B-I-1 Kopa tričiek

V našom prvom riešení využijeme, že počty nosení tričiek sú rozumne malé (neprevyšujú 10^6). Namiesto jednej obrovskej kopy si spravíme veľa menších kôp: pre každé x si spravíme kopu, na ktorej budú všetky tričká, ktoré mal Žaba zatiaľ na sebe presne x-krát. Na začiatku tam tričká rozmiestnime tak, aby ich poradie zhora dole zodpovedalo poradiu zhora dole v pôvodnej kope.
Navyše si ešte spravíme jednu pomocnú premennú, v ktorej si budeme pamätať, ktoré tričko je práve na vrchu celej kopy.
Ako bude vyzerať deň podľa Lucky? Ráno Žaba nájde najmenšie x, ktorému zodpovedá neprázdna kopa tričiek. Z tejto kopy si zoberie vrchné tričko – teda to, ktoré bolo najbližšie k vrchu pôvodnej kopy. No a keď ho večer operie a usuší, položí ho na vrch kopy s číslom $x+1$. (Všimnite si, že aj po tomto úkone platí, že tričká na kôpke $x+1$ sú uložené v tom poradí, v ktorom by v skutočnosti boli v pôvodnej kope – práve nosené tričko totiž šlo na vrch úplne celej pôvodnej kopy.) No a na záver si už len práve donosené tričko uložíme do premennej pre vrch kopy.
A ako bude vyzerať deň podľa Žabu? Jeho spôsob vyberania trička sa simuluje ľahko: stačí sa pozrieť do pomocnej premennej na tričko na vrchu kopy. Potom, ako ho donosí, ho však nesmieme zabudnúť presunúť „o kôpku vyššie“ – teda ak doteraz bolo na kôpke pre y nosení, odteraz bude na kôpke pre $y+1$.
Na záver zopár implementačných detailov:

- Keďže žiadnemu tričku počet nosení nemôže ubudnúť, nemusí Žaba hľadať najmenšiu neprázdnu kopu vždy odznova. Namiesto toho môžeme šikovne využiť, že ak naposledy bral tričko z kôpky x, najbližšie ho bude brať z kôpky s číslom aspoň x.
- Na zapamätanie kôpky nám stačí ľubovoľná dátová štruktúra, z ktorej vieme na jednom konci efektívne vyberať prvky a aj na ten istý koniec nové vkladať. Základnou takouto štruktúrou je zásobník. Môžeme však použiť aj všeobecnejšie dátové štruktúry (ako napr. vector v C++, ArrayList v Jave, prípadne list v Pythone).
- Pri šikovnej implementácii dostávame časovú zložitosť $O(n+q+m)$, kde n je počet tričiek, q je počet dní, ktoré treba odsimulovať, a $m = \max t_i$ je najväčší počet oblečení trička doteraz.

Listing programu (Python)

```
N, Q = [ int(_) for _ in input().split() ]
T = [ int(_) for _ in input().split() ]
M = max(T)
T = [ None ] + T
na_vrchu = 1

# spravíme si prázdne kôpky
kopky = [ [] for _ in range(M+Q+1) ]

# rozmiestnime tričká na kôpky podľa počtu nosení
for n in reversed(range(1,N+1)):
    kopky[ T[n] ].append(n)

odkial_ber = 0

# čítame otázky a simulujeme
for q in range(Q):
    if input() == 'Z':
        print(na_vrchu)
        kde = T[na_vrchu]
        kopky[kde].pop()
        kopky[kde+1].append(na_vrchu)
        T[na_vrchu] += 1
    else:
        while kopky[odkial_ber] == []:
            odkial_ber += 1
        nosil = kopky[odkial_ber].pop()
        print(nosil)
        na_vrchu = nosil
        kopky[odkial_ber+1].append(nosil)
        T[nosil] += 1
```

Iné efektívne riešenie môžeme založiť na použití pokročilejšej dátovej štruktúry: usporiadanej množiny. Presnejšie, dvoch takýchto množín. V jednej si budeme udržiavať všetky tričká usporiadané podľa toho, kedy boli

naposledy nosené (čiže podľa poradia vo veľkej kope zhora dole) a v druhej budú usporiadané primárne podľa počtu nosení a až sekundárne podľa poradia vo veľkej kope.
Podľa toho, kto si v ten deň vyberá, vyberieme záznam o tričku zo začiatku jednej alebo druhej usporiadanej množiny. Následne tento záznam z oboch zmažeme a namiesto neho tam vložíme nový záznam, v ktorom už má toto tričko o jedno nosenie viac a navyše dostalo nový vyšší timestamp (čiže je aj naposledy nosené zo všetkých).
Každá operácia s usporiadanou množinou, v ktorej je n tričiek, nám trvá $O(\log n)$, preto má toto riešenie časovú zložitosť $O((n+q)\log n)$.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;

struct tricko { int cislo, timestamp, pocet_noseni; };

struct porovnaj_timestamp {
    bool operator() (const tricko &A, const tricko &B) const { return A.timestamp > B.timestamp; }
};

struct porovnaj_pocet {
    bool operator() (const tricko &A, const tricko &B) const {
        if (A.pocet_noseni != B.pocet_noseni) return A.pocet_noseni < B.pocet_noseni;
        return A.timestamp > B.timestamp;
    }
};

int main() {
    int N, Q;
    cin >> N >> Q;

    // spravime si dve mnoziny triciek: jednu usporiadanu primarne podla poctu noseni,
    // druhu podla toho kedy bolo naposledy nosene
    set<tricko,porovnaj_pocet> najmenej_nosene;
    set<tricko,porovnaj_timestamp> naposledy_nosene;

    // ulozime tricka do oboch mnozin
    for (int n=0; n<N; ++n) {
        int t;
        cin >> t;
        najmenej_nosene.insert( {n+1, N-1-n, t} );
        naposledy_nosene.insert( {n+1, N-1-n, t} );
    }

    // spracuvame jednotlive dni
    for (int q=0; q<Q; ++q) {
        string kto;
        cin >> kto;
        tricko dnes;
        if (kto == "Z") dnes = *naposledy_nosene.begin(); else dnes = *najmenej_nosene.begin();
        cout << dnes.cislo << endl;
        najmenej_nosene.erase(dnes);
        naposledy_nosene.erase(dnes);
        dnes.timestamp = N+q;
        ++dnes.pocet_noseni;
        najmenej_nosene.insert(dnes);
        naposledy_nosene.insert(dnes);
    }
}
```

B-I-2 O Jankinom bratovi

Predstavme si, že spomedzi slov, ktoré sme už spracovali, vieme vybrať dve rôzne postupnosti, ktoré končia tým istým písmenom. Napríklad nech jedna z nich je (cat,tea) a druhá (dog,golf,flea). Potrebujeme si pamätať obe? Zjavne nie. Tú prvú, kratšiu, nemá zmysel si pamätať. Hocijaké slovo, ktoré vieme doplniť za ňu, vieme doplniť aj za tú dlhšiu a dostaneme tak lepšie riešenie.
Z toho dostávame nasledujúce pozorovanie: keď postupne čítame slová zo vstupu, jediná informácia, ktorú si potrebujeme pamätať, je pre každé písmeno abecedy dĺžka najdlhšej postupnosti, ktorá ním končí.
Ešte si potrebujeme popísať, ako si takto pamätané informácie budeme postupne upravovať podľa toho, aké slová prečítame zo vstupu.
Na začiatku vstupu si pre každé písmenko x povieme, že najlepšie riešenie končiace týmto písmenkom má dĺžku

$p_x = 0$, čiže ide o prázdnu postupnosť.
Predstavme si teraz, že už sme prečítali nejakú časť vstupu a v premenných $p_\star$ máme príslušné dĺžky postupností slov. Nech teraz napríklad ako ďalšie prečítame zo vstupu slovo „apple“. Toto slovo začína na „a“, takže ho môžeme pridať za najdlhšiu postupnosť skôr spracovaných slov ktorá končí práve písmenom „a“. Takto dostaneme o jedno dlhšiu postupnosť slov, ktorá už ale končí na písmeno „e“. No a už sa stačí len pozrieť, či táto nová postupnosť končiaca na „e“ nie je dlhšia ako najdlhšia, ktorú sme vedeli spraviť doteraz. Inými slovami, nová hodnota p_e bude maximom zo starej hodnoty p_e a z hodnoty $p_a + 1$. Ostatné hodnoty $p_\star$ sa zjavne nezmenia.
No a to už je všetko. Každé slovo zo vstupu spracujeme v konštantnom čase, a teda časová zložitosť je lineárna od počtu slov.

Listing programu (Python)

```
def pismenko_na_cislo(z):
    return ord(z) - ord('a')

def prve_pismenko(slovo):
    return pismenko_na_cislo( slovo[0] )

def posledne_pismenko(slovo):
    return pismenko_na_cislo( slovo[-1] )

N = int( input() )
P = [0]*26

for i in range(N):
    slovo = input()
    prve, posledne = prve_pismenko(slovo), posledne_pismenko(slovo)
    P[posledne] = max( P[posledne], P[prve]+1 )

print(max(P))
```

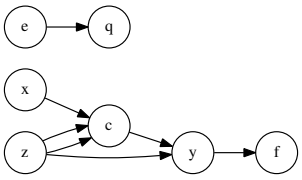
B-I-3 Cudzokrajná abeceda

Každá dvojica po sebe idúcich slov nám dá buď práve jednu informáciu tvaru „toto písmeno musí byť v abecede skôr ako tamto“, alebo sa z nej nedozvieme nič.
Väčšinou nastane ten prvý prípad. Ktoré dve písmená to sú? Tie, na ktorých sa dotyčná dvojica po sebe idúcich slov prvýkrát líši. Napríklad `srnka` bude v zozname slov skôr ako `srdce` vtedy a len vtedy, keď je `n` v cudzokrajnej abecede skôr ako `d`.
Kedy nastáva ten druhý prípad? Vtedy, keď je prvé slovo *prefixom* druhého – teda napríklad keď prvé slovo je `pes` a druhé `pestry`. Pre úplne ľubovoľnú cudzokrajnú abecedu bude `pes` vždy skôr ako `pestry`.

Predstavme si teraz, že sme sa postupne pozreli na všetky dvojice *po sebe idúcich* slov a vypísali sme si všetky podmienky pre cudziu abecedu, ktoré sme našli. Je zjavné, že pre hocijakú abecedu, ktorá spĺňa takto získané podmienky, už bude zoznam slov v správnom poradí. Dostávame teda novú úlohu: pre danú množinu podmienok nájsť ľubovoľnú abecedu, ktorá ich všetky spĺňa.

Topologické usporiadanie

Informácie, ktoré sme získali vyššie popísaným postupom, si vieme dobre reprezentovať ako orientovaný graf. Vrcholy grafu budú jednotlivé písmenká abecedy. Informáciu „x musí byť v abecede skôr ako y“ si zaznačíme ako orientovanú hranu z vrcholu x do vrcholu y. Na obrázku je príklad grafu, v ktorom je okrem iného zaznačené, že `e` má byť skôr ako `q`.



Našou úlohou je teraz všetky vrcholy grafu (teda celú abecedu) zoradiť do takého poradia, aby pre každú hranu $u \to v$ platilo, že u je uvedené skôr ako v. Takéto poradie nazývame *topologickým usporiadaním* vrcholov grafu.

Tranzitívny uzáver

Jeden ľahký spôsob, ako takéto poradie zostrojiť, je výpočet takzvaného *tranzitívneho uzáveru*. Začneme tým, že si graf budeme v pamäti reprezentovať ako maticu susednosti: pre každé dva vrcholy u a v si budeme v dvojrozmernom poli pamätať, či z u do v vedie hrana. (Všimnite si, že takto „zadarmo“ odstránime násobné hrany – ak aj bolo na vstupe hrán $u \to v$ sedem, stále si ich zapamätáme rovnako ako keby bola len jedna.)
Tranzitívny uzáver grafu vznikne tak, že doň doplníme všetky hrany, ktoré si vieme odvodiť z tých existujúcich. Teda pre ľubovoľné tri vrcholy x, y a z: ak máme v grafe hrany $x \to y$ aj $y \to z$, pridáme doň aj hranu $x \to z$ (keďže si vieme odvodiť, že x musí byť v abecede skôr ako z).
Systematický spôsob, ako všetky takéto hrany doplniť, ponúka Floydov algoritmus. Ten je veľmi jednoduchý:

```
pre každé k:
    pre každé i:
        pre každé j:
            ak máš hranu z i do k aj hranu z k do j:
                doplň hranu z i do j
```

(Dôkaz správnosti tohto algoritmu je pomerne komplikovaný a nebudeme ho tu uvádzať. Jeho časová zložitosť je $O(p^3)$, kde p je počet písmen abecedy, čiže počet vrcholov grafu. Zovšeobecnením tohto algoritmu je algoritmus Floyda a Warshalla, pomocou ktorého vieme počítať dĺžky najkratších ciest v ohodnotených grafoch.)
Ako nám tranzitívny uzáver pomôže? Pre každé písmeno si budeme vedieť presne spočítať, koľko iných má byť od neho menších. No a teraz už jednu možnú abecedu zostrojíme ľahko: stačí všetky písmená usporiadať práve podľa tohto počtu. Totiž ak máme v pôvonom grafe hranu z u do v, tak v tranzitívnom uzáveri do v nutne vedie aspoň o jednu hranu viac ako do u, a teda u vypíšeme skôr ako vu vypíšeme skôr ako vu vypíšeme skôr ako vu vypíšeme skôr ako v.

Prehľadávanie do hĺbky

Ukážeme si ešte jeden spôsob, ako nájsť topologické usporiadanie grafu. Tento nový spôsob bude efektívnejší – presnejšie, jeho časová zložitosť bude priamo úmerná veľkosti grafu na vstupe. Topologické usporiadanie pri ňom vypíšeme „od konca“.
Na každú hranu $u \to v$ sa teraz môžeme dívať ako na inštrukciu „skôr, než vypíšeš u, musíš vypísať v“. No a práve pri prehľadávaní do hĺbky vieme ľahko zabezpečiť splnenie týchto inštrukcií: keď prvýkrát navštívime vrchol u, tak najskôr rekurzívne prehľadáme (a následne vypíšeme) všetky vrcholy, do ktorých z neho vedú hrany, a až potom vypíšeme samotný u.
Pseudokód algoritmu:

```
funkcia prehľadaj(vrchol u):
    ak u ešte nebol navštívený:
        označ u ako navštívený
        pre každý vrchol v taký, že z u vedie hrana do v:
            prehľadaj(v)
        vypíš(u)

hlavný program:
    pre každý vrchol x:
        prehľadaj(x)
```

Ak si graf budeme pamätať ako zoznam okolí vrcholov (presnejšie, pre každý vrchol zoznam vrcholov, do ktorých z neho vedú hrany), je zjavné, že počas behu tohto algoritmu každý vrchol práve raz spracujeme a počas toho sa práve raz pozrieme na každú hranu grafu. V našom prípade máme p vrcholov a n hrán, dostávame teda časovú zložitosť $O(p + n)$.

B-I-4 Šimon a UFO

Postupne si preriešime všetky úlohy v poradí, v ktorom boli zadané.

Podúloha A (2 body)

Ak Šimon vie, že $n = 10$, je záchrana macka jednoduchá. Jeden možný postup: Šimon najskôr $10\times$ stlačí tlačidlo ℵ. Ak po desiatom stlačení pri ňom pristane UFO s mackom, zvíťazil. V opačnom prípade je ℵ zjavne tlačidlo, ktoré posiela macka o 10 km ďalej. To ale znamená, že macko je teraz od Šimona presne 110 km, a teda stačí $110\times$ stlačiť tlačidlo ℘.

Podúloha B (7 bodov)

Ako ale má Šimon postupovať, ak nevie hodnotu n? Dá sa to vôbec?
Základná schéma nášho riešenia bude nasledovná:

```
1. postláčaj tlačidlá tak, aby si macka určite zachránil, ak n = 1
    1a. zachráň macka ak ho prvé tlačidlo volá bližšie
    1b. zachráň macka ak ho druhé tlačidlo volá bližšie
2. postláčaj tlačidlá tak, aby si macka určite zachránil, ak n = 2
    2a. zachráň macka ak ho prvé tlačidlo volá bližšie
    2b. zachráň macka ak ho druhé tlačidlo volá bližšie
3. postláčaj tlačidlá tak, aby si macka určite zachránil, ak n = 3
    ...
```

Rozmyslime si najskôr exaktne ako bude vyzerať prvých pár krokov tohto postupu, potom si sformulujeme, ako ho zovšeobecniť.

1a Ak $n = 1$ a prvé tlačidlo volá macka bližšie: Stačí ho raz stlačiť.

1b Ak $n = 1$ a druhé tlačidlo volá macka bližšie: Už sme raz stlačili prvé tlačidlo, macko by teda teraz bol vo vzdialenosti 2 km. Treba teda dvakrát stlačiť druhé tlačidlo.

2a Nech $n = 2$ a prvé tlačidlo volá macka bližšie. Doteraz sme stlačili ℵ℘℘. Mackova vzdialenosť sa teda menila nasledovne: $2 \to 1 \to 3 \to 5$. Teraz teda treba $5\times$ stlačiť prvé tlačidlo.

2b Nech $n = 2$ a druhé tlačidlo volá macka bližšie. V tomto scenári by momentálne bol macko 12 km od zeme, treba teda $12\times$ stlačiť druhé tlačidlo.

Šimonov postup teda začína nasledovne: ℵ℘℘ℵℵℵℵ℘℘℘℘℘℘℘℘℘℘℘℘...
Po stlačení týchto tlačidiel má už Šimon istotu, že macka zachráni, ak $n = 1$ alebo $n = 2$. A už by malo byť zjavné, ako bude pokračovať ďalej. Postupne pre každé väčšie n zopakuje úvahu, ktorú sme si ukázali na príklade: spočíta si, kde by pre dotyčné n práve bol macko, a následne postláča príslušný počet tlačidiel, ktorý by ho mal dostať späť na Zem.
Celý Šimonov postup teda môžeme naprogramovať nasledovne:

Listing programu (Python)

```
stlacil_prve = 0
stlacil_druhe = 0
n = 1
while True:
    # ak prvé tlačidlo volá macka bližšie:
    kde_je_macko = n + n*stlacil_druhe - stlacil_prve
    print( kde_je_macko, 'krát stlač prvé tlačidlo' )
    stlacil_prve += kde_je_macko

    # ak druhé tlačidlo volá macka bližšie:
    kde_je_macko = n + n*stlacil_prve - stlacil_druhe
    print( kde_je_macko, 'krát stlač druhé tlačidlo' )
    stlacil_druhe += kde_je_macko

    n += 1
```

Podúloha C (1 bod)

Ako sme už uviedli v zadaní, táto podúloha bola ťažká a bola myslená ako výzva pre najlepších. Jej riešenie bude síce pomerne stručné, ale nevyhneme sa pri ňom poschodovým mocninám.
Všimnite si ešte raz riešenie podúlohy A. Už vieme, že toto riešenie funguje pre $n = 10$, tak sme ho stvorili. Lenže keď sa nad tým trochu zamyslíme, ľahko si všimneme, že funguje aj pre všetky menšie hodnoty n. Totiž ak je $n \leq 10$ a prvé tlačidlo volá UFO bližšie, tak na nanajvýš 10 stlačení tlačidla príde UFO k nám. A ak je $n \leq 10$ a správne tlačidlo je druhé, desať stlačení prvého tlačídla poslalo macka do vzdialenosti $11n \leq 110$, a teda následných 110 stlačení druhého tlačidla privedie macka domov.
Toto isté je pravda aj pre ľubovoľnú inú hodnotu n a ľubovoľný iný postup: ak máme postup, ktorý funguje pre konkrétne n, musí zjavne fungovať aj pre všetky menšie hodnoty.
Nemusíme teda jednotlivé hodnoty n spracúvať každú zvlášť. Namiesto toho sa na celú úlohu môžeme pozrieť akoby z opačného konca: zvolíme si konkrétny spôsob stláčania tlačidiel tak, aby na každé n prišiel rad dostatočne skoro.
Takýchto postupov je veľa, my si ukážeme jeden z nich, ktorý nám prišiel ľahko popísateľný.

```
nastav x = 2
aktívne tlačidlo je prvé
do nekonečna opakuj:
    x-krát stlač aktívne tlačidlo
    umocni x na druhú
    zmeň aktívne tlačidlo na opačné
```

Začiatok teda vyzerá nasledovne: $2\times$ prvé, $4\times$ druhé, $16\times$ prvé, $256\times$ druhé, $65\,536\times$ prvé, $4\,294\,967\,296\times$ druhé, ... Ak si jednotlivé iterácie cyklu očíslujeme od nuly, v i-tej iterácii aktívne tlačidlo stlačíme 2^{2^i}-krát.

Teraz si pre ľubovoľné i položme nasledujúcu otázku: ak je aktívne tlačidlo to správne (teda to, ktoré volá macka bližšie), pre aké hodnoty n zachránime macka najneskôr v i-tej iterácii cyklu?
V tejto iterácii postupne zavoláme macka o $b = 2^{2^i}$ km bližšie k nám. Pre $i \geq 2$ sme ho už aj skôr volali bližšie k nám, týchto stlačení však bolo v porovnaní s 2^{2^i} tak málo, že ich spokojne zanedbáme.
Koľkokrát sme predtým poslali macka o n km ďalej od nás? Dokopy je to $d = \left(2^{2^{i-1}} + 2^{2^{i-3}} + \cdots\right) + 1$, pričom tá $+1$ na konci zodpovedá tomu, že macko začina n km od Zeme. Na záver teraz ešte spravíme posledný, tentokrát veľmi voľný odhad: Zjavne pre ľubovoľné $i \geq 3$ platí, že ak zoberieme $n \leq 2^{2^{i-2}}$, tak $b > nd$. Inými slovami, pre ľubovoľné $n \leq 2^{2^{i-2}}$ stlačíme v i-tej fáze aktívne tlačidlo dosť veľakrát na to, aby sme macka zachránili, ak je toto aktívne tlačidlo to správne.

No a keďže aktívne tlačidlo po každej iterácii vymeníme, platí teda, že pre ľubovoľné $i \geq 1$ a ľubovoľné $n \leq 2^{2^i}$ macka zachránime prinajhoršom počas iterácie $i + 3$. Pri tom ale spravíme dokopy určite menej ako $2 \cdot 2^{2^{i+3}}$ stlačení tlačidla. No a to už je všetko.
Najhorší možný prípad pre každé i je totiž $n = 2^{2^i} + 1$ pre ktoré máme horný odhad $2 \cdot 2^{2^{i+4}}$ stlačení tlačidla. No a $2 \cdot 2^{2^{i+4}} < 2n^{16}$. A teda môžeme smelo prehlásiť, že bez ohľadu na to, aké veľké je n, Šimon vždy dostane macka do svojich rúk po menej ako $2n^{16}$ stlačeniach tlačidla. Na záver ešte raz zdôrazníme, že tento odhad je veľmi voľný, keďže sme uprednostnili stručnosť vzorcov pred tesnosťou odhadov.

TRIDSIATY ŠTVRTÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2018