita, čiže kde je koniec poukladaných kuriozít. Na začiatku bude rovná 1 a zakaždým, keď Samko získa novú kuriozitu, tak sa premenná k zvýši o 1. Takže Samko vždy bude vedieť číslo vitríny, do ktorej pridá novú kuriozitu.

Podúloha b)

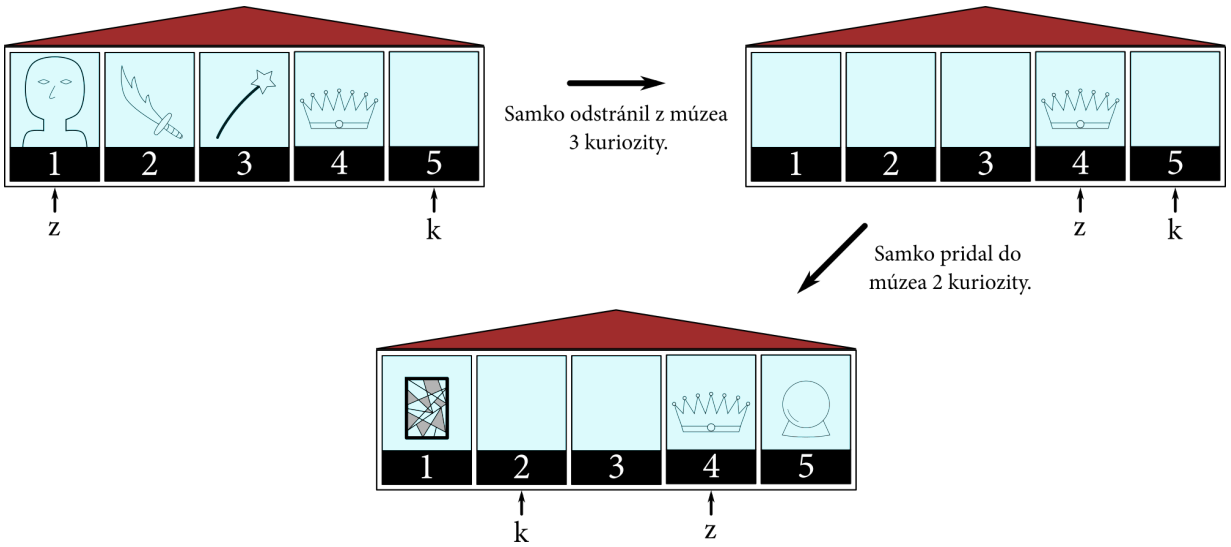
Pri druhej podúlohe si bude Samko musieť pamätať ďalšiu premennú, ktorú si označme z ako začiatok. Bude symbolizovať číslo vitríny, v ktorej je momentálna najstaršia kuriozita, čiže začiatok poukladaných kuriozít. Keďže si Samko ukladá kuriozity od prvej vitríny, tak na začiatku bude $z = 1$. Pridávanie kuriozít do múzea z nijak neovplyvní, keď však bude Samko chcieť odstrániť najstaršiu kuriozitu, tak odstráni tú, ktorá je vo vitríne s číslom z. Po odstránení sa z zvýši o 1, pretože teraz je najstaršia kuriozita v nasledujúcej vitríne.

Podúloha c)

Samko teraz neovplyvňuje, kedy získa novú a kedy odstráni najstaršiu kuriozitu. Ako teda zistí, že má plné múzeum a musí si kúpiť nové? Najpriamočiarejšie riešenie je to, že premenná k bude popisovať číslo vitríny, ktorá v múzeu neexistuje – predchádzajúca kuriozita bola pridaná do poslednej vitríny a k nesie číslo nasledujúcej, ktoré ale presahuje počet vitrín. Problém je však ten, že v takomto prípade by mohol mať na začiatku múzea prázdne vitríny, a tak sa Samkovi neoplatí kúpiť si nové múzeum – to sa mu oplatí iba vtedy, ak je múzeum naozaj úplne plné. Teraz má dve možnosti:

- Posunúť všetky kuriozity tak, aby boli znova od prvej vitríny. Tento spôsob je ale veľmi neefektívny, lebo musí veľa presúvať.
- Nastaviť si k znova od 1 a pridávať nové kuriozity do vitrín na začiatku. Týmto nám vznikne "hadík", ktorý si naháňa svoj chvost. Tento spôsob je výhodnejší, lebo nemusíme presúvať žiadne kuriozity.

Keďže chce byť čo najefektívnejší, tak použije druhú možnosť. Tu je ukážka, ako táto možnosť funguje v praxi:



Pri nej ale musí nájsť iný spôsob, akým zistí, či má plné múzeum. Aj tento spôsob je pomerne jednoduchý. Múzeum bude plné, keď najmladšia kuriozita "dobehne" najstaršiu, teda keď sa $k = z$. V takom prípade by Samko chcel nasledujúcu kuriozitu pridať do vitríny s číslom k, ale v nej sa už nachádza najstaršia kuriozita.

V prípade, keď si Samko musí kúpiť nové múzeum, postupuje rovnako ako v prvej sade – kúpi si múzeum s dvojnásobným počtom vitrín.

Čo sa teraz stane s kuriozitami, keď treba kúpiť nové múzeum? Tým, že aj tak sa všetky kuriozity idú presúvať, tak si ich do nového múzea môže usporiadať tak, aby mal najstaršiu kuriozitu v prvej vitríne až postupne po najmladšiu kuriozitu. Čiže z sa prepíše na 1 a k sa prepíše na počet kuriozít $+1$, pretože k je číslo vitríny, do ktorej budeme pridávať novú kuriozitu.

vzorák napísal(a) Adam
(max. 0 b za riešenie)

4. Substringy

Teória

Aby sme vedeli spočítať skóre súrodencov, potrebujeme spôsob, ako určiť, či je písmeno samohláska, alebo spoluhláska. Bohužiaľ túto vlastnosť písmen počítač nedokáže rozpoznať sám od seba a musíme mu najskôr povedať ako taká samohláska vyzerá. Aby sme si to zjednodušili zapamätáme si len ako vyzerajú samohlásky a necháme program rozhodnúť či kontrolované písmeno je alebo nie je samohláska. Kontrola prebehne tak, že program prehľadá pole samohlások a pri zhode s kontrolovaným písmenom okamžite rozhodne, že písmeno je samohláska, inak po dokončení prehľadávania rozhodne, že písmeno je spoluhláska (pretože to nebola žiadna zo samohlások). Takýmto spôsobom program nemusí vôbec vedieť o tom, ako vyzerajú spoluhlásky a stále dostaneme správne riešenie. Ak by sme pri riešení implementovali aj separátne pole pre spoluhlásky, tak by program v pomalších jazykoch nezbehol na plný počet bodov.

Riešenie hrubou silou

Najpriamočiarejšie riešenie je skontrolovať, akým písmenom sa začína každý možný substring v slove pomocou spôsobu, ktorý sme si vysvetlili v teórii. Všetkých substringov je rádovo $n^2/2$, lebo pre písmeno na indexe i existuje $n-i$ substringov, ktoré ním začínajú a i substringov, ktoré sa ním končia. Takéto riešenie má časovú zložitosť $O(n^2)$ a v časovom limite vyrieši len prvé 2 sady.

Za takéto riešenie bolo možné získať 40 bodov.

Listing programu (Python)

```
n = int(input())
a = input()
s = [0, 0]

# zname samohlasky
sam = ["a", "e", "i", "o", "u", "y", "A", "E", "I", "O", "U", "Y"]

# spocitame zaciatky spol. a sam. vsetkych substringov v slove
for i in range(n):
    for j in range(n):
        if a[i:j+1].startswith(a[i]):
            if sam.__contains__(a[i]):
                s[0] += 1
            else:
                s[1] += 1
print(*s)
```

Vzorové riešenie

Ak chceme zlepšiť časovú zložitosť, musíme znížiť počet krokov, ktoré program vykoná. Môžeme si všimnúť, že počet substringov začínajúcich písmenom na pozícii i je $n-i$. Od začiatku substringu n_i, ktorý poznáme, sa substring vie predlžovať len do jedného smeru a to len presne $n-i$ krát, lebo tam slovo končí a substring sa už nemá ako predĺžiť. Počet predĺžení od písmena vieme pre každé písmeno v slove zistiť v konštantnom čase, takže celú úlohu vieme vyriešiť v lineárnom čase $O(n)$. Pamäťová zložitosť je $O(1)$, lebo si v pamäti držíme len konštantný počet samohlások a výsledky.

Keby sme si počet samohlások označili s, časová zložitosť by bola $O(ns)$. Keby sme si tieto samohlásky pamätali vo vhodnej štruktúre, vedeli by sme o každom písmene zistiť, či je to samohláska v konštantom čase. Takouto štruktúrou je napríklad `unordered_set` v C++, alebo `set` v pythone. Toto vylepšenie však na plný počet bodov nebolo potrebné.

Listing programu (C++)

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int n;
    string a;
    cin >> n;
    cin >> a;
    long long out[] = {0,0};

    // zname samohlasky
    string sam = "aeiouyAEIOUY";

    // pre kazde pismeno pripocitam pocet substringov
    for(int i=0; i<n; i++) {

        // je pismeno samohlaska?
        bool result = false;
        for(int j=0; j<sam.size(); j++) {
            if (a[i] == sam[j]) result = true;
        }

        if (result) {
            out[0] += n - i;
        } else {
            out[1] += n - i;
        }
    }

    cout << out[0] << " " << out[1] << endl;
    return 0;
}
```

Listing programu (Python)

```
n = int(input())
a = input()
s = [0, 0]

# zname samohlasky
sam = ["a", "e", "i", "o", "u", "y", "A", "E", "I", "O", "U", "Y"]

# pre kazde pismeno pripocitam pocet substringov
for i in range(n):
    if sam.__contains__(a[i]):
        s[0] += n - i
    else:
        s[1] += n - i
print(*s)
```

vzorák napísal(a) Žaba
(max. 100 b za riešenie)

5. Korytnačie preteky

Naprogramovať správne riešenie k tejto úlohe je pomerne jednoduché – stačí postupne simulovať jednotlivé kroky korytnačiek. Najjednoduchšie je vytvoriť si pre každé políčko hracieho plánu samostatné pole, v ktorom sú poukladané korytnačky v poradí, v akom stoja na danom políčku. Hodnoty na vstupe nám dokonca určujú, na ktorom políčku stojí korytnačka, ktorú presúvame, stačí preto z tohto poľa vybrať všetky korytnačky, ktoré sú nad ňou a v správnom poradí ich uložiť do cieľového políčka (teda to s číslom $p_i + x_i$).

Takéto riešenie avšak **nie je efektívne**. Môžeme však na ňom založiť rýchlejšie riešenie. Potrebujeme iba porozmýšľať, čo spôsobuje jeho neefektívnosť. Problémom samozrejme je, že pri simulovaní presúvame každú

korytnačku samostatne a takýchto korytnačiek môže byť príliš veľa. Potrebovali by sme preto vymyslieť, ako nasimulovať to, čo vieme jednoducho spraviť v skutočnom živote – presunúť všetky korytnačky naraz.

Prvé riešenie si pre každú korytnačku v podstate pamätá, na ktorom políčku sa nachádza a táto informácia sa pri jednom presune naozaj zmení veľa korytnačkám. Aká informácia sa však pri jednom presune nezmení skoro žiadnej korytnačke? Predsa to, **ktorá korytnačka sa nachádza priamo podomnou**. Keď presúvame nejaký stĺpec korytnačiek, všetky okrem tej najspodnejšej ostanú sedieť na tej istej korytnačke a ich hodnota sa vôbec nemusí meniť. Na spracovanie jedného presunu preto stačí opraviť jednu hodnotu.

Navyše, túto informáciu na začiatku poznáme, sú ňou zadané hodnoty naskladaných korytnačiek a vieme pomocou nej vypočítať aj výsledok. Na konci totiž jednoducho prejdeme cez korytnačky a budeme sa posúvať vždy na nižšie ležiacu korytnačku, čím dostaneme jeden výsledný stĺpec.

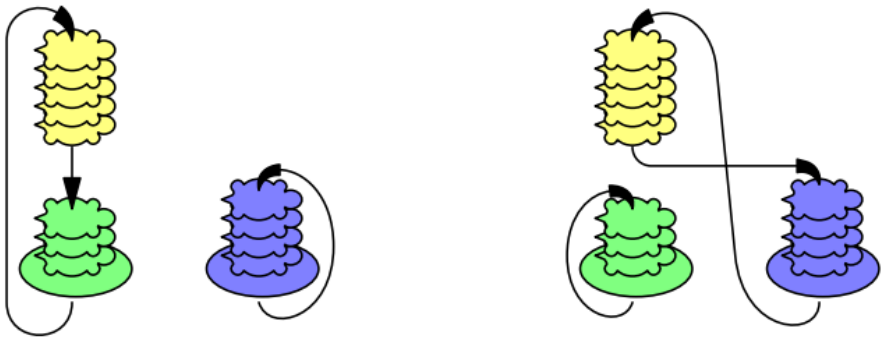
Všímavý/á čitateľ/ka si však určite všimol/la jeden problém. Ako upravíme hodnotu presúvanej korytnačky? Ako vieme, na chrbát ktorej inej korytnačky sa postaví? Poprípade, čo ak na novom políčku ešte nie je žiadna korytnačka?

Pri každom presune sa korytnačka postaví navrch stĺpika na danom políčku. Pre každé políčko by sme preto chceli vedieť **číslo najvyššej korytnačky, ktorá na ňom stojí**. A v prípade, že tam žiadna korytnačka nestojí, môžeme si zapamätať napríklad hodnotu 0. Vieme však aj tieto hodnoty pri presune upravovať rovnako rýchlo?

Zoberme si dva stĺpiky korytnačiek na políčkach A a B a korytnačku x, ktorú presúvame z A na vrch B. Zmenia sa nasledovné tri hodnoty:

- číslo korytnačky, na ktorej sedí korytnačka x – táto hodnota bude najvyššia korytnačka na políčku B
- hodnota najvyššej korytnačky na políčku B – toto bude najvyššia korytnačka políčka A, ktorá sa spolu s x presunula na vrch B
- hodnota najvyššej korytnačky na políčku A – z políčka A sme odobrali všetky korytnačky, ktoré boli nad x, najvyššie teda ostane korytnačka, na ktorej predtým sedela korytnačka x

Túto zmenu hodnôt pekne ilustruje nasledovný obrázok, v ktorom sa žlté korytnačky, ktoré sedeli na zelených posunuli na vrch modrých korytnačiek.



Pre každú korytnačku a políčko si teda stačí pamätať jedno číslo a pri každom presune zmeniť tri tieto čísla, čo vieme spraviť v konštantnom čase. Na konci sa jednoducho pre každé políčko pozrieme na najvyššiu korytnačku a z nej sa budeme posúvať dodola až kým nenarazíme na korytnačku, ktorá si pamätá, že nesedí na žiadnej korytnačke. Celková časová zložitosť takéhoto riešenia je $O(n+p+q)$ a pamäťová zložitosť je $O(n+p)$.

Listing programu (C++)

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int n, p, q;
```

```
    cin >> n >> p >> q;

    vector<int> podo_mnou(n + 1, 0); // kladna je koryt, 0 policko
    vector<int> vrchol(p + 1, 0);    // kladna je koryt, 0 nic

    int before = 0;
    for (size_t i = 0; i < n; ++i)
    {
        int kor;
        cin >> kor;
        podo_mnou[kor] = before;
        before = kor;
    }
    vrchol[1] = before;

    int k_i, p_i, x_i;
    while (cin >> k_i >> p_i >> x_i)
    {
        int old_vrchol = vrchol[p_i];
        int new_vrchol = vrchol[p_i + x_i];
        int pod = podo_mnou[k_i];

        vrchol[p_i] = pod;
        vrchol[p_i + x_i] = old_vrchol;

        podo_mnou[k_i] = new_vrchol;
    }

    for (size_t i = 1; i < vrchol.size(); ++i)
    {
        vector<int> out;

        int v = vrchol[i];
        while (v != 0)
        {
            out.push_back(v);
            v = podo_mnou[v];
        }

        reverse(out.begin(), out.end());

        cout << out.size();
        for (int kor : out)
            cout << " " << kor;
        cout << "\n";
    }
}
```