Korešpondenčný seminár z programovania
XXXVI. ročník, 2018/19
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola letnej časti

kubik (kubik@ksp.sk)

1. Reaktor v problémoch

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

Problémy v reaktore našťastie nie sú tentokrát také zlé. Niektorým ľudom ale dali trochu zabrať.

Jedným z riešení je skúsiť každý interval so všetkými ostatnými a skontrolovať, či sa prekrývajú. Toto riešenie ale nie je moc efektívne, pretože porovnávame každý interval s každým iným, čo je $O(n^2)$. Keďže n môže byť až milión, toto nie je úplne postačujúce na plný počet bodov.

Kľúčovým pozorovaním je, že časových blokov je *iba* milión. Milión je pre počítač vcelku malé číslo - bežný počítač vie urobiť približne 10^8 operácií za sekundu. Pre vyriešenie úlohy na plný počet bodov si teda stačí vytvoriť jedno pole veľkosti n (najviac milión) a označovať si, v ktorých blokoch už je žiarovka zapnutá. Pokiaľ narazíme na druhé zapnutie žiarovky na nejakom bloku (inými slovami, nájdeme kolíziu), vypíšeme `ANO`, pokiaľ sme žiadnu kolíziu nenašli, vypíšeme `NIE`.

Toto riešenie má časovú zložitosť $O(n)$, keďže na každý blok sa pozrieme jedenkrát. Pamäťovú zložitosť má tiež $O(n)$, keďže potrebujeme vytvoriť pole veľkosti n. Treba si dávať pozor, aj keď n môže byť iba milión, nejedná sa o konštantú časovú zložitosť. Toto ''umelé" obmedzenie n je iba z dôvodov testovania vašich riešení.

Listing programu (Python)

```
#!/usr/bin/env python3
n, m = map(int, input().split())

bloky = [False] * 10**6
for _ in range(m):
    a, b = map(int, input().split())
    for i in range(a, b):
        if bloky[i] is True:
            print("ANO")
            exit(0)
        else:
            bloky[i] = True

print("NIE")
```

Hodobox (hodobox@ksp.sk)

2. Ako sa časy menia

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

Skúšame všetky heslá

Priamočiare riešenie ktoré vyrieši prvú sadu je jednoducho vyskúšať všetky Masívne heslá. Tých je 2^N, keďže každé číslo v Masívnom hesle buď je, alebo nie je, a teda za každé možné číslo sa počet možností zdvojnásobí.

Vyskúšať všetky možnosti môžeme napríklad jednoduchou rekurzívnou funkciou, v ktorej si udržiavame zoznam čísel na ktorý sa chceme opýtať (najprv prázdny), a číslo o ktorom sa práve rozhodujeme či ho do zoznamu pridáme, alebo nie. Zakaždým vyskúšame obe možnosti – pridať toto číslo do zoznamu a rekurzívne rozhodnúť nad ďalším, alebo ho nepridať a rekurzívne rozhodnúť nad ďalším. Keď už sme sa rozhodli nad všetkými N číslami, opýtame sa testovača, a keď nám dá kladnú odpoveď, uložíme si tento zoznam ako kandidáta na Masívne heslo práve vtedy, keď pozostáva z viac čísel ako globálny zoznam v ktorom si udržiavame momentálnu odpoveď. Masívne heslo je totiž práve taká sada čísel na ktorú dostaneme JOP, ktorá má v sebe čísel najviac.

Po vyskúšaní všetkých odpovedí oznámime testovaču najdlhšie heslo na ktoré sme dostali JOP.

Hesiel musíme vyskúšať $O(2^N)$, pričom zakaždým musíme vypísať až $O(N)$ čísel v otázke. Časovú zložitosť teda máme $O(2^N)$. Pamätáme si pritom jeden globálny zoznam a jeden zoznam v rekurzií o dĺžke najviac N, a konštantne veľa premenných, čiže pamäťe zaberieme $O(N)$.

Vylepšenie pre druhú sadu

Malou úpravou horeuvedeného riešenie vieme vyriešiť aj vstupy druhej sady, kde máme zaručené že počet platných Masívnych hesiel je malý. Ak teda v rekurzií nebudeme skúšať pridať do zoznamu ďalšie čísla, ak už by presiahol dĺžku M, vyskúšame len týchto zopár platných hesiel a získame nejaké body navyše.

Časová zložitosť je vo všeobecnom pripade rovnaká, ale pre M o dosť menšie ako N bude tesnejší odhad na opýtané otázky N nad M.

Optimálne riešenie

Optimálnym riešením, ktoré sa dokonca programuje ľahšie ako už spomínaná rekurzia, je všimnúť si že sa môžeme opýtať na všetky heslá o dĺžke 1. Odpoveď na otázku so samotným číslom i nám potom priamo povie, či sa v Masívnom hesle i nachádza. Keď sa teda spýtame na všetky čísla od 1 po N, teda N otázok a zároveň N čísel, budeme o každom vedieť či sa v hesle nachádza alebo nie, a tie ktoré sme zistili že v ňom sú môžeme všetky vypísať.

Pamäťová zložitosť tohoto riešenia je $O(N)$, keďže si musíme o každom čísle pamätať či v hesle je, alebo nie (prípadne si robíme zoznam s číslami ktoré v ňom naozaj sú, ale aj tých môže byť až N). Časová zložitosť je $O(N)$, keďže sa pýtame N otázok na konštantne veľa čísel, a nakoniec vypíšeme odpoveď v ktorej je tiež $O(N)$ čísel.

Dôkaz správnosti

Teraz si poďme dokázať, že úlohu naozaj nevieme vyriešiť na menej otázok, alebo s lepšou časovou zložitosťou.

Čo keby bolo Masívne heslo práve jedno číslo? Potom ľubovoľná otázka s viac ako jedným číslom nám neposkytne žiadnu informáciu – odpoveď je vždy NOP, a my nevieme či je v tejto otázke to správne číslo. Jediný spôsob ako to zistiť je naozaj sa spýtať na každé číslo zvlášť. Musíme sa teda opýtať aspoň N otázok.

Keďže sa musíme opýtať aspoň N otázok, a teda aspoň toľko čísel musíme vypísať, určite úlohu nevyriešime rýchlejšie ako na $O(N)$ operácií. Iný môžný dôkaz je to, že v heslo môžu byť všetky čísla, a bezohľadu na to ako to uhádneme, budeme ich musieť všetky vypísať.

Listing programu (C++)

```
#include <iostream>

using namespace std;

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

        bool heslo[N];
        int velkost = 0;

        for(int i=0;i<N;++i)
        {
                cout << "1 " << i+1 << endl;
                string odpoved;
                cin >> odpoved;

                if(odpoved == "JOP")
                {
                        heslo[i] = true;
                        velkost++;
                }
                else
                        heslo[i] = false;
        }

        cout << "-1" << endl;

        cout << velkost;

        for(int i=0;i<N;++i)
        {
                if(heslo[i])
                        cout << " " << i+1;
        }
        cout << endl;

}
```

Samo (`samoc@ksp.sk`)
(max. 0 b za popis, 20 b za program)

3. Dehumanizovaná klasifikácia

Ako ste si rozhodne všimli, v zadaní nie sú jednotlivé druhy zvierat charakterizované a vašou úlohou bolo objaviť ich empiricky. Uvažujme vzorového riešiteľa Sama. Samo si po prečítaní úlohy uvedomil, že nemá veľa iných možností ako poslať aspoň nejaký program a dúfať, že sa stane niečo zaujímavé. Krvopotne teda napísal program, ktorý je v súlade s príkladom v zadaní a odoslal ho. A hľa, jeho riešenie zomrelo už na druhom vstupe s výsledkom `Zlá odpoveď`. Testovač ale Sama obdaril fotografiou, na ktorej jeho program zlyhal, tvrdiac, že je to ovca:

```
4
      _
 /mmmmmO
o wwwww
  || ||
```

Samo si uvedomil, že nie je ideálne mať celú fotografiu uloženú v programe a vstup s ňou porovnať, ale radšej ich odlíšiť na základe nejakej jednoduchej vlastnosti. Vlastnosť, ktorú si všimol ako prvú, je počet riadkov obrázka. Zatiaľ čo fotka hada má jeden riadok, fotka ovce má štyri. A tak napísal nový program, ktorý vie rozoznať ovcu a hada. Takto chvíľu pokračoval, nasledovala stonožka, ryba a mačka, no vtom narazil na problém. Stonožka aj mačka majú obe tri riadky. No Samo je šikovný a rýchlo si všimol, že mačka má, narozdiel od stonožky, uši. A tak, pomocou operátora `in` v pythone a pomocou funkcie v C++, ktorú si pripravil, skontroloval, či sa v prvom riadku obrázka nachádzajú uši. Následne stretol niekoľko variácii už známych zvierat, barany a veľryby, ktoré rozoznal od svojich náprotivkov podobne ako mačku od stonožky. Potom prišiel mravec, čo bola vlastne stonožka dlhá presne tri články. Samo sa teda jednoducho pozrel na šírku obrázka.

Po ošetrení prípadu mravca sa potešil, pretože už získal za úlohu 8 bodov, no čakalo ho nemilé prekvapenie. Ako sa aj zamestnanci organizácie PETA obávali, reaktor na matfyze pôsobil na ZOO nadmernou radiáciou a nachádzali sa v nej zmutované zvieratá:

```
1
-=+====O<
```

V týchto prípadoch mal Samov program vypísať slovo `mutant`. Rozoznať zmutovaného hada bolo celkom jednoduché, keďže len niektorá časť tela mala namiesto znaku `=` znak `+`. A tak Samo použil rovnakú metódu ako doteraz. “Aaa!” Nahlas sa zľakol dvojhlavej ovce.

```
4
-     -
OmmmmmO
 wwwww
 || ||
```

V tomto prípade sa nestačí len pozrieť, či obrázok obsahuje nejaký znak, ale bolo by ideálne výskyty spočítať. Za týmto účelom Samo použil v pythone metódu `str.count` a v C++ si vytvoril ďalšiu funkciu (prípadne použil `count`). A teda jednoducho spočítal hlavy ovce, berúc do úvahy to, že barana mohol stretnúť podobný osud. Neskôr sa Samo stretol s ovcami so zmutovanou vlnou, ktorá sa opäť dala detekovať prítomnosťou znaku.

Nasledovali stonožky s chýbajúcimi nohami. Samovi napadlo, že spočíta, koľko nôh stonožka má a porovná s tým, koľko by ich mať mala, podľa šírky obrázka. Neskôr sa objavili stonožky s hlavami na náhodných miestach:

```
3
\|||/
ooOoO
/|||\
```

Toto sa opäť dalo vyriešiť spočítaním hláv. Dvojhlavé ryby boli o niečo komplikovanejšie, keďže hlava ryby sa skladá z rovnakých znakov ako chvost (ryba môže byť aj otočená), ale v prípade, že sa v obrázku nachádzajú obidva znaky ($<$ a $>$) dvakrát, ryba musí mať dve hlavy. V prípade, že je takto zmutovaná veľryba, správnym výstupom je `velmutant`. Objavili sa mačky, ktoré nemali štyri nohy, prípadne mali viac ako dve uši, toto nebol žiadny problém, stačilo spočítať znaky. Nakoniec sa objavila ovca s trochu inou mutáciou vlny, no problém sa neodklonil od spočítania výskytov niektorého znaku. Týmto Samo dokončil svoj program na rozoznávanie druhov a získal 20 bodov. Celkovo mu to trvalo 18 submitov.

Listing programu (Python)

```
n = int(input())
o = [input() for _ in range(n)]
if n == 1:
    if '+' in o[0]:
        print("mutant")
    else:
        print("had")
elif n == 2:
    a = "ryba"
    if o[1].count('<') == o[1].count('>') == 2:
```

```
        a = "mutant"
    if 'v' in o[0]:
        print("vel"+a)
    else:
        print(a)
elif n == 3:
    if '^' in o[0]:
        if o[2].count('|') != 4 or o[0].count('^') != 2:
            print("mutant")
        else:
            print("macka")
    else:
        if len(o[0]) == 3:
            print("mravec")
        else:
            if o[0].count('|') < len(o[0])-2 or o[2].count('|') < len(o[2])-2 or o[1].count('O') > 1:
                print("mutant")
            else:
                print("stonozka")
else:
    if o[1].count('O') > 1 or 'W' in o[2] or 'M' in o[1]:
        print("mutant")
    elif '@' in o[0]:
        print("baran")
    else:
       print("ovca")
```

Marekc (marekc@ksp.sk)
(max. 12 b za popis, 8 b za program)

4. i-Build

Objem, obsah alebo dĺžka?

"Stavebnica sa skladá z hranolov s rozmermi 1 cm, 1 cm a n cm."

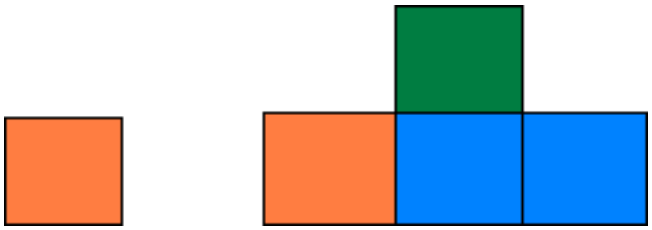
Síce máme zistiť objem staebnice, zaujíma nás len jeden rozmer každého dieliku, keďže tie zvyšné dva sú stále konštantné 1 cm a 1 cm. Zaujímajú nás teda len dĺžky všetkých hranolov. Tiež vieme, že súčet dĺžok hranolov sa rovná súčtu objemov hranolov, čiže celkovému objemu stavebnice, keďže prierez každého hranolu je štvorec s rozmermi 1 cm a 1 cm.

Dobre. Aké dĺžky majú hranoly stavebnice? Pripomeňme si pravidlá:

- Vrch veže je kocka s rozmermi 1 cm, 1 cm a 1 cm.
- Medzi podstavami (spodkami) veží je medzera 1 cm.
- Medzi koncom hranola a podstavou veže je 1 cm.

Príklad

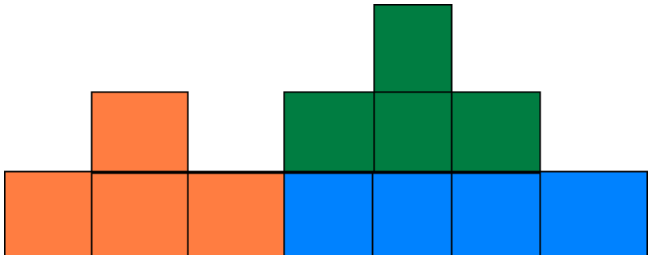
Hranoly si rozdelíme na kocky s rozmermi 1 cm, 1 cm a 1 cm. Postupne budeme do stavebnice pridávať nové časti a ukážeme si, ako sa podľa toho menia dĺžky hranolov.



Najprv mala stavebnica len jednu malú vežu, čiže jeden hranol (oranžový) o dĺžke 1 cm. Potom sme ale naň pridali ešte jeden hranol (zelený) a vytvorili tak vežu o výške 2. Tým pádom sa spodný hranol predĺžil na 3 cm. Prečo konkrétne?

Predstavme si, že sme najprv priložili zelenú kocku tak, aby vľavo od nej ešte bola medzera 1 cm na oranžovej kocke. Tým pádom je z ľavej strany splnené tretie pravidlo - medzi koncom spodného hranol a zeleným hranolom je 1 cm.

Zelená kocka by teraz ale levitovala vo vzduchu, pod ňou by ostalo prázdne miesto. Preto pod celú dĺžku zleného hranola dáme modré kocky. A aby sme splnili tretie pravidlo aj z pravej strany, pridáme ešte jednu kocku vpravo.



Teraz sa v stavebnici objavila nová (zelená) veža. Je uložená na už existujúcom hranole (oranžový), ktorý sa predĺžil o modrú časť. Všimnime si, že modrá časť má dlžku zase šírky podstavy novej veže a jednu krajnú kocku naviac.

Dĺžka hranola

Takže akú dlžku má hranol stavebnice? Vždy aspoň 1 cm. Pre každý hranol na ňom prirátame jeho dĺžku a 1 cm za kocku vpravo, aby medzi koncom spodného a horného hranola bola medzera 1 cm alebo aby existovala medzera 1 cm medzi dvoma susednými vežami. Z toho nám teda vychádza akýsi vzorec: 1 + súčet dĺžok podstáv (šírok) veží + počet veží.

Ak si túto stavebnicu predstavíme ako strom, kde každý vrchol je jeden hranol stavebnice, a rozdelíme si ho na podstromy, tak šírka každého podstromu závisí od jeho podstromov. Každý podstrom reprezentujúci vežu má nejakú šírku, čiže dĺžku hranola v jeho koreni – dĺžku podstavy.

Rekurzia

Keďže vo "vzorci" na výpočet dĺžky jedného hranola používame šírky veží na ňom (dĺžky ďalších hranolov), ktoré závisia od dĺžok ďalších hranolov nad nimi atď., vidíme tu akýsi rekurzívny princíp.

Na to aby sme zistili dĺžku koreňa nejakého podstromu, musíme najprv zrátať dĺžky jeho synov a ich synov atď.

Táto rekurzia ale musí mať konečnú hĺbku. Pre ktoré hranoly/vrcholy vieme určiť ich dĺžku bez potreby spúšťania ďalšej rekurzie? Hmm... Listy tohto stromu, čiže vrcholy veží, majú vždy dĺžku 1 cm. To sedí aj podľa nešho vzorca. Ak na hranole nie sú žiadne ďalšie hranoly, má dĺžku 1 cm.

Keďže je zaručené, že graf stavebnice na vstupe bude strom, a vieme, že každý strom určite má listy, máme istotu, že vytvorená rekurzia sa nezacyklí.

Zložitosti

Takouto rekurziou (alebo prehľadávaním stromu do hĺbky) navštívime každý vrchol práve raz, čiže časová zložitosť tohto riešenia je tak lineárna $O(n)$, kde n je počet vrcholov grafu, počet hranolov v stavebnici.

Pamäťová zložitosť tohto riešenia je tiež lineárna $O(n)$, keďže nám stačí pamätať si zoznam susedov pre daný graf. Vieme, že strom s n vrcholmi má $n - 1$ hrán.

Listing programu (C++)

```
#include <cstdio>
#include <vector>

using namespace std;

vector <vector <int> > Edge;
long long result;

long long dfs(int node, int parent) {
        int size = 0;

        for (int i : Edge[node])
                if (i != parent)
                        size += dfs(i, node);

        result += size + Edge[node].size();
        return size + Edge[node].size();
}

int main() {
        int t;
        scanf("%d", &t);
        for (int j = 0; j < t; j++) {
                int n;
                scanf("%d", &n);

                Edge.resize(n);
                Edge.clear();
                for (int i = 0; i < n - 1; i++) {
                        int a, b;
```

```
                scanf("%d %d", &a, &b);
                Edge[a].push_back(b);
                Edge[b].push_back(a);
            }

            result = 0;
            dfs(0, -1);
            printf("%lld\n", result + 1);
        }
}
```

mišof (misof@ksp.sk)

5. Ááále, to sa spraví...

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

Začneme tým, že si vyriešime jednoduchšiu úlohu: odignorujeme zatiaľ, že máme nejakých internov, a budeme sa zaujímať len o to, koľko najmenej *čísel* treba zmeniť, aby medián vyšiel presne m.

Predstavme si, že sme si všetky hodnoty usporiadali a pozreli sme sa do stredu poľa. Ak v strede poľa vidíme želanú hodnotu m, netreba meniť nič.

Ak vidíme hodnotu menšiu ako m, máme v poli priveľa malých čísel. Zistime si teda, pokiaľ v našom poli siahajú čísla menšie ako m. Nech sú primalé čísla nielen v strede poľa, ale aj na nasledujúcich $x-1$ políčkach. Čo to pre nás znamená? Aspoň x spomedzi čísel v pôvodnom poli nutne musíme zväčšiť na aspoň m, inak budeme mať ešte stále priveľa malých čísel a v strede usporiadaného poľa bude naďalej primalá hodnota.

Na druhej strane však ľahko nahliadneme, že keď zmeníme *ľubovoľných* x primalých hodnôt na hodnotu presne m, dostaneme platné riešenie. Totiž keď po tejto zmene preusporiadame prvky, prvky s novou hodnotou m obsadia presne x posledných políčok z úseku, v ktorom dovtedy boli primalé hodnoty – a teda aj v strede poľa už dostaneme hodnotu m.

Situácia, keď sme v strede poľa uvideli priveľkú hodnotu, sa rieši symetricky.

Späť k pôvodnej úlohe

Bez ujmy na všeobecnosti predpokladajme, že súčasný medián je primalý, potrebujeme teda niektoré prvky zväčšiť. A vyššie popísaným spôsobom vieme nájsť x také, že určite treba spraviť aspoň x zmien.

Pre každého interna si teraz zistíme, koľko má medzi svojimi meraniami čísel menších ako m. Nanajvýš toľko ich samozrejme tento konkrétny intern vie zväčšiť na m. My hľadáme takú sadu internov, aby ich bolo čo najmenej a aby dokopy vedeli zväčšiť aspoň tých potrebných x hodnôt. Je zjavné, že toto vieme spraviť pažravo: usporiadame si internov podľa toho, koľko hodnôt vedia zväčšiť na m, a budeme ich postupne brať (začínajúc tým, ktorý vie zmien spraviť najviac) až kým nebudú dokopy vedieť spraviť dostatočný počet zmien.

Zjavne platí, že keď nami vybraná skupina internov spraví ľubovoľných x zmien (hodnoty menšej ako m na hodnotu m), dostaneme platné riešenie. A tiež platí, že žiadna menšia skupina internov takto veľký počet zmien nevie spraviť, nájdené riešenie je teda optimálne.

Implementácia

K implementácii už len podotknem, že usporadúvať samotné čísla vôbec netreba – totiž si stačí v lineárnom čase spočítať, koľko je primalých a koľko je priveľkých. Pri usporadúvaní internov podľa počtu dobrých zmien, ktoré vedia spraviť, som použil CountSort, celkovo má teda nižšie uvedená implementácia časovú zložitosť lineárnu od veľkosti vstupu: $O(ab)$.

Listing programu (Python)

```
A, B, M = [ int(_) for _ in input().split() ]
data = [ [ int(_) for _ in input().split() ] for a in range(A) ]

pocet_mensich = sum( x<M for row in data for x in row )
pocet_vacsich = sum( x>M for row in data for x in row )
max_moze      = (A*B - 1) // 2
zmenit        = 0
zmenitelny    = lambda x: False

if pocet_mensich > max_moze:
    zmenit = pocet_mensich - max_moze
    zmenitelny = lambda x: x < M

if pocet_vacsich > max_moze:
    zmenit = pocet_vacsich - max_moze
    zmenitelny = lambda x: x > M

# pocet_internov[x] bude pocet internov, ktori vedia spravit presne x zmien
pocet_internov = [ 0 for _ in range(B+1) ]
for row in data:
    pocet_internov[ sum( zmenitelny(x) for x in row ) ] += 1

vybrat = 0
```

```
for b in range(B,0,-1):
    while zmenit > 0 and pocet_internov[b] > 0:
        pocet_internov[b] -= 1
        zmenit -= b
        vybrat += 1

print(vybrat)
```

Žaba (zaba@ksp.sk)
(max. 12 b za popis, 8 b za program)

6. Centrifúgy sa dotočia

Na začiatku stojí pomerne nemožná úloha – vytvoriť slovo `X`, ktoré obsahuje a zároveň neobsahuje zadaný reťazec `S`. Pointa úlohy však spočíva v tom, že kontrola toho, či `X` obsahuje `S` je implementovaná zle. Ako prvé je teda potrebné zistiť, kde je problém v zadanej implementácii. Našťastie, k dispozícii máme aj ukážkový vstup, ktorý nefunguje.

Listing programu (Python)

```
def obsahuje(S, X):
    ps, px = 0, 0
    while True:
        if ps == len(S):
            return True
        if px == len(X):
            return False
        if S[ps] == X[px]:
            ps += 1
            px += 1
        elif ps == 0:
            px += 1
        else:
            ps = 0
```

Prečo vyššie uvedený kód prehlási, že text `AAAB` neobsahuje `AAB`? Premenná `px` označuje pozíciu v reťazci `X`, ktorú práve kontrolujeme. Premenná `ps` predstavuje to isté v reťazci `S` a ukazuje, koľko posledných písmen v `X` a `S` sa zhodovalo. Na pozícii `px` by teda mal končiť reťazec, ktorého posledné písmená sú zhodné s prvými `ps` písmenami `S`. Ako je to však v našom prípade?

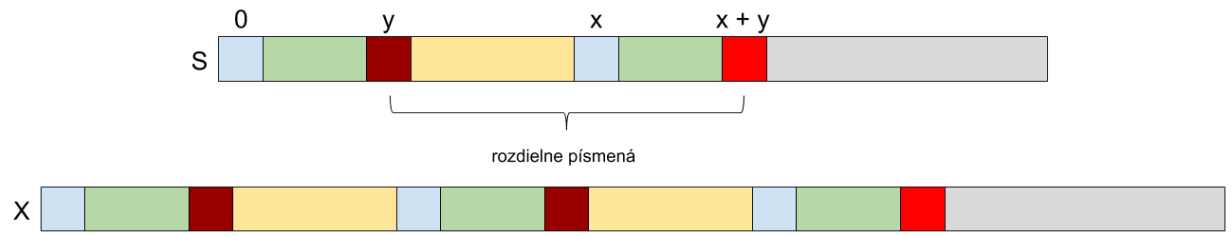
Zo začiatku všetko ide správne a hodnoty premenných sú `px = 2` a `ps = 2`. To je naozaj správne, prvé dve pozícii `X` sa zhodujú s prvými dvoma pozíciami `S`. Ďalšie písmená sa však už nezhodujú, preto program prejde do `else` podmienky, kde nastaví `ps = 0`. Pri ďalšej kontrole bude platiť, že `S[ps] = X[px]` a teda `px = 3` a `ps = 1`. Teda koniec úseku dĺžky tri `AAA` sa zhoduje so začiatkom slova `S` dlhým 1, teda `A`. To je samozrejme pravda, platí však aj silnejšie tvrdenie, že sa zhoduje až s dvoma písmenami `S`. Hodnota `ps` by teda mala byť 2.

A to je presne problém tejto implementácie. V tomto okamihu je totiž `ps` pozadu a už to nemá ako dobehnúť. Pri ďalšom porovnaní správne zhodnotí, že `A` sa nerovná `B`, ale znak `A` porovnáva len preto, že hodnota `ps` nie je najväčšia možná. To nám napovedá, ako túto úlohu riešiť všeobecne. Zadaný program totiž dokážeme oklamať ak vhodne spojíme dve kópie zadaného slova. Zo začiatku sa bude všetko zhodovať, v nejakom momente, bez toho aby si to program všimol, však prejdeme do druhého slova. V okamihu, keď program nájde chybu, bude už neskoro, pretože sme už prešli začiatok slova `S`, ktoré sa v `X` nachádza a program sa už nevie vrátiť.

Ak chceme spojiť dve kópie slova `S` tak, aby si zadaný program nevšimol, že prešiel z jedného do druhého, najdôležitejšie je zamaskovať prvé písmeno `S`. Napríklad, ak sa prvé písmeno `S` v ňom nachádza iba raz, toto sa nám nepodarí a centrifúgu nepokazíme (napr. slovo `CENTRIFUGA`).

Všeobecný tvar nášho riešenia teda musí vyzerať nasledovne: Majme slovo `S`, pozíciu x takú, že `S[0] = S[x]` a y je najmenšie také číslo, pre ktoré platí `S[y] != S[x+y]`. Potom slovo, ktoré dostaneme zreťazením reťazcov `S[0:x]` (prvých x písmen slova `S`) a `S`, pokazí centrifúgu.

Zadaný program začne spracovávať toto slovo a prvých $x + y$ písmen sa bude zhodovať. Potom však nájde chybu a nastaví `ps = 0`. Podľa správnosti by však hodnota `ps` mala byť y, preto do konca slova nedosiahne `ps` dostatočnú hodnotu.



Netreba ešte zabudnúť na to, že zadanie po nás chcelo najkratšie možné slovo. Všimnite si však, že dĺžka slova závisí iba od hodnoty x, tým pádom sa snažíme nájsť čo najmenšie vhodné x, číže prvý vhodný výskyt znaku `S[0]` v reťazci `S` (vynímajúc $x = 0$).

V tomto momente vieme implementovať jednoduché $O(n^2)$ riešenie. Postupne vyskúšame každé možné x také, že `S[0] = S[x]` a skontrolujeme, či vzniknuté slovo naozaj spôsobí pokazenie centrifúgy. To spravíme buď tak, že nájdeme hodnotu y takú, že `S[y] != S[x+y]` alebo proste vložíme vytvorené slovo do funkcie zo zadania. Uvedomme si, že toto overenie je skutočne potrebné, pretože funkcia zo zadania musí v niektorom momente nájsť dva rôzne znaky. Slovo `ABAB` je príkladom slova, pri ktorom centrifúgu nevieme pokaziť.

K optimálnemu riešeniu nám už chýba iba jedno pozorovanie. V skutočnosti totiž stačí overiť iba prvé $x > 0$, pre ktoré `S[0] = S[x]`. Buď sa nám totiž podarí vytvoriť vhodné slovo, a tým pádom máme najkratšie možné riešenie, alebo zistíme, že slovo `S` je istým spôsobom periodické a úloha preň nemá riešenie.

V tom druhom prípade totiž platí, že `S[i] = S[x + i]` pre všetky možné i. To ale vynucuje periodickosť slova `S`, bude totiž platiť aj to, že `S[x + i] = S[x + x + i]`, a teda aj `S[i] = S[2x + i]`. Tým pádom, rovnaký znak ako je na pozícii 0 je práve na pozíciách `S[x]`, `S[2x]`... Pre žiadne $k > 1$ však potom nemôže platiť, že `S[0] = S[kx]` a zároveň existuje y také, že `S[y] != S[kx + y]`.

Optimálne riešenie je potom naozaj jednoduché. Stačí nájsť prvú hodnotu $x > 0$, na ktorej je znak zhodný s prvým písmenom `S` a overiť, či reťazec `S[0:x]S` pokazí centrifúgu. Na to nám stačí časová zložitosť $O(n)$.

Listing programu (Python)

```
s = input()

kde = -1
for i in range(1, len(s)):
    if s[i] == s[0]:
        kde = i
        break

if kde == -1:
    print('Centrifuga sa nepokazi!')
else:
    poz = 0
    while kde + poz < len(s) and s[kde + poz] == s[poz]:
        poz += 1
    if kde + poz == len(s):
        print('Centrifuga sa nepokazi!')
    else:
        print(s[:kde] + s)
```

Hodobox (`hodobox@ksp.sk`)
(max. 12 b za popis, 8 b za program)

7. Ideálne cestovanie metrom

Krátka rekapitulácia úlohy: Máme ovahovaný graf – n staníc metra prepojených m tunelmi rôznej nebezpečnosti – a q otázok na dve stanice, odpoveďou je také číslo r, že nevieme precestovať medzi týmito dvoma stanicami bez toho, aby sme použili tunel s nebezpečnosťou aspoň r.

Hrubá sila

V prvej sade máme všetkého – staníc, tunelov, otázok – najviac 15. Môžeme si teda dovoliť skoro hocičo. Napríklad si môžeme pre každú otázku vyskúšať všetky podmnožiny tunelov, overiť či sa vieme dostať medzi dvoma stanicami z otázky, a ak áno, pozrieť aký najnebezpečnejší tunel sme v podmnožine nechali. Odpovedáme najmenšiou nebezpečnosťou s ktorou sa nám to niekedy podarí. Takéto niečo by zbehlo v $O(q \cdot (n + m)2^n)$.

Jediná správna cesta

Predstavte si, že začíname s metrom v ktorom niesu vybudované žiadne tunely, a teda sa medzi žiadnymi rôznymi stanicami nevieme dostať. Začneme teraz budovať všetky tunely od najmenej nebezpečnejšieho po najnebezpečnejší. Vždy keď ideme stavať tunel, pozrieme sa či už sa medzi danými dvoma stanicami vieme nejako dostať, a ak áno, nepostavíme ho. V istom bode prepojíme celé metro, čiže sa vieme dostať medzi všetkými dvojicami staníc, a žiadny z nevystavaných tunelov by sme ajtak nepoužili – každý z nich je nebezpečnejší ako nejaká množina tunelov, ktoré nám dovoľujú sa pohybovať medzi tými stanicami ktoré tento nebezpečný tunel spája. Získame jednoducho najlacnejšiu kostru mesta, ktorá má stromovú štruktúru.

Najmenej nebezpečná cesta medzi dvoma stanicami a a b je potom presné tá jediná cesta v našej vystavanej kostre, ktorá medzi nimi vedie. Jediné čo potrebujeme zistiť je najdrahšia hrana, ktorá na nej leží.

Túto informáciu si môžeme predpočítať – keď dostaviame kostru, spustíme v cykle s každej stanice prehľadávanie do šírky, v ktorom si navyše pamätáme akú najnebezpečnejšiu hranu sme dosiaľ použili, aby sme sa

dostali na danú stanicu. Vždy keď použijeme tunel, túto hodnotu prepíšeme na maximum z nej, a nebezpečnosti tohoto tunela, a túto hodnotu si zapíšeme do tabuľky. Keď nám prehľadávanie do šírky skončí, máme $n \times n$ tabuľku, v ktorej na indexe `[a][b]` máme zapísanú najdrahší tunel, ktorý bol prehľadávaním použitý na cesta z a do b.

Každú otázku potom jednoducho odpovedáme v konštantnom čase.

Pamäťová zložitosť nášho riešenia je $O(n^2)$ a časová zložitosť je $O(m \log n)$ na postavanie najlacnejšej kostry, $O(n^2)$ na prehľadávania do šírky a $O(q)$ na odpovedanie otázok. Dokopy teda $O(n^2 + q)$, keďže m n.

Ak sa cítime trochu lenivejšie, môžeme namiesto stavania najlacnejšej kostry rovno pustiť prehľadávanie dijkstrou, v ktorej preferujeme vrcholy ktoré sme dosiahli s najmenšou nebezpečnosťou, čo nám síce o logaritmus zhorší zložitosť, ale v druhej sade prejde.

Listing programu (C++)

```
Zdrojové kódy budú zverejnené až po 19.8. 23:59:59. Dovtedy sa môžete pokúsiť úlohu naprogramovať sami
(a získať tým nejaké body navyše).
```

A však máme strom

Keďže najlacnejšia kostra nášho metra je nutne strom, môžeme siahnuť po nejakom algoritme ktorý na stromoch vieme púšťať. Každá otázka požaduje zistiť informáciu o nejakej ceste v našom strome – najnebezpečnejšiu hranu na nej. Táto informácia má dobrú vlastnosť, že keď máme dve cesty ktoré sa napájajú (cesta z a do b a z b do c) a pre ktoré vieme odpoveď, a spojíme ich do jednej cesty (a do c), tak pre ňu vieme ľahko spočítať odpoveď – je to maximum z odpovedí pre jednotlivé cesty.

Algoritmus ktorý takéto niečo spokojne zráta je LCA – najnižší spoločný predok[1]. Ak tento algoritmus neovládate, určite si o ňom prečítajte, zvyšok vzoráku ho len priamočiaro použije. Zakoreníme si našu najlacnejšiu kostru o ľubovoľný vrchol, a pri predrátavaní si predkov si rátame aj najnebezpečnejšiu hranu v každom jednotlivom skoku.

Keď dostaneme dotaz na cestu medzi stanicami a a b, poskáčeme našimi skokmi mocnín dvojky k ich najnižšiemu spoločnému predkovi, čo teda predstavuje niekoľko ciest ktoré sa dokopy napoja na práve tú jedinú cestu medzi nimi v našom strome, a naša odpoveď je najväčšie číslo ktoré v týchto skokoch stretneme. Každý teda odpovieme v logaritmickom čase.

Pamäťová zložitosť je $O(n \log n)$ na logaritmický počet skokov z každej stanice.

Postaviť kostru nám trvá $O(m \log n)$, predrátať skoky na predkov $O(n \log n)$, a odpovedať na všetky dotazy $O(q \log n)$, celkovo teda $O((m + n + q) \log n)$.

Listing programu (C++)

```
Zdrojové kódy budú zverejnené až po 19.8. 23:59:59. Dovtedy sa môžete pokúsiť úlohu naprogramovať sami
(a získať tým nejaké body navyše).
```

Jano (janoh@ksp.sk)

8. Anihilácia

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

Reťazec písmen S dokážeme celý zmazať práve vtedy, keď platí aspoň jedna z týchto podmienok:

- S je prázdny reťazec
- S má dĺžku aspoň 2, začína a končí tým istým písmenom, a keď zmažeme všetky výskyty tohto písmena zo začiatku a z konca, dostaneme reťazec, ktorý sa dá zmazať celý. (Čiže napr. "aaaaaUaaa", kde U sa dá zmazať, alebo "bbbbbb").
- S sa dá rozseknúť na neprázdne reťazce U a V, aj U aj V sa dajú celé zmazať.
- S sa dá rozseknúť na 5 častí x, U, x, V, x, kde x je ľubovoľné písmeno a U aj V sú neprázdne reťazce, ktoré sa dajú celé zmazať.

Dôkaz:

Ľahko overíme, že ak reťazec spĺňa niektorú podmienku, dá za zmazať celý. Stačí nám ukázať, že ak sa reťazec dá zmazať, musí spĺňať aspoň jednu podmienku, čo skúsime ukázať sporom. Majme reťazec, ktorý sa dá zmazať celý ale nespĺňa žiadnu zo štyroch podmienok.

Keď by sme tento reťazec mazali, ako posledné nám musel ostať úsek aspoň dvoch rovnakých písmen, označme si toto písmeno x. Pôvodný reťazec má tvar $T_0 x_1 T_1 x_2 ... x_k T_k$, kde x_i je niekoľko znakov x a T_i je reťazec, ktorý sa dá celý zmazať (lebo na konci nám ostali len znaky x.).

[1] https://www.ksp.sk/kucharka/lca/

Ak T_0 resp. T_k sú neprázdne, tak daný reťazec sa dá rozseknúť na T_0 a zvyšok, resp. T_k a zvyšok a teda reťazec spĺňa tretiu podmienku (spor). Ďalej teda predpokladajme, že T_0 aj T_k sú prázdne.

Ak k je 1, máme reťazec tvaru x_1, ktorý spĺňa druhú podmienku (spor).

Ak k je 2, máme reťazec tvaru $x_1T_1x_2T_2x_3$. Ak ktorýkoľvek z x_1, x_2, x_3 má aspoň dva znaky, tak reťazec spĺňa tretiu podmienku, inak spĺňa štvrtú (spor).

Ak k je viac ako 2, reťazec spĺňa tretiu podmienku (spor).

□

Ľahko napíšeme rekurzívnu funkciu, ktorá overí všetky tieto podmienky v čase $O(n)$, kde n je dĺžka reťazca, plus čas strávený v rekurzívnych volaniach. Pridáme memoizáciu, aby sme pre každý podúsek vstupného reťazca počítali odpoveď (či sa dá celý zmazať) len raz. Takto dosiahneme časovú zložitosť $O(n^3)$.

Pamäťová zložitosť je $O(n^2)$.

Listing programu (Python)

```
import sys
sys.setrecursionlimit(1000000)

# memoizacia
def sol(S, a, b):
    if memo[a][b] == -1: memo[a][b] = _sol(S,a,b)
    return memo[a][b]

# da sa odstranit S[a:b]?
def _sol(S, a, b):
    # prazdny retazec je dobry
    if b-a == 0:
        return True

    # a..aUa..a je dobre, ak U je dobre
    if b-a > 1 and S[a] == S[b-1]:
        aa, bb = a, b
        while aa < b and S[aa] == S[a]:
            aa += 1
        while bb > aa and S[bb-1] == S[b-1]:
            bb -= 1
        if sol(S, aa, bb):
            return True

    for i in range(a+1, b-1):
        # UV je dobre, ak U aj V su dobre
        if sol(S, a, i) and sol(S, i, b):
            return True

        # aUaVa je dobre, ak U aj V su dobre
        if S[a] == S[i] and S[i] == S[b-1] and sol(S, a+1, i) and sol(S, i+1, b-1):
            return True

    return False

t = int(input())
for i in range(t):
    s = input()
    n = len(s)
    memo = [[-1 for i in range(n+1)] for j in range(n+1)]
    print(('nie', 'ano')[sol(s,0,n)])
```