Korešpondenčný seminár z programovania
XXIX. ročník, 2011/12
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola letnej časti

opravoval Jano

1. Zmätený bocian

(max. 7 b za popis, 3 b za program)

Ako prvé by som chcel pochváliť riešiteľov za veľké množstvo správnych riešení. Na druhej strane, mohli by ste písať popisy trochu poriadnejšie: čím ďalej, tým sa viac blížite k hranici, že stratíte bod za neporiadnosť. Tentokrát som za to body nestŕhal – prižmúril som oko, lebo to bola jednotka a nebolo tam potrebné sa nejako vykecávať. Bububu, keď sa nezlepšíte, tak pôjdu body dolu :) Tak, keď ste sa už nad sebou zamysleli, môžeme sa pustiť do riešenia.

Najprv si všimnime, že z každej atrakcie máme jedinú možnosť, ako sa pohnúť ďalej. Toto navádza na nasledovný postup: Začneme v nejakej atrakcii, z nej prejdeme do ďalšej, z nej do ďalšej a takto skáčeme napríklad dovtedy, kým nás to neprestane baviť.

A keď už skáčeme, určite príde vhod zapamätať si, kde sme boli. Preto vždy, keď prejdeme **do** nejakej atrakcie, tak si ju označíme ako navštívenú.

Slovíčko **do** nie je zvýraznené preto, že sem zablúdil zvýrazňovač syntaxe jazyka Pascal alebo C/C++, ale preto, že je naozaj dôležité. Keby sme si značili **z** ktorých atrakcií sme šli, tak stratíme informáciu o tom, či sa vieme dostať aj do tej atrakcie, kde sme začali (a to, že sme z nej šli, vieme aj tak, lebo sme v nej začali).

Druhá zaujímavá vec je, že pri takomto skákaní sa nám raz stane, že skočíme do už navštívenej atrakcie (nemôžeme navštevovať nové atrakcie donekonečna). Vtedy môžeme spokojne skončiť, lebo už ďalej nič nové neobjavíme.

Riešenie štýlu „prečítam zadanie a naprogramujem, čo je tam napísané“ by vyzeralo asi tak, že skúsim začať v nejakej atrakcii, robím kroky, kým nenavštívim niečo druhýkrát a overím, či sa viem dostať všade. Toto zopakujem pre všetky možné štartovné atrakcie. Takéto riešenie by malo kvadratickú časovú zložitosť a mohlo dostať najviac 5 bodov za popis.

Kto by sa potom ešte dobre zamyslel, prišiel by na to, že stačí pustiť skákanie iba raz z ľubovoľnej atrakcie, napríklad z jednotky. Náš program odpovie áno práve vtedy, keď po skončení budú všetky atrakcie navštívené, inak odpovie nie.

Teraz skúsme dokázať, že náš program odpovie správne. Ak sa dá dostať z každej atrakcie do každej, dá sa dostať aj z prvej atrakcie do všetkých, takže všetky atrakcie okrem jednotky postupne označíme ako navštívené. (Lebo označíme práve tie atrakcie, do ktorých sa vieme dostať z prvej.) Navyše sa dá z poslednej atrakcie dostať do jednotky, cez poslednú atrakciu sme prešli, takže sme museli označiť aj jednotku ako navštívenú. Program teda odpovie áno.

Druhá možnosť je, že sa z nejakej atrakcie nedá dostať do nejakej inej. Buď sa nedá do nejakej atrakcie dostať z jednotky – potom tú atrakciu nikdy neoznačíme, alebo sa z jednotky dá dostať všade ale nedá sa dostať z nejakej atrakcie do jednotky. V tomto prípade neoznačíme ako navštívenú jednotku, lebo sa do nej nemôžeme dostať. (Ak sa dá z jednotky dostať všade a zo všadiaľ sa dá dostať do jednotky, potom sa dá zo všadiaľ dostať všade – cez jednotku.) □

Časová zložitosť je $O(n)$, pretože každé políčko označíme ako navštívené najviac raz a na konci prejdeme pole len raz. Pamäťová zložitosť je tiež $O(n)$, pretože nám stačí pamätať si trasy vláčikov a navštívené vrcholy.

Listing programu (Pascal)

```
var n, i, odkial:longint;
    kam: array [1..1000047] of longint;
    navstivene: array [1..1000047] of boolean;
begin
    read(n);
    for i:=1 to n do read(kam[i]);
    for i:=1 to n do navstivene[i] := false;
    odkial := 1;
    while(navstivene[kam[odkial]] = false) do begin
        navstivene[kam[odkial]] := true;
        odkial := kam[odkial];
    end;
    for i:=1 to n do if navstivene[i] = false then begin
        writeln('Nie');
        exit;
```

```
    end;
    writeln('Ano');
end.
```

Na záver poznamenáme, že možných riešení bolo viacero, preto uvedieme ešte jedno. Program odpovie áno práve vtedy, keď sa dá dostať z atrakcie 1 do atrakcie 1 na viac než nula pohybov a navyše najkratšia takáto cesta má práve n krokov. Dôkaz správnosti nechávame na čitateľov :)

Listing programu (C++)

```
#include<cstdio>
int N, dalsi[1000047], pocet, x;
int main(){
    scanf("%d",&N);
    for(int i = 0; i<N; ++i) scanf("%d",dalsi+i+1);
    pocet = x = 1;
    while(pocet<N+47 && dalsi[x]!=1){
        pocet++;
        x = dalsi[x];
    }
    if (pocet == N) printf("Ano\n");
    else printf("Nie\n");
}
```

opravovala Maru

2. Zúfalstvo z postupností

(max. 5 b za popis, 5 b za program)

S aritmetickou postupnosťou nebol problém, stačilo kontrolovať rozdiely medzi každými dvoma nasledujúcimi číslami.

Zaujímavejšia časť úlohy bola geometrická postupnosť. Ponaučenie, ktoré vyplýva z tejto úlohy, spočíva v troch jednoduchých zásadách:

1. Ak sa dá, chceme sa vyhnúť práci s desatinnými číslami.

Oveľa lepšie je pracovať s celými. Pri reálnych číslach v počítači dochádza totiž k zaokrúhľovacím chybám, ktoré vedú k nepresnostiam a tie zas k zlým odpovediam. Áno, je pravda, že to, či je postupnosť geometrická, sa dá zisťovať pomocou kvocientu, ktorý získame z podielu prvých dvoch členov (podobne ako pri aritmetickej postupnosti). Lenže toto môže byť nepresné. Kto má chuť, nech sa napríklad pohrá s veľkými Fibonacciho číslami. Pre ne platí, že podiely dvoch po sebe idúcich čísel konvergujú k zlatému rezu. To znamená, že pre dosť veľké čísla si bude počítač myslieť, že kvocient je stále rovnaký, no nie je tomu celkom tak.

Čo s tým teda spravíme?

Upravíme zlomky :) Nech a, b, c sú tri po sebe idúce čísla geometrickej postupnosti. Platí teda, že $\frac{b}{a} = \frac{c}{b}$. Jednoduchou úpravou si túto rovnosť vieme prepísať na $b^2 = ac$. Už vidíte, ako krásne sme sa zbavili delenia? Na testovanie geometrickej postupnosti sme teda testovali za sebou idúce trojice čísel. Pozor na to, že ak máme čísla, ktoré sa zmestia do 32-bitových integerov, ich súčin sa už nemusí.

Dostávame sa k druhej programátorskej zásade: **2. Nezabudnúť na okrajové prípady.**

Poďme sa zamyslieť nad geometrickými postupnosťami, ktoré majú niekde v sebe nulu. Špeciálne, kto riešil úlohu pomocou delenia, bolo si treba dať pozor na delenie nulou. Pri postupe s násobením celých čísel zas treba dať pozor na postupnosť typu: $0, 0, 8$. Ak dosadíme do našej rovnosti, tá bude platiť, no vidíme, že postupnosť geometrická nie je. Toto ošetríme jednoduchou podmienkou: po nule nesmie nasledovať nenulové číslo.

Teraz je postup už veľmi ľahký. Jednoducho prechádzame postupnosťou a testujeme aritmetickosť aj geometrickosť. Časová zložitosť je od veľkosti vstupu závislá lineárne, pretože ho potrebujeme celý prejsť, teda zapíšeme $O(n)$. V našom riešení je pamäťová zložitosť takisto lineárna, pretože si postupnosť pamätáme v poli, čiže $O(n)$. Keby sme chceli, vystačili by sme si s konštantnou pamäťou, pretože nám stačí pamätať si iba tú časť postupnosti, ktorú práve kontrolujeme. Konštantná pamäť by sa formálne zapísala ako $O(1)$.

Na záver si dovolím ešte poslednú z dnešnej série zásad: **3. Ak sme náhodou v situácii, že si bez desatinných čísel neporadíme, v C++ používame typ double, ktorý je presnejší ako float.**

Čo by k tomu dodal Mišof? Double je ešte ako-tak tolerovateľný, ak sú čísla malé, ale float nikdy! To je ako Comic Sans v novinách. :)

Listing programu (C++)

```
#include <iostream>
#include <climits>
#include <cstdio>
#include <cstdlib>
using namespace std;

int main () {
        int n;
```

```
        cin >>n;
        long long a[n];
        for (int i = 0; i < n; i++) {
                cin >> a[i];
        }

        bool ar = true;
        long long k = a[0]-a[1];
        for (int i = 1; i < n-1; i++) {
                if (a[i] - a[i+1] != k) {
                        ar = false;
                        break;
                }
        }
        if (ar) printf ("ano\n");
        else printf ("nie\n");

        bool geom = true;
        if (a[0] == 0 && a[1] != 0) geom = false;
        if (n > 2) {
                for (int i = 0; i < n-2; i++) {
                        if (a[i+1] == 0 && a[i+2] != 0) geom = false;
                        if (a[i]*a[i+2] != a[i+1]*a[i+1]) {
                                geom = false;
                                break;
                        }
                }
        }
        if (geom) printf ("ano\n");
        else printf ("nie\n");

        return 0;
}
```

opravoval Žaba

3. Zábavná hra

(max. 7 b za popis, 3 b za program)

Vždy, keď riešite nejakú úlohu, je dobré si to najskôr začať pekne kresliť, ručne simulovať a snažiť sa zistiť, aké vlastnosti to zadanie má. Netreba túto časť podceniť, často totiž rozhoduje o tom, či preniknete do podstaty úlohy a nenecháte sa nachytať na tom, že ak niečo platí na sampli, tak to platí aj všeobecne.

A keď sa pozrieme na túto úlohu, tak si pomyslíme, že simulovať to by nemusel byť taký zlý nápad. Zoberiete si teda dvojrozmerné pole (alebo set, alebo čokoľvek, v čom si budete pamätať, ktoré farby už malo dané dieťa v ruke) a začnete odmotávať klbká. Postupne prechádzate cez postupnosti hádzania klbiek. Máte premennú i, v ktorej si udržiavate, ktoré klbko v poradí spracovávate. Vždy, keď bolo toto klbko hodené dieťaťu x, pozriete sa na všetky farby, ktoré zatiaľ dieťa x držalo. Ak sa tam už nachádza klbko i, nič sa nezmení; inak si poznačíme, že dieťa x držalo o farbu viac a že to bola farba i.

Takto by sme dostali riešenie, ktoré by bolo správne a možno nie až tak pomalé. Vždy sa však treba zamyslieť, či nerobíme niečo zbytočne. Zjavne sa musíme pozrieť na všetky hádzania, aby sme presne vedeli, koľko klbiek jednotlivé deti držali. Vylepšovať teda môžeme len časť, kde kontrolujeme, či dieťa x držalo farbu i. A tu si môžeme uvedomiť, že sa nám stačí pozrieť na poslednú farbu, ktorú dieťa x držalo v ruke. Všetky farby, ktoré držalo pred poslednou, sa totiž určite celé minuli, lebo ich spracovávame postupne. Ak dieťa už malo v ruke farbu i, tak žiadna iná farba nemôže byť posledná (lebo práve spracovávame klbko i). Naopak, ak dieťa ešte nemalo farbu i v ruke, potom naposledy držalo nejakú inú farbu y ($y < i$), a teda to chceme upraviť.

Celý algoritmus bude prebiehať tak, že si budeme upravovať dve polia. V prvom bude počet rôznych farieb, ktoré už dieťa x držalo (to bude zaznačené na x-tej pozícii), a v druhom si budeme pamätať farbu, ktorú toto dieťa držalo v ruke ako poslednú. Potom nám bude stačiť prejsť postupnosť hádzaní klbiek a upravovať tieto dve polia. Ak zistíme, že dieťa x ako posledné držalo klbko i, nič sa nezmení; inak zvýšime hodnotu v prvom poli o 1 a hodnotu v druhom nastavíme na i. Na konci len prezrieme celé prvé pole a nájdeme dieťa, ktoré držalo najviac klbiek.

Ak si označíme celkový počet hádzaní klbiek ako p, tak výsledná časová zložitosť bude $O(p+n)$ a pamäťová bude $O(n)$, lebo máme dve polia s n prvkami.

Listing programu (Pascal)

```
var n,m,i,x,id:longint;
t:boolean;
Posledny,Pocet: array[1..1000047] of longint;

begin
  read(n,m);
  for i:=1 to n do begin Posledny[i]:=0; Pocet[i]:=0; end;
  for i:=1 to m do
    begin
      t:=true;
      while t=true do
        begin
```

```
          read(x);
          if x=0 then t:=false;
          if (t=true) and (Posledny[x]<>i) then begin Posledny[x]:=i; inc(Pocet[x]); end;
        end;
    end;
  id:=1;
  for i:=1 to n do
    if(Pocet[id]<Pocet[i]) then id:=i;
  writeln(id);
end.
```

opravoval Luxusko

4. Zamotané mosty

(max. 10 b za popis, 5 b za program)

Najprv si uvedieme pojmy a pozorovania. Pre ostrov O je nasledovníkom ostrov, do ktorého vedie most z O, a predchodcom zase ostrov, z ktorého vedie most do O. Každý ostrov má práve jedného nasledovníka, ale môže mať ľubovoľný počet predchodcov. Mosty môžeme nazývať hranami a ostrovy vrcholmi, čím si mapu Jazerbajdžanu premeníme na orientovaný graf. Na prechádzke z ľubovoľného ostrova sa zrejme časom zacyklíme – z počiatočného vrcholu nemôžeme prichádzať stále do nových vrcholov, lebo máme obmedzený počet ostrovov. Z každého vrcholu sa teda dá dostať do cyklu, alebo ten vrchol priamo leží v nejakom cykle.

K efektívnemu riešeniu vedie napríklad prehľadávanie z každého vrcholu. Budeme rozoznávať tri typy vrcholov – nenavštívený, práve prehľadávaný (ešte o ňom nevieme, či leží v cykle) a už spracovaný. Z počiatočného vrcholu pôjdeme v ďalšom kroku vždy do nasledovníka. Po príchode do nenavštíveného vrcholu si ho označíme ako práve prehľadávaný. Časom prídeme do vrcholu, ktorý bol navštívený. Ak je už spracovaný, tak to znamená, že žiadny z našich práve prehľadávaných vrcholov neleží v cykle (lebo by sme sa do nich dostali už v tom prehľadávaní, v ktorom sme navštívili tento vrchol prvýkrát). Ak sme prišli do práve prehľadávaného vrcholu, našli sme cyklus – môžeme po ňom prejsť ešte raz a spočítať vrcholy v ňom. V oboch prípadoch sme na konci prehľadávania a všetky vrcholy, do ktorých sme prišli, môžeme označiť za už spracované. Nové prehľadávanie začíname, samozrejme, len z nenavštívených vrcholov.

Túto myšlienku používala väčšina riešení. Niektoré veci sa dajú pri implementácii tohoto riešenia zmeniť tak, aby nám niečo zjednodušili – môžeme číslovať jednotlivé prehľadávania (nemusíme potom dodatočne označovať prehľadané vrcholy ako už spracované) alebo v jednotlivých prehľadávaniach číslovať vrcholy v poradí, v akom ich navštevujeme (vieme potom rýchlo zistiť dĺžku cyklu, ak nejaký nájdeme).

Listing programu (Pascal)

```
const
  MAXN = 1000000;

var
  n, i, j, r: longint;
  A, V: array [1..MAXN] of longint;

BEGIN
  ReadLn(n);
  for i := 1 to n do
    Read(A[i]);

  for i := 1 to n do
    V[i] := 0;  { 0 = nenavštívený vrchol }
  for i := 1 to n do
    if V[i] = 0 then begin
      j := i;
      while V[j] = 0 do begin
        V[j] := 1;  { 1 = práve prehľadávaný vrchol }
        j := A[j];
      end;
      while V[j] = 1 do begin
        V[j] := 2;  { 2 = vrchol na cykle }
        j := A[j];
      end;
      j := i;
      while V[j] = 1 do begin
        V[j] := 3;  {3 = vrchol mimo cyklu }
        j := A[j];
      end;
    end;

  r := 0;
  for i := 1 to n do
    if V[i] = 2 then
      Inc(r);
  WriteLn(r);
END.
```

Iný prístup je vyhadzovať postupne vrcholy, do ktorých nevedie žiadna hrana, lebo tieto určite neležia v žiadnom cykle. Pri načítaní grafu si spočítame, koľko hrán vedie do každého vrcholu. Tie vrcholy, do ktorých vedie nula hrán, si pripravíme na vyhodenie. Pri vyhadzovaní vrcholu zmažeme hranu z neho, takže znížime

počet hrán do jeho nasledovníka. Ak už má aj nasledovník nula prichádzajúcich hrán, pridáme ho medzi vrcholy na vyhodenie. Postupne vyhodíme všetky vrcholy, do ktorých už nič nevedie. Na konci nám ostane niekoľko cyklov – spočítame, koľko nám ostalo vrcholov, a vypíšeme výsledok.

Prečo nám na konci ostanú len cykly? Zamyslime sa, že cyklus takýmto spôsobom nemôžeme narušiť, lebo vrchol ležiaci v cykle nikdy nebude mať nula hrán smerujúcich doň. Všetky ostatné vrcholy môžeme rozdeliť na cesty vedúce do cyklov, ktoré postupne od koncových vrcholov povyhadzujeme. Toto riešenie nájdete aj implementované nižšie.

Listing programu (Pascal)

```
var n, i, t, vyhodenych:longint;
    kam, vstupnych_hran, vyhodit: array[1..1000047] of longint;

procedure pridaj(v : longint);
begin
    if vstupnych_hran[v] = 0 then begin
        inc(vyhodenych);
        vyhodit[vyhodenych] := v
    end;
end;

begin
    read(n);
    for i := 1 to n do vstupnych_hran[i] := 0;
    for i := 1 to n do begin
        read(t);
        kam[i] := t;
        inc(vstupnych_hran[t])
    end;
    vyhodenych := 0;
    for i := 1 to n do pridaj(i);
    i := 1;
    while i <= vyhodenych do begin
        t := kam[vyhodit[i]];
        dec(vstupnych_hran[t]);
        pridaj(t);
        inc(i)
    end;
    writeln(n-vyhodenych);
end.
```

Časová aj pamäťová zložitosť uvedených prístupov je lineárna.[1] Riešenia s kvadratickou časovou zložitosťou mohli získať najviac 7 bodov. Pomalšie riešenia neboli. Za nedostatočný popis samotného algoritmu boli body dole – robíme tu síce len jednoduché prehľadávanie, ale v ňom niečo počítame a to, akým spôsobom to počítame, je dôležité a rozhodujúce o funkčnosti. Za nesprávne či chýbajúce odhady boli taktiež body dole. Nefunkčné riešenia získavali do 4 bodov, funkčné získalo vždy aspoň 5.

5. Odpornosti

opravoval Usamec
(max. 17 b za popis, 8 b za program)

Podúloha a)

V prvom rade potrebujeme jednotlivé úlohy utriediť. Väčšina slušných jazykov na to má nejakú knižničnú funkciu. Tí, čo to vo svojom obľúbenom jazyku nemajú, musia implementovať nejaké vlastné triedenie. Dobré rýchle triedenia sú napríklad *heap sort* a *merge sort* (v prípade neznalosti týchto pojmov ich odporúčam zadať do Googlu). Takéto triedenie nám usporiada prvky v čase $O(n \log n)$.

Teraz už môžeme prideľovať prácu. Triviálne riešenie prezerá pre každú úlohu všetkých úradníkov a vyberá toho, čo má najmenej práce. Toto funguje v zložitosti $O(nk)$.

Pre rýchlejšie riešenie treba použiť trochu šikovnejšiu dátovú štruktúru. Na hľadanie minima z viacerých prvkov sa javí ako ideálna halda (angl. *heap*, opäť odporúčam hľadať v starších vzorákoch alebo na Googli). Táto dátová štruktúra podporuje vkladanie prvku v $O(\log n)$ a vyberanie minima v $O(\log n)$, kde n je počet prvkov v halde. (Niektoré implementácie podporujú navyše aj úpravu prvkov, ale to nám nebude treba.)

Na začiatku teda nahádžeme všetkých úradníkov do haldy. Každý úradník bude usporiadaná dvojica (t, i), kde t je doterajší čas práce úradníka a i je jeho číslo. Úradníkov budeme porovnávať najprv podľa prvej zložky, potom podľa druhej.

Teraz pre každú úlohu postupne: Nájdeme najmenej vyťaženého úradníka, vyberieme ho z haldy, upravíme jeho vyťaženie, vložíme ho znovu do haldy. Celý tento krok zaberie čas $O(\log k)$.

Na konci len vyprázdnime haldu a nájdeme úradníka s najdlhším pracovným časom.

[1] V prvom riešení pri prehľadávaní navštívime (a preznačíme) každý vrchol iba dvakrát. V druhom riešení každý vrchol a hranu odstránime najviac raz.

Celkovo máme teda čas $O((n+k)\log(n+k))$; vzhľadom na to, že väčšinou $n > k$, tak môžeme povedať, že čas behu je $O(n \log n)$. Pamäťové nároky sú $O(n)$.

Poznámka: Keďže nám vlastne je jedno, ktorý úradník dostáva príslušnú robotu, tak si v implementácii pamätáme len čas práce úradníkov (nie ich čísla).

Listing programu (C++)

```
#include <cstdio>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

typedef long long ll;

int main() {
  int n, k; scanf("%d %d", &n, &k);
  vector<ll> dat(n);
  for (int i = 0; i < n; i++)
    scanf("%d", &dat[i]);
  sort(dat.begin(), dat.end());
  priority_queue<ll, vector<ll>, greater<ll> > halda;
  for (int i = 0; i < k; i++)
    halda.push(0);

  for (int i = n-1; i >= 0; i--) {
    ll x = halda.top(); halda.pop();
    halda.push(x + dat[i]);
  }
  ll mm = 0;
  while (!halda.empty()) {
    mm = halda.top(); halda.pop();
  }
  printf("%Ld\n", mm);
}
```

Podúloha b)

Môžeme napríklad nechať medzi dvoch úradníkov rozdeľovať práce $3, 3, 2, 2, 2$. Optimálne je jednému úradníkovi dať $3, 3$ a druhému $2, 2, 2$. Náš algoritmus ale dá jednému $3, 2, 2$ a druhému $3, 2$. Najdlhší čas práce je v optimálnom prípade 6, no v prípade nášho algoritmu 7.

Podúloha c)

Všimnime si, že pri danej podmienke dostane v optimálnom riešení každý úradník maximálne dve úlohy. Ak by dostal tri, tak jeho čas práce bude väčší ako t^*, čo je spor. Náš algoritmus potom robí (v najhoršom prípade, keď je úloh presne $2k$) nasledovnú vec: Spáruje najdlhšiu úlohu s najkratšou, druhú najdlhšiu s druhou najkratšou, ... Toto je intuitívne najlepšie možné riešenie. Poctivo sa to dá dokázať nasledovne:

Majme nejaké optimálne priradenie R_o a naše priradenie R. Ukážeme, že vieme R_o prerobiť na R a pritom najdlhšie pracujúci úradník bude pracovať rovnako dlho. Bez ujmy na všeobecnosti, nech $n = 2k$. Ak to tak nie je, doplníme si niekoľko úloh, ktoré trvajú 0 (toto síce odporuje podmienke, že sú dlhšie ako tretina optimálneho času, ale my chceme len dokázať, že priradenie najdlhšia s najkratšou, atď., je najlepšie).

Máme naše utriedené úlohy. Označme ich ako $a_1 \geq a_2 \geq \cdots \geq a_n$. Ak v R_o nejaký úradník dostal dve úlohy z hornej polovice, teda dve úlohy a_i, a_j, kde $i < j \leq n/2$, tak potom nejaký iný úradník dostal dve úlohy z dolnej polovice – a_k, a_l, kde $n/2 < k < l$. Určite $a_i + a_j \geq a_k + a_l$. A určite aj $a_i + a_l \leq a_i + a_j$ a $a_j + a_k \leq a_i + a_j$. Teda zámenou a_j a a_l sme naše riešenie určite nepokazili. Takýmito zámenami vieme riešenie R_o dostať do takého stavu, že každý úradník má práve jednu úlohu z $a_1, a_2, \ldots, a_{n/2}$.

Ďalej vieme úradníkov preusporiadať tak, že prvý dostane a_1, druhý a_2, atď.

Ešte stále sa môže riešenie R_o líšiť od nášho riešenia R. Naše riešenie R k úlohe a_1 pridá úlohu a_n, k úlohe a_2 úlohu a_{n-1}, atď.

Zoberme si prvé miesto, kde sa R_o a R líšia. Teda najmenšie i, kde k a_i nie je pridelená a_{n-i+1}, ale nejaká a_k. Nájdime si miesto, kde je pridelená a_{n-i+1}, bude to a_j. Určite platí $j > i$ (lebo po i sa R_o zhoduje s R). Z podobných dôvodov $k < a_{n-i+1}$. Teraz platí: $a_i + a_k \geq a_i + a_{n-i+1}$. A aj $a_i + a_k \geq a_j + a_k$. Opäť sme dosiahli iné riešenie, v ktorom sme nič nezhoršili, a je bližšie k R. A takto postupne upravíme R_o na tvar R a najdlhší čas práce úradníka zostane zachovaný.

Podúloha d)

Pozrime sa na úradníka, u ktorého sa pracuje najdlhšie. A pozrime sa na jeho poslednú pridelenú úlohu. Môžu nastať dva prípady. Ak je táto úloha dlhšia ako $t^*/3$, tak tvrdíme, že náš algoritmus dáva optimálne riešenie.

Stav po pridaní tejto úlohy je určite optimálny (vyplýva to z časti c). A úlohy, ktoré pridá náš algoritmus neskôr, už naše riešenie nepokazia (lebo ostatní úradníci pracujú kratšie).

Ak je táto úloha menšia alebo rovná $t^*/3$, tak sa pozrime, koľko práce mal úradník pred pridelením tejto úlohy (označme to ako p). Tvrdíme, že $p \leq (\sum_{i=1}^{n} a_i)/k \leq t^*$. Prvá nerovnosť plynie z toho, že náš úradník má pred priradením úlohy najmenej práce a nemôže sa stať, že všetci budú pracovať viac ako priemerne. Druhá nerovnosť plynie z toho, že priemerné rozdelenie práce je určite optimálne. Ale potom, keď dostane náš úradník svoju poslednú úlohu, tak to bude určite menej ako $4t^*/3$.

6. Obdarený nespratník

opravoval Bc. Bob
(max. 12 b za popis, 8 b za program)

Napriek tomu, že riešenie tejto úlohy vyžaduje rozobrať veľa špeciálnych prípadov, vaše programy si s nimi (zväčša) poradili. Možno sme nemali v zadaní zverejniť toľko vzorových výstupov, mohlo by to dopadnúť zaujímavejšie `;-)`

Čo sa vlastne deje so zloženými číslami, keď po nich Georg trieska svojím kladivkom? Štiepia sa. Vezmime si rozklad čísla $n > 1$ na prvočísla: $n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdots p_m^{\alpha_m}$. Po jeho rozbití vzniknú čísla $x = p_1^{\beta_1} \cdot p_2^{\beta_2} \cdots p_m^{\beta_m}$ a $y = p_1^{\gamma_1} \cdot p_2^{\gamma_2} \cdots p_m^{\gamma_m}$, pričom bude platiť $\alpha_i = \beta_i + \gamma_i$ pre všetky $1 \leq i \leq m$. Niektoré z hodnôt $\beta_1, \ldots, \beta_m$ môžu byť rovné 0, ale určite nie všetky – vtedy by sa totiž x rovnalo 1, čo nie je možné (podobne aj pre hodnoty $\gamma_1, \ldots, \gamma_m$).

Ekvivalentne: predstavme si, že každé prirodzené číslo väčšie ako 1 je kôpka guľôčok – prvočísel z jeho rozkladu. Niektoré guľôčky majú rovnakú farbu, lebo prislúchajú rovnakému prvočíslu. Rozbiť číslo kladivkom teda znamená rozdeliť kôpku na dve menšie neprázdne kôpky. Jasne vidíme, že prvočísla (kôpky obsahujúce jedinú guľôčku) sú nerozbitné.

Poďme teraz robiť Georgovi napriek a snažme sa rozdeľovať číslo n tak, aby sme sa vyhli číslu k. Ak k obsahuje z niektorého prvočísla viac kusov ako n, vôbec sa nemusíme namáhať: akokoľvek budeme deliť kôpky, nové guľôčky nám nepribudnú. Ďalej sa preto zaujímame len o opačný prípad, t.j. keď $k \mid n$.

Ak $k = n$, prehrali sme. Čo už, snáď budeme mať v ostatných prípadoch ($k < n$) väčšie šance.

Keď budú v kôpke čísla k aspoň dve rôzne prvočísla (označme ich p a q), môžeme hneď pri prvom rozdelení n vytvoriť čísla x a y také, že $p \nmid x$ a $q \nmid y$ (napríklad $x = q^\alpha$, $y = n/q^\alpha$, kde α je exponent prvočísla q v rozklade n). Keďže po takom kroku budú všetky výskyty prvočísla p oddelené od všetkých výskytov prvočísla q, v žiadnom prípade už k nevznikne.

Zostáva nám poriešiť prípad, keď je k deliteľné jediným prvočíslom (teda $k = p^\alpha$). Ak $\alpha = 1$, potom je k prvočíslom a my máme smolu. Pretože $p \mid n$, pri akomkoľvek rozbití n vznikne aspoň jedna kôpka, ktorá bude p obsahovať. Ani ďalším delením sa nám nepodarí zbaviť sa p. Keďže veľkosti kôpok sa postupne zmenšujú, nakoniec nám nutne zostane p ($= k$) osamostatnené.

Ešte sa pozrime na prípad $k = p^\alpha$ pre $\alpha \geq 2$. Zapíšme číslo n v tvare $p^\beta \cdot m$, kde $p \nmid m$; zrejme platí $\alpha \leq \beta$. Pri troche šťastia sa m nebude rovnať 1, potom môžeme z n postupne odoberať po jednom kuse p. Dostaneme tak čísla $p^{\beta-1} \cdot m$, $p^{\beta-2} \cdot m$, …, m, ktoré sú rôzne od k (vždy tam zavadzia m). Ďalej už môžeme m rozbíjať ľubovoľne.

Fajn, zostáva len možnosť $k = p^\alpha$, $n = p^\beta$ pre $\beta > \alpha \geq 2$. Ak sa však α rovná 2, nijako nezabránime vzniku k; hodnota p^2 sa totiž pri rozbíjaní nedá preskočiť. Kto tomu neverí, nech si to indukciou dokáže. A kto je navyše lenivý, nech si len skúsi rozbiť n na β prvočísel p a potom sa pozrieť na číslo, ktoré rozbíjal v poslednom kroku.

Naozaj posledný prípad: $k = p^\alpha$, $n = p^\beta$, $\beta > \alpha \geq 3$. Máme vyhraté: stačí postupne odoberať z n po jednom kuse p, až kým nedostaneme $p^{\alpha+1}$. Toto rozbijeme nejak inak, napríklad na p^2 a $p^{\alpha-1}$. Keďže $\alpha \geq 3$, podarilo sa nám pokaziť Georgovi radosť a vyhnúť sa číslu k.

Hurá, koniec rozoberaniu prípadov! Poďme si zhrnúť podmienky, ktoré musí naše riešenie overiť: Aby bola odpoveď `ano` (Georgovi sa vždy podarí získať k), musí platiť $k \mid n$ a aspoň jedna z možností:

- $k = n$,
- k je prvočíslo,
- $k = p^2$, $n = p^\beta$ a p je prvočíslo.

Overiť, či je k prvočíslo, vieme v čase $O(\sqrt{k})$ tak, že ho postupne vyskúšame vydeliť všetkými celými číslami od 2 do $\lfloor \sqrt{k} \rfloor$. Každé zložené číslo má totiž deliteľa (rôzneho od 1), ktorý je menší alebo rovný druhej odmocnine tohto čísla. (Dôkaz? Naozaj? No dobre: vezmime deliteľa d rôzneho od 1 aj k, ktorý je väčší ako $\sqrt{k}$, a pozrime sa na k/d.)

Overiť, či je číslo n nejakou mocninou prvočísla p, vieme v čase $O(\log n)$ napríklad tak, že budeme postupne deliť n prvočíslom p dovtedy, kým to pôjde bezo zvyšku. Ak nám nakoniec zostane 1, bola to mocnina.

Vzorové riešenie má teda v najhoršom prípade časovú zložitosť $O(\sqrt{k} + \log n)$. Pamäťová zložitosť je konštantná.

S trochou snahy by sme mohli naše riešenie urýchliť. Na testovanie prvočíselnosti totiž existujú algoritmy s polylogaritmickou časovou zložitosťou (vzhľadom na k), napríklad pravdepodobnostný Miller-Rabinov test.[2] Potom by sme sa však mali prestať tváriť, že aritmetické operácie s našimi celočíselnými premennými trvajú len konštantný čas.

Testovanie, či je n mocninou p, robíme tiež zbytočne pomaly. Hodnota exponentu β sa dá totiž binárne vyhľadať: najprv by sme postupne generovali čísla p^1, p^2, p^4, ..., p^{2^i}, kým by sme neprekročili n. Hodnotu β by sme potom binárne dohľadali v získanom intervale možných hodnôt. Keďže β je rádu $O(\log n)$, časová zložitosť by sa zlepšila na $O(\log \log n)$.

Tieto vylepšenia však neboli nutné na získanie plného počtu bodov.

Listing programu (C++)

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

bool solve(long long n, long long k) {
        if (n == k)
                return true;
        if (n % k != 0)
                return false;
        long long i;
        for (i = 2; i * i < k; ++i)
                if (k % i == 0)
                        return false;
        if (i * i == k) {
                while (n % i == 0)
                        n /= i;
                return n == 1;
        }
        return true;
}

int main() {
        long long n, k;
        cin >> n >> k;
        if (solve(n, k))
                cout << "ano" << endl;
        else
                cout << "nie" << endl;
}
```

opravoval Žaba

7. Opička, banány a ADHD

(max. 13 b za popis, 7 b za program)

Riešenia tejto úlohy sa delili do troch kategórií: riešenia v $O(n^2)$, riešenia v $O(n)$ a Jarovo riešenie. Všetkým gratulujem (aj keď Jaro je farmár), lebo zvládli napísať dobrý popis a zároveň úspešne naimplementovať svoje riešenie. A ako vidíte, občas je fajn odoslať aj pomalší program, ktorý však stojí oveľa menej námahy, lebo sa zaň dalo dostať až 11 bodov.

A teraz vzorák. Zjavne nemôžeme prechádzať všetky podpostupnosti, lebo ich je rádovo n^2. Poďme teda skúsiť niečo zlepšiť. Prvé pomocné pozorovanie je, že nemusíme riešiť minimum a maximum podpostupnosti naraz, sú od seba nezávislé a stačí vyrátať súčet maxím, súčet miním a na konci odčítať.

Zamerajme sa teraz na rátanie maxím a ako uvidíme, minimá sa budú dať rátať tak isto. Zamyslime sa nad tým, koľkokrát sa daný prvok na pozícii x s hodnotou $p(x)$ vyskytne ako maximum. Najskôr si všimnime, že ak na úseku $i \leq x \leq j$ je prvok x maximum, tak aj pre úsek i_1 až j_1, kde $i \leq i_1 \leq x \leq j_1 \leq j$, je prvok x maximum. Potrebujeme nájsť teda najdlhší úsek, na ktorom je $p(x)$ najväčšie. Potom všetky jeho podúseky obsahujúce x budú mať za maximum $p(x)$. Ak je ten najväčší úsek od i po j, tak $p(x)$ pridáme do súčtu maxím $(x - i + 1) \cdot (j - x + 1)$-krát – ľudsky povedané, vyberieme prvok naľavo od x, napravo od x, a pre tento úsek bude $p(x)$ maximum.

Kde sa nachádza tento najdlhší úsek i, j? Nech i je najmenší taký index naľavo od x, že medzi i a x nie je väčší prvok ako $p(x)$. Potom na mieste $i - 1$ je prvýkrát prvok väčší ako $p(x)$ alebo $i = 0$ (indexujem od 0), čo vyplýva z definície i. Podobne, na $(j + 1)$-vom mieste je prvýkrát prvok väčší ako $p(x)$. Takže sme si úlohu mierne zmenili. Ak by sme vedeli nájsť pre každý prvok prvý väčší prvok vpravo a vľavo, vedeli by sme pre každý prvok zistiť počet podúsekov, v ktorých je ten prvok maximum.

[2] http://en.wikipedia.org/wiki/Miller-Rabin_primality_test

Pozrime sa teraz ako čo najrýchlejšie pre každý prvok nájsť prvý prvok naľavo od neho, ktorý je väčší ako on. Zoberieme si jednoduchý zásobník a budeme prvky prechádzať v poradí zľava doprava. Na začiatok si do zásobníka dáme dvojicu -1 a ∞, čo značí, že na pozícii -1 je akože nekonečno. Vezmime si nasledujúci prvok s hodnotou $p(a)$ a pozrime sa na vrch zásobníka. Dúfame, že bude obsahovať informáciu o tom, kde je prvý väčší prvok naľavo. Nech je na vrchu zásobníka prvok s hodnotou $p(b)$. Ak $p(b) < p(a)$, môžeme si dovoliť prvok b vyhodiť zo zásobníka, lebo už nikdy nebude hranicou pre iný prvok. Pre prvky $c > a$ a $p(c) < p(b)$ to bude najviac prvok a, lebo $b < a$. A pre prvky väčšie ako $p(b)$ to nemôže byť prvý väčší prvok. Takto postupne vyhadzujeme veci zo zásobníka, až kým nenarazíme na prvok väčší ako $p(a)$. Index tohoto prvku je index prvého väčšieho prvku naľavo od a. Následne hodnotu a a $p(a)$ vložíme do zásobníka.

Je snáď jasné, že rovnaký prístup vieme použiť aj na zistenie prvého väčšieho prvku napravo (stačí prechádzať postupnosť v opačnom smere) a minimá zistíme obrátením znakov nerovnosti. Ešte je treba poznamenať, že nám trochu robia problém rovnaké prvky, ale to ošetríme tak, že napravo budeme hľadať prvky väčšie/menšie alebo rovné a naľavo prvky ostro väčšie/menšie. Pomocou tohto už vieme pre každý prvok povedať koľkokrát bude maximom a minimom a tieto hodnoty len sčítame.

Celá časová zložitosť bude nasledovná. Zistenie hraníc nám trvá $O(n)$, lebo každý prvok raz do zásobníka vložíme a raz vyberieme. Takisto porátať výsledok je len jeden prechod predspracovaným poľom, teda dokopy to bude stále $O(n)$. No a pamätať si potrebujeme danú postupnosť, zásobník a vypočítané hranice, teda $O(n)$.

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>
#include <stack>
using namespace std;

typedef long long ll;

int main()
{
  int n;
  scanf("%d ",&n);
  vector<ll> A; A.resize(n);
  for(int i=0; i<n; i++) scanf("%lld ",&(A[i]));
  vector<pair<ll,ll> > M1; M1.resize(n);
  vector<pair<ll,ll> > M2; M2.resize(n);
  stack<pair<ll,ll> > Maxi;
  stack<pair<ll,ll> > Mini;
  Maxi.push(make_pair(1000047,-1));
  for(int i=0; i<n; i++)
  {
  while(Maxi.top().first<=A[i]) Maxi.pop();
    M1[i].first=Maxi.top().second;
    Maxi.push(make_pair(A[i],i));
  }
  while(!Maxi.empty()) Maxi.pop();
  Maxi.push(make_pair(1000047,n));
  for(int i=n-1; i>=0; i--)
  {
    while(Maxi.top().first<A[i]) Maxi.pop();
    M1[i].second=Maxi.top().second;
    Maxi.push(make_pair(A[i],i));
  }
  Mini.push(make_pair(-1,-1));
  for(int i=0; i<n; i++)
  {
    while(Mini.top().first>=A[i]) Mini.pop();
    M2[i].first=Mini.top().second;
    Mini.push(make_pair(A[i],i));
  }
  while(!Mini.empty()) Mini.pop();
  Mini.push(make_pair(-1,n));
  for(int i=n-1; i>=0; i--)
  {
    while(Mini.top().first>A[i]) Mini.pop();
    M2[i].second=Mini.top().second;
    Mini.push(make_pair(A[i],i));
  }
  ll vys=0ll;
  for(int i=0; i<n; i++)
  {
    vys+=(ll)A[i]*(ll)(i-M1[i].first)*(ll)(M1[i].second-i);
    vys-=(ll)A[i]*(ll)(i-M2[i].first)*(ll)(M2[i].second-i);
  }
  printf("%lld\n",vys);
return 0;
}
```

8. Oblej to!

opravoval MišoF
(max. 2 b za popis, 20 b za program)

Ako pristupovať k takýmto úlohám? V prvom rade netreba podceniť fázu spoznávaciu: nainštalujem hru na

mobil, prípadne nájdem nejaký applet, a klikám až kým ma prsty nebolia :). Okrem toho, že je to lepšia zábava ako učiť sa na maturity, to aj skutočne pomáha – zistíte, čo zhruba funguje a čo nie. Zase som prehral hru, lebo mi v niektorom rohu ostala guča, ktorú som už nestihol „rozpustiť“? Asi neuškodí dať si na to nabudúce pozor.

No a keď už my hru hrať ako-tak vieme (ja mám úspešnosť niekde okolo 70%, zväčša som lenivý príliš myslieť), môžeme ju skúsiť naučiť hrať aj počítač.

Začiatok je u všetkých dobrých riešení rovnaký – treba si v programe vedieť reprezentovať rôzne stavy hracieho plánu a napísať si funkciu, ktorá pomocou prehľadávania zrealizuje ťah. Táto funkcia dostane na vstupe starý plán a zvolenú farbu, na výstupe vráti nový plán. (V programe na reprezentáciu stavu používame pomerne novú fičúriu C++: `array`, teda pole pevnej veľkosti. V princípe si hrací plán rozmerov 22×22 pamätáme ako dvojrozmerné pole rozmerov 24×24, ktoré má navyše okraj tvorený nulami. Len teda nejde o statické dvojrozmerné pole, ale objekt, s ktorým sa nám potom pohodlnejšie pracuje.)

No a keď už toto všetko máme, môžeme sa začať zamýšľať nad stratégiou. Prezrieť všetky možné priebehy hry zrovna nepôjde, 6^{36} je predsa len dosť veľa. Zato 6^6 nie je ani 50 tisíc. Takže je len rozumne málo možností, čo vlastne v nasledujúcich napr. šiestich ťahoch spravíme. Mohli by sme ich prezrieť všetky a vybrať tú najlepšiu.

No hej, ale ktorá je tá najlepšia? V kolektíve riešiteľov najobľúbenejší názor (Kuko, Basha, Xellos, Viki, Martin, Vlejd) bol „tá, kde má hlavná oblasť najväčšiu plochu“. Toto vôbec nie je zlá stratégia. Väčšina riešení s týmto prístupom sa pohybovala medzi 50% a 70% úspešnosťou. Najlepšie riešenie z tejto kategórie odovzdal Askar, ktorý pooptimalizoval kód, a tak stihol pozerať na viac ťahov dopredu ako ostatní a vyžmýkal úspešnosť tesne cez 80%.

Okolo 50% sa dalo dosiahnuť aj stratégiou „najlepší hrací plán je ten, kde má hracia plocha najväčší obvod“, ako sa to podarilo Máriovi.

Najlepšie riešenia, teda Baklažánovo a moje, boli založené na rôznych ale podobných kritériách. Obe súvisia nie s plochami, ale so vzdialenosťami. Ak sa totiž pozrieme na riešenia maximalizujúce plochu, zistíme ľahko ich slabinu: prehrajú väčšinou tak, že v niektorom rohu ostane pár vrstiev nesprávne ofarbených štvorčekov. Dobrá situácia totiž nevyzerá nutne tak, že mám veľkú jednofarebnú plochu. Dôležitejšie je to, aby nikam nebolo príliš ďaleko.

To isté inými slovami: Veľmi dobrá situácia je taká, keď každá oblasť okrem hlavnej priamo s hlavnou oblasťou susedí. Z takejto situácie vieme veľmi rýchlo vyhrať – stačí práve raz zvoliť každú z farieb ešte prítomných oblastí. O niečo horšia situácia je taká, v ktorej máme aj oblasti s hlavnou nesusediace. Tam už potom záleží na poradí, v akom volíme farby – musíme sa k tým vzdialenejším oblastiam postupne „prehrýzť“.

Ja som si teda zvolil nasledovné kritérium: pre každú oblasť na hracom pláne si spočítame, koľko najmenej ťahov treba na to, aby som *túto konkrétnu oblasť* pripojil k hlavnej. (Toto sa dá spraviť jedným prehľadávaním do šírky z ľavého horného rohu. Presun na susedné políčko ma stojí 0, ak je rovnakej farby ako moje, resp. 1, ak je odlišnej.) No a hrací plán je tým lepší, čím je *maximum* týchto hodnôt menšie. (Resp. hrací plán je zlý vtedy, keď je nejaká oblasť ďaleko od hlavnej.)

No a k tomuto už pribalíme pozeranie na dostatočne veľa ťahov dopredu a máme to. (Program z listingu sa zväčša pozerá 3 ťahy dopredu, ale aj 4 by bez problémov stíhal v časovom limite.) Záverečná poznámka: Je *ľahšie* vyriešiť 1000 levelov za 1000 sekúnd ako 20 za 20 sekúnd. Prečo? Lebo veľa levelov vyriešime rýchlo a potom nám ostáva *do bludu veľa* času na tých pár ťažkých. A tam potom môžeme napr. skúsiť pozerať až na 5 či 6 ťahov dopredu.

Listing programu (C++)

```
#include <algorithm>
#include <iostream>
#include <sstream>
#include <cassert>
#include <vector>
#include <array>
#include <queue>
using namespace std;

#define SIZE 22

typedef array< array<int,SIZE+2>, SIZE+2> Board;
int dr[]={-1,1,0,0}, dc[]={0,0,-1,1};

// takto mozeme naucit ostream aby nam vypisal board, napr. prikazom cout << B;
ostream& operator<<(ostream &out, const Board &B) {
    for (auto row:B) { for (auto c:row) out << c; out << endl; }
    return out;
}

// distance_board vyrobi Board plny "nekonecien"
Board distance_board() {
    Board output;
    for (int r=0; r<=SIZE+1; ++r) for (int c=0; c<=SIZE+1; ++c)
        output[r][c]  =  (r==0 || r==SIZE+1 || c==0 || c==SIZE+1) ? -1 : 987654321;
    return output;
```

```
}

// read_board nacita Board zo standardneho vstupu
Board read_board() {
    Board output;
    for (int r=0; r<=SIZE+1; ++r) for (int c=0; c<=SIZE+1; ++c) output[r][c] = 0;
    for (int r=1; r<=SIZE; ++r) {
        string line; cin >> line;
        for (int c=1; c<=SIZE; ++c) output[r][c] = line[c-1]-'0';
    }
    return output;
}

// bfs prehladavanim do sirky vyfarbi komponent zacinajuci na (1,1) farbou newc
// navratova hodnota je pocet zaplavenych policok
int bfs(Board &B, int newc) {
    int answer=1, oldc=B[1][1];
    assert(newc != oldc);
    queue<int> Q;
    B[1][1]=newc;
    Q.push(1); Q.push(1);
    while (!Q.empty()) {
        int r=Q.front(); Q.pop();
        int c=Q.front(); Q.pop();
        for (int d=0; d<4; ++d) {
            int nr=r+dr[d], nc=c+dc[d];
            if (B[nr][nc]!=oldc) continue;
            B[nr][nc]=newc;
            ++answer;
            Q.push(nr); Q.push(nc);
        }
    }
    return answer;
}

// farthest_cell_distance najde prehladavanim do sirky cislo K s nasledujucou vlastnostou:
// pre kazde policko existuje nejaka postupnost K tahov, ktora toto policko dostane do hlavnej oblasti
// (pozor, toto este neznamena, ze na K tahov vieme vyhrat -- rozne policka mozu mat rozne postupnosti)
pair<int,int> farthest_cell_distance(const Board &B) {
    Board dist = distance_board();
    deque< pair<int,int> > Q;
    dist[1][1] = 0;
    Q.push_front( make_pair(1,1) );
    while (!Q.empty()) {
        pair<int,int> tmp = Q.front(); Q.pop_front();
        int kr=tmp.first, kc=tmp.second;
        for (int d=0; d<4; ++d) {
            int nr=kr+dr[d], nc=kc+dc[d];
            int add = B[kr][kc]==B[nr][nc] ? 0 : 1;
            if (dist[nr][nc] > dist[kr][kc]+add) {
                dist[nr][nc] = dist[kr][kc]+add;
                if (add) Q.push_back(make_pair(nr,nc)); else Q.push_front(make_pair(nr,nc));
            }
        }
    }
    int maxv=-1, maxc=0;
    for (int r=1; r<=SIZE; ++r) for (int c=1; c<=SIZE; ++c) {
        if (dist[r][c]>maxv) { maxv=dist[r][c]; maxc=0; }
        if (dist[r][c]==maxv) ++maxc;
    }
    return {maxv,maxc};
}

// pomocna funkcia: uz sme vyhrali?
bool is_solved(const Board &B) {
    for (int r=1; r<=SIZE; ++r) for (int c=1; c<=SIZE; ++c) if (B[r][c]!=B[1][1]) return false;
    return true;
}

// pomocna funkcia: mame este takuto farbu?
bool contains_color(const Board &B, int col) {
    for (int r=1; r<=SIZE; ++r) for (int c=1; c<=SIZE; ++c) if (B[r][c]==col) return true;
    return false;
}

// pomocna funkcia: vieme uz vsetky policka danej farby vyzrat jednym tahom?
bool can_eliminate_color(const Board &B, int col) {
    if (!contains_color(B,col)) return false;
    if (B[1][1]==col) return false;
    Board copy = B;
    bfs(copy,col);
    bfs(copy,7);
    return !contains_color(copy,col);
}

// pazravo vyzieraj farby, ktore uz vies eliminovat
// navratova hodnota je retazec spravenych tahov
string greedy_steps(const Board &B, bool &ok) {
    ok = true;
    if (is_solved(B)) return "";
    for (int c=1; c<=6; ++c) if (can_eliminate_color(B,c)) {
        Board copy=B;
        bfs(copy,c);
        return char('0'+c) + greedy_steps(copy,ok);
    }
    ok = false;
    return "";
}
```

```
string solve(Board &B) {
    int steps = 0;
    string answer = "";
    while (true) {
        if (is_solved(B)) return answer;
        int col=0;

        // ak sa da vyzrat zvysok nejakej farby, vyzer ju
        for (int c=1; c<=6; ++c) if (can_eliminate_color(B,c)) { col=c; break; }
        if (col) { ++steps; answer+=char('0'+col); bfs(B,col); continue; }

        // ak uz bolo 25 tahov, vyskusame vsetky mozne 5-tahove postupnosti
        // ak niektora z nich vedie do pazravo vyhratelnej situacie, vyhrali sme
        if (steps == 25) {
            for (int c1=1; c1<=6; ++c1) if (B[1][1]!=c1) {
            for (int c2=1; c2<=6; ++c2) if (c1!=c2) {
            for (int c3=1; c3<=6; ++c3) if (c2!=c3) {
            for (int c4=1; c4<=6; ++c4) if (c3!=c4) {
            for (int c5=1; c5<=6; ++c5) if (c4!=c5) {
                Board C = B;
                bfs(C,c1); bfs(C,c2); bfs(C,c3); bfs(C,c4); bfs(C,c5);
                bool ok;
                string g = greedy_steps(C,ok);
                if (ok) {
                    stringstream ss; ss << c1 << c2 << c3 << c4 << c5;
                    return answer + ss.str() + g;
                }
            }}}}}
        }

        // inak pozerame 3 tahy dopredu a vyberame si tah, ktory minimalizuje
        // hodnotu vratenu funkciou farthest_cell_distance
        int bestd=100, bestc=0, bestmove=-1;
        for (int c1=1; c1<=6; ++c1) if (B[1][1]!=c1) {
        for (int c2=1; c2<=6; ++c2) if (c1!=c2) {
        for (int c3=1; c3<=6; ++c3) if (c2!=c3) {
            Board C = B;
            bfs(C,c1);
            bfs(C,c2);
            bfs(C,c3);
            pair<int,int> tmp = farthest_cell_distance(C);
            if (tmp.first<bestd || (tmp.first==bestd && tmp.second<bestc)) {
                bestd = tmp.first;
                bestc = tmp.second;
                bestmove = c1;
            }
        }}}
        ++steps; answer+=char('0'+bestmove); bfs(B,bestmove);
    }
}

// pomocna funkcia: uprav riesenie na vypis
string fix(string in) {
    if (in.size()>36u) in=in.substr(0,36);
    if (in.size()<36u) in+=string(36-in.size(),'1');
    return in;
}

int main() {
    int H; cin >> H;
    while (H--) { Board B = read_board(); cout << fix(solve(B)) << endl; }
}
```

opravoval Usamec

1::9. Trololo

(max. 15 b za popis, 10 b za program)

Najprv si trochu preformulujme úlohu. Miesto toho, aby sme hľadali jednu cestu, ktorá ide najprv zo západu na východ a potom z východu na západ, budeme hľadať dve disjunktné cesty zo západu na východ (jednu z nich potom otočíme a máme našu požadovanú okružnú cestu).

Celú našu úlohu vyriešime dynamickým programovaním. Budeme si počítať hodnotu $A[i][j]$, pre všetky $i \leq j$, ktorá nám hovorí, koľko najviac miest vieme navštíviť, ak z mesta 1 urobíme dve disjunktné cesty zo západu na východ, ktoré končia v mestách i a j.

Pokiaľ je $i = j$ a zároveň nie sú rovné 1, resp. n, tak $A[i][i] = -\infty$. Špeciálne $A[1][1] = 0$.

Teraz si ukážeme, ako spočítať bežnú hodnotu $A[i][j]$. Do mesta j vedú zo západu lety z miest $a_1, a_2, a_3, ..., a_k$. A práve jedným z týchto letov tu musíme prísť. Takže stačí zobrať maximum z hodnôt $A[i][a_x]$, resp. $A[a_x][i]$ pre všetky možné x, pričítať k nemu jednotku a máme hodnotu $A[i][j]$. Presne rovnakým postupom dopočítame aj hodnotu $A[n][n]$.

Seriózne riešenie ešte ukáže, že táto dynamika fakt nenájde horšiu cestu ako optimálnu a každá cesta, čo nájde, bude spĺňať podmienky zo zadania. Tieto ľahké dôkazy nechávame na pozorného čitateľa.

Zadanie ešte požadovalo vypísať jednu najdlhšiu cestu. To už skúseného riešiteľa určite nezaskočí, stačí si v dynamike pamätať, odkiaľ sme sa na ktoré miesto dostali.

Celková časová zložitosť riešenia je $O(nm)$ (každým letom v dynamike prejdeme n-krát), pamäťová $O(n^2)$.

Listing programu (C++)

```
#include <cstdio>

#include <cstdio>
#include <algorithm>
#include <vector>
#include <deque>

using namespace std;

int N, M;
vector<vector<int> > prev;

#define INF 123456789

int main() {
  scanf("%d %d", &N, &M);
  prev.resize(N);
  for (int i = 0; i < M; i++) {
    int a, b; scanf("%d %d", &a, &b);
    a--; b--;
    int mi = min(a,b), ma = max(a,b);
    prev[ma].push_back(mi);
  }

  vector<vector<pair<int, int> > > dyna(
      N, vector<pair<int, int> >(N, make_pair(-INF, -1)));

  dyna[0][0] = make_pair(0, -1);
  for (int i = 0; i < N; i++) {
    for (int j = min(N-1, i+1); j < N; j++) {
      for (int k = 0; k < prev[j].size(); k++) {
        if (prev[j][k] == i && i != 0) continue;
        int mi = min(i, prev[j][k]);
        int ma = max(i, prev[j][k]);
        if (dyna[i][j].first < dyna[mi][ma].first + 1) {
          dyna[i][j].first = dyna[mi][ma].first + 1;
          dyna[i][j].second = prev[j][k];
        }
      }
    }
  }
  if (dyna[N-1][N-1].first < 0) {
    printf("TROLOLOLOLO\n");
  } else {
    deque<int> path;
    path.push_back(N-1);
    int mi = N-1;
    int ma = N-1;
    while (mi != 0 || ma != 0) {
      int k = dyna[mi][ma].second;
      if (path.back() == ma) path.push_back(k);
      else path.push_front(k);
      int nmi = min(mi, k);
      int nma = max(mi, k);
      mi = nmi;
      ma = nma;
    }
    for (int i = 0; i < path.size(); i++)
      printf("%d%c", path[i] + 1, i == path.size() - 1 ? '\n' : ' ');
  }
}
```

opravoval Usamec

1::10. Tajný kód

(max. 10 b za popis, 15 b za program)

Veľmi dôležitou pomôckou v celom vzorovom riešení bude nasledovné tvrdenie: Počet prvočísel v rozsahu od 1 do n je približne $n/\ln n$. Dôkaz tohoto tvrdenia je celkom netriviálny a nebudeme sa mu venovať.

Prvým dobrým pozorovaním je to, že výsledok sa nám zmestí do 64-bitovej premennej (dokonca aj so znamienkom). Dôkaz prenechávame na čitateľa (stačí použiť vyššie spomínané tvrdenie).

Celkom jasne teda platí, že $q < 63$. Na tom sa dá už postaviť celkom dobré riešenie s nasledovnou myšlienkou: Pre všetky možné prvočíselné q nájdeme prvých k prvočísel p takých, že p^q je väčšie ako n a zároveň p^q je menšie ako 2^{63}. Z týchto vygenerovaných čísel p^q vyberieme k-te najmenšie.

Ako by takéto riešenie dopadlo? Pre $q = 3$ potrebujeme prvočísla do veľkosti maximálne 2 100 000. Pre väčšie q táto hranica prudko klesá nadol. Takže situáciu pre $q \geq 3$ odbavíme tým, že vygenerujeme pomocou Eratostenovho sita prvočísla do 2 100 000 a tie využijeme. Dá sa povedať, že túto časť vieme odbaviť v čase $O(n^{1/3} \log\log n + k \log k)$ (urobíme Eratostenovo sito a potom utriedime rádovo $O(k)$ čísel).

Problém je s $q = 2$. Potrebujeme prvočísla približne veľkosti 10^9. Eratostenovo sito nám tak vysoké prvočísla vygenerovať nestihne. Taktiež triviálne riešenie, ktoré ide postupne po číslach a testuje prvočíselnosť, je pomerne pomalé.

Trik vzorového riešenia je nasledovný. Pomocou vety na začiatku vzoráku odhadneme potrebný rozsah, kde hľadať prvočísla (dobrý odhad je napríklad od $\sqrt{n}$ do $\sqrt{n} + 25k$). Teraz na tomto rozsahu budeme vyškrtávať

zložené čísla podobne ako v Eratostenovom site. Každé zložené číslo z tohoto rozsahu je deliteľné nejakým číslom z rozsahu od 2 do $\approx n^{1/4} + k$. Takže prejdeme všetkými týmito číslami a pomocou nich vyškrtneme všetky potrebné zložené čísla. Zložitosť tejto operácie bude $O((n^{1/4} + k \log k) \log \log n)$.

Tým je náš vzorák hotový. V kóde je pre jednoduchosť navyše veľa vecí natvrdo zadrôtovaných.

Listing programu (C++)

```
#include <cstdio>
#include <set>
#include <vector>
#include <cmath>
using namespace std;

/* MAXS^3 > 2^63 */
#define MAXS 2100000

long long N, K;
set<long long> R;
bool neniprv[MAXS];
vector<long long> prv;

int triv[] = {
  3, 2097151,
  5, 6208,
  7, 511,
  11, 52,
  13, 28,
  17, 13,
  19, 9,
  23, 6,
  29, 4,
  31, 4,
  37, 3,
  41, 2,
  43, 2,
  47, 2,
  53, 2,
  59, 2,
  61, 2,
  0, 0
};

long long pomalypow(long long p, long long q) {
  long long r = 1;
  while(q--) r *= p;
  return r;
}

bool jeprv(long long x) {
  if (x < MAXS) return !neniprv[x];
  int i = 0;
  for (i = 0; prv[i] * prv[i] <= x; i++)
    if (x % prv[i] == 0) return false;
  //fprintf(stderr, "pre %lld sme zistovali %d\n", x, i);
  return true;
}

int main() {
  scanf("%lld %lld", &N, &K);

  // sito
  neniprv[0] = 1;
  neniprv[1] = 1;
  for (int i = 2; i*i < MAXS; i++)
    if (!neniprv[i])
      for(int j = i*i; j < MAXS; j += i)
        neniprv[j] = 1;
  for (int i = 0; i < MAXS; i++)
    if (!neniprv[i]) prv.push_back(i);
  //fprintf(stderr, "dositovali sme s %d\n", prv.size());

  // q>2
  for (int i = 0; triv[i]; i += 2) {
    int q = triv[i], maxp = triv[i+1];
    for (unsigned i = 0; prv[i] <= maxp; i++)
      R.insert(pomalypow(prv[i], q));
  }
  //fprintf(stderr, "R je velke %d\n", R.size());

  // q=2 kurzor
  long long kurzor = pow(N, 0.5);
  kurzor -= 20;
  if (kurzor < 0) kurzor = 0;
  while (kurzor*kurzor <= N) kurzor++;
  //fprintf(stderr, "mame odmocninu\n");

  // q=2 sito2
  vector<bool> sito2(K*25, true);   // 25 je radovo ln(sqrt(2^63))
  long long spodok = kurzor, vrch = kurzor + sito2.size();
  for (unsigned i = 0; i < prv.size(); i++) {
    long long p = prv[i];
    long long z = (spodok / p) * p;
    while (z < spodok || z == p) z += p;
    z -= spodok;
```

```
    for ( ; z < sito2.size(); z += p) sito2[z] = false;
  }
  //fprintf(stderr, "mame sito2 spodok=%lld vrch=%lld\n", spodok, vrch);

  // q=2 cyklus
  int nasli = 0;
  for ( ; nasli < K+1; kurzor++) {
    if (kurzor >= vrch) {
      // pomaly test lebo sme vyliezli z predpokladanej oblasti (velkost sito2 je len kvalifikovany odhad)
      if (!jeprv(kurzor)) continue;
    }
    else {
      // v sito2 uz je plna informacia o vsetkych prvocislach co nas zaujimaju
      if (sito2[kurzor-spodok] == false) continue;
    }
    R.insert(kurzor*kurzor);
    nasli++;
  }
  //fprintf(stderr, "presli sme %lld cize od %lld do %lld\n", kurzor - spodok, spodok, kurzor);

  // vysledok
  //fprintf(stderr, "finalne R je %d\n", R.size());
  typeof(R.begin()) it = R.upper_bound(N);
  for (int i = 0; i < K-1; i++) ++it;
  printf("%lld\n", *it);

  return 0;
}
```

2::9. Thajsko a chobotník

opravoval Jano
(max. 15 b za popis, 10 b za program)

Jedno z mnohých ponaučení tejto úlohy je, že kategórie T sa netreba báť, aj v nej sa občas vyskytnú jednoduchšie úlohy. Svedčí o tom aj dosť veľké množstvo plných počtov.

Keď si nevieme rady s ťažkým problémom, môže sa oplatiť rozbiť problém na viacero menších, ktoré nejako súvisia. Ideálne by bolo, keby sme pomocou nejakých čiastkových informácií vedeli ľahko vyskladať ďalšie a ďalšie, až kým sa nedopracujeme ku konečnému výsledku. Takéto niečo sa zvykne nazývať dynamické programovanie.

Náš ťažký problém je zistiť, koľko Zemčo dokáže najesť pri ceste z $[a, b]$ do $[0, 0]$. Malý čiastkový problém bude, koľko toho dokáže Zemčo zjesť cestou z políčka $[x, y]$ do $[0, 0]$ pre $0 \leq x \leq a$ a $0 \leq y \leq b$.

Pomalšie riešenie

Najskôr sa skúsme zamyslieť, čo by sa stalo, keby rozmery zálivu boli do 1000×1000 a my by sme si mohli dovoliť spracovať každé políčko mapy zvlášť. Takýto prístup má dve veľké výhody.

- Hodnoty veľkého množstva políčkok už poznáme, napríklad cestou z políčok $[-1, y]$ a $[x, -1]$ nedokáže zjesť Zemčo nič, lebo z týchto políčok sa nevie dostať domov. (Zemčo vie svoje súradnice iba zmenšovať.) Hodnotu na políčku $[0, 0]$ tiež poznáme, lebo cestou z políčka $[0, 0]$ Zemčo dokáže zjesť iba to, čo je na tomto políčku.
- Zvyšné hodnoty si ľahko dopočítame z predchádzajúcich. Hodnoty počítame v rozumnom poradí, napríklad prvé počítame tie, ktoré majú najmenšiu x-ovú súradnicu a v prípade rovnosti vyberieme tie s menšou y-ovou súradnicou. Hodnota na políčku $[x, y]$ bude maximum z $[x-1, y]$ $[x, y-1]$ plus počet rýb na políčku $[x, y]$. Totiž Zemčo sa z tohto políčka vie pohnúť len dvoma spôsobmi, dole alebo doľava. Najlepšie čo teda môže spraviť je, zjesť ryby na svojom políčku a pohnúť sa na výhodnešie políčko z oboch susedných. Výhodnejšie je to, z ktorého sa mu cestou domov dožičí viac jedla, a my už túto hodnotu máme vypočítanú, pretože ich rátame postupne od najmenších súradníc po najväčšie.

Postupne teda prejdeme každé políčko z mriežky $a \times b$ a pre každé si v konštantnom čase pozrieme jedno pod ním a jedno vľavo a spočítame si optimálny zisk na ceste z políčka $[a, b]$ do $[0, 0]$. Takéto riešenie má časovú zložitosť $O(ab)$.

Pokiaľ však záliv bude veľký a rozmery budú až do $10^9 \times 10^9$, nemôžeme si dovoliť prejsť všetky políčka zálivu.

Vzorové riešenie

(Pozor, nepomýľte si pri čítaní pojmy veľkosť húfu = počet rýb na konkrétnom políčku a hodnota políčka = koľko rýb dokážeme zjesť cestou z políčka do bodu $[0, 0]$.)

Hoci si nemôžeme dovoliť prejsť všetky políčka zálivu, prejsť všetky políčka s húfmi rýb môžeme, veď tie voľné nás veľmi nezaujímajú. Navyše môžeme aplikovať úplne ten istý princíp. Pre každé políčko si zistíme, kam

sa nám najviac oplatí pohnúť. Teraz sa nemôžeme zaoberať pohybom doľava a dole o jedno políčko, pretože voľné políčka rátať nechceme. Do úvahy (kam sa môžeme pohnúť) pripadajú všetky políčka s húfmi, ktoré môžeme prejsť cestou z $[x, y]$ do $[0, 0]$, takže všetky tie, ktorých súradnice sú aspoň $[0, 0]$ a najviac $[x, y]$.

Ostáva nám zdolať jeden problém, ako čo najrýchlejšie zistiť, ktoré políčko z tých vo štvorci od $[0, 0]$ po $[x, y]$ je najvýhodnejšie, teda má najvyššiu hodnotu.

Predstavme si, že máme pole čísel a chceme s ním robiť dve operácie. Prvá operácia nastaví číslo v poli na ľubovoľnú hodnotu. Druhá povie maximum z čísel na políčkach $0, \ldots, p$.

Dátová štruktúra, ktorá robí obe tieto operácie v čase $O(\log b)$ (b je veľkosť poľa, v našom prípade výška zálivu) sa nazýva maximový intervalový strom a funguje na pomerne jednoduchom princípe. O ňom si povieme neskôr.

Predstavme si, že už vieme vykonávať tieto dve operácie. Môžeme si zoradiť rybie húfy podľa x-ovej súradnice a postupne ich spracovávať od najväčšej po najmenšiu. (V prípade rovnosti x-ových súradnic uprednostníme spodnejší húf.) Robíme to preto, aby tie húfy, ktoré sú od aktuálneho naľavo sú práve tie, ktoré sme už spracovali. Potom nám stačí rozlišovať, ktoré húfy sú pod nami a ktoré nie. A dokonca ich ani netreba rozlišovať. My len potrebujeme nájsť najväčšiu z hodnôt, ktoré sú pod nami (alebo zarovno nás). A presne toto robí maximový intervalový strom.

Vždy, keď spracujeme nejaké políčko na pozícii $[x, y]$, zapíšeme si jeho hodnotu na y-té políčko v poli v čase $O(\log b)$ pomocou intervalového stromu. Keď chceme zistiť, koľko najviac vieme získať, zistíme si maximum v poli od pozície 0 po y tiež v čase $O(\log b)$. Dôležité je to, že v poli na pozíciách 0 až y sú tie hodnoty, do ktorých sa vieme dostať, pretože sú pod pod nami a naľavo od nás (lebo sú už spracované).

Ešte si môžeme všimnúť, že tam nie sú úplne všetky hodnoty, môže sa nám stať, že si nejaké políčko prepíšeme vyššou hodnotou, ale to nám pri hľadaní maxima nevadí. K takto zistenej maximálnej hodnote pripočítame veľkosť húfu na políčku a zapíšeme.

Takto teda postupne prejdeme všetky húfy a v čase $O(n \log b)$ zistíme aj hodnotu políčka $[a, b]$, odkiaľ Zemčo vyráža. (Buď si pridáme prázdny húf na túto pozíciu, alebo na konci jednoducho pozrieme maximum v poli na pozíciách $0, \ldots, b$.)

Ostáva nám vyriešiť jeden drobný kozmetický detail, ktorým sú stále veľké rozmery zálivu. Totiž húfy sú na pozíciách $0, \ldots, 10^9$, ale pole s miliardou prvkov si nemôžeme dovoliť. Keďže z týchto 10^9 políčok je obsadených najviac n, môžeme vykonať tvz. kompresiu súradníc – t.j. vynechať všetky voľné riadky, aby stačilo pole veľkosti n. Táto kompresia sa dá spraviť napríklad tak, že si utriedime húfy podľa y-ovej súradnice, a prepíšeme im súradnice na napríklad čísla od 1 po n. Vzhľadom na to, že tým nezmeníme vzájomné poradie, tak nezmeníme ani výsledok, takže to pokojne môžeme spraviť.

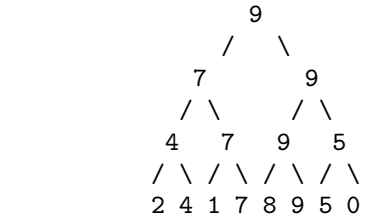
Vďaka tomu sa zmenší veľkosť potrebnej pamäte z $O(b)$ na $O(n)$ a časová zložitosť operácií s poľom klesne na $O(\log n)$.

Časová zložitosť všetkého dokopy bude $O(n \log n)$ a pamäťová $O(n)$.

Maximový intervalový strom

Ak vás zaujíma, čo je to ten intervalový strom, tak stručný popis nájdete tu.

Maximový intervalový strom sa skladá z niekoľkých vrstiev vrcholov v strome, pričom spodná vrstva je naše pole (ktoré si doplníme na konci nulami, aby malo dĺžku 2^i pre nejaké i). Každá ďalšia vrstva má $2\times$ menej prvkov ako tá pod ňou. Pričom hodnoty vo vrcholoch v ďalších vrstvách sa vypočítajú ako maximum z ich synov (dvoch vrcholov pod nimi).



Keď si vrstvy zapíšeme do poľa zaradom od prvého políčka (najskôr prvky prvej vrstvy, potom prvky druhej vrstvy, atď., až nakoniec prvky samotného poľa, ktoré začínajú na pozícii 2^i), tak bude navyše platiť, že otec vrcholu na pozícii x má pozíciu $x/2$ a synovia vrcholu s pozíciou y majú pozície v poli $2y$ a $2y + 1$. V našom prípade bude toto pole vyzerať (aj s 0. prvkom) $[0, 9, 7, 9, 4, 7, 9, 5, 2, 4, 1, 7, 8, 9, 5, 0]$

Nastavenie čísla v poli – operácia **set** – so sebou nesie aj aktualizovanie hodnôt vyššie v strome.

```
set(prvok, hodnota){ // nastavi pole['prvok'] na cislo 'hodnota'
   pole[2^i+prvok] = hodnota;
```

```
   update((2^i+prvok)/2);
}

update(vrchol){ // aktualizuje vrchol
    pole[vrchol] = max(pole[2*vrchol], pole[2*vrchol+1]);
    update(vrchol/2);
}
```

Keďže vrstiev je i, čo je $O(\log n)$, táto operácia zaberie logaritmický čas.

Podobne vieme vykonať aj druhú operáciu. Pre prehľadnosť si definujme interval pokrytý vrcholom V ako všetky prvky poľa, ktoré sú pod týmto vrcholom. Označme ho $interval(V)$. V našom prípade ak chceme pokryť prvky poľa od 0 po 4, teda čísla $[2, 4, 1, 7, 8]$, tak nám stačí 2. vrchol (= číslo 7 na 2. vrstve), ktorý pokryje $[2, 4, 1, 7]$ a 12. vrchol (číslo 8 na spodnej vrstve). Maximum z poľa $[2, 4, 1, 7, 8]$ nájdeme ako maximum z pokrývajúcich vrcholov, v tomto prípade $\max\{7, 8\}$. Interval čísel od $0 \ldots pokial$ označím $\langle 0 \ldots pokial \rangle$. Operácia **get_max** môže vyzerať aj takto.

```
get_max(pokial){ // vrati maximum intervalu 0...pokial
    return get(pokial, 1);
}

get(pokial, vrchol){ // vrati maximum z prieniku <0...pokial> a intervalu(V)
    if (interval(vrchol) patri cely do <0...pokial>)
        return pole[vrchol];
    if (interval(vrchol) nema nic spolocne s <0...pokial>)
        return 0;
    // inak, je prekryv ciastocny, rozdelim sa na dve casti, pod seba
    return max(get(pokial, 2*v), get(pokial, 2*v+1));
}
```

Možno to tak na prvý pohľad nevyzerá, ale aj toto zaberie $O(\log n)$ krokov, podrobný dôkaz necháme na čitateľa, treba si všimnúť, že vždy keď sa v rekurzii rozdelíme na 2 časti, tak aspoň jedna z nich sa už nebude ďalej deliť a v konštantnom čase vráti odpoveď. Tým pádom na každej z $O(\log n)$ vrstiev strávime konštantný čas.

Implementácia

Namiesto intervalového stromu sa dá použiť aj fínsky, a to preto, že sa hodnoty v poli iba zvyšujú. V nasledujúcom programe je implementácia oboch dátových štruktúr, na ich prepínanie slúži makro FINSKY.

Listing programu (C++)

```
//Fruit of Light
//FoL CC
//Apple Strawberry

#include<cstdio>
#include<algorithm>
#include<vector>
using namespace std;
#define For(i, n) for(int i = 0; i<(n); ++i)

// nechat tak pre Finsky strom, zakomnentovat pre Intervalovy
#define FINSKY

#ifdef FINSKY // ---------- varianta - finsky strom --------------
    #define MAXN 200047
    int fin[MAXN];
    void set(int x, int v){ // nastavi p[x] na v
        for(; x<MAXN; x+=x&(-x)) fin[x] = max(fin[x],v);
    }
    int get(int x){ // vrati maximum p[1..x]
        int v = 0;
        for(; x>0; x-=x&(-x)) v = max(v,fin[x]);
        return v;
    }
#else // ------------- varianta - intervalovy strom --------------
    #define MAXN (1<<18) // MAXN musi byt mocnina 2ky
    int vrcholy[2*MAXN];
    // nastavi p[x] na v
    void set(int x, int v){
        int vrchol = MAXN+x-1; // pre kompatibilitu s finskymn stromom odcitavame 1
        vrcholy[vrchol] = v;
        vrchol/=2;
```

```
        while(vrchol>0){
            vrcholy[vrchol] = max(vrcholy[2*vrchol],vrcholy[2*vrchol+1]);
            vrchol/=2;
        }
    }
    // vrati maximum prieniku intervalov <0..pokial) interval(vrchol) = <zac, kon)
    int get_mit(int pokial, int vrchol, int zac, int kon){
        if (pokial>=kon) return vrcholy[vrchol]; // cely je vnutri
        if (pokial<=zac) return 0; // nema nic spolocne
        return max(
            get_mit(pokial, 2*vrchol, zac, (zac+kon)/2),
            get_mit(pokial, 2*vrchol+1, (zac+kon)/2, kon)
                ); // cast je vnutri, musim sa delit
    }
    int get(int x){ // vrati maximum p[1..x]
        return get_mit(x, 1, 0, MAXN);
    }
#endif

// --------------- samotny program -------------------

int N, X[MAXN], Y[MAXN], F[MAXN], P[MAXN],x;
inline int sort_by_x(int x, int y) {
    return (X[x]==X[y])?Y[x]<Y[y]:X[x]<X[y];
}
inline int sort_by_y(int x, int y) {
    return (Y[x]==Y[y])?X[x]<X[y]:Y[x]<Y[y];
}

int main(){
    scanf("%d␣%d␣%d",&x, &x, &N);
    For(i,N) scanf("%d␣%d␣%d",X+i, Y+i, F+i);
    For(i,N) P[i] = i;
    sort(P, P+N,sort_by_y);
    For(i,N) Y[P[i]] = i+1; // kompresia suradnic
    sort(P, P+N,sort_by_x);
    For(i,N) set(Y[P[i]],get(Y[P[i]])+F[P[i]]); // zistenie hodnoty N-teho policka
    printf("%d\n",get(N)); // finalna odpoved
};
```

opravoval Tomi, vzorák MišoF

2::10. Tunely v hyperpriestore

(max. 10 b za popis, 15 b za program)

Pre jednoduchosť budeme planétu Aspir Adora volať „naša planéta“. A teraz bez nejakých okolkov rovno začneme prvým užitočným pozorovaním: Určite existuje optimálne riešenie, v ktorom všetky tunely, ktoré postavíme, začínajú na našej planéte. Prečo? Majme optimálne riešenie s nejakým iným tunelom, vedúcim z x do y. Ten vieme použiť len tak, že po pôvodných tuneloch prídeme na x a odtiaľ pôjdeme novým tunelom na y. Ak by sme namiesto toho postavili tunel z našej planéty rovno na y, budeme vedieť spraviť to isté s jediným rozdielom – po príchode do y budeme mať k dispozícii viac krokov na cestu ďalej. A keďže toto je pre nás prospešné, aj nové riešenie je určite optimálne.

Výborne, takže už sa potrebujeme len rozhodnúť, ktoré z $n-1$ možných tunelov z našej planéty inam chceme postaviť. Mimochodom, všimnite si, že na predchádzajúcu úvahu sme nijak nepotrebovali poznať štruktúru existujúcej siete tunelov. Na ďalšie úvahy ju však už potrebovať budeme. A odteraz budeme pre jednoduchosť veselo miešať pôvodnú a grafovú terminológiu – planéty sú vrcholy a tunely sú orientované hrany.

Postavme sa do ľubovoľného vrcholu nášho grafu. Z každého vrcholu vedie práve 1 hrana, takže ak sa začneme pohybovať v smere hrán, bude naša prechádzka nekonečná a jednoznačne určená. No a keďže vrcholov je len konečne veľa, časom nejaký vrchol navštívime druhýkrát. A od tejto chvíle teda zjavne chodíme po cykle.

Z tejto úvahy je jasné, ako náš graf (odborne nazývaný „graf s outdegree 1“) vyzerá: je v ňom niekoľko cyklov, a navyše nejaké ďalšie vrcholy. Z každého z tých zvyšných vrcholov vedie cesta na jeden z cyklov. Tieto cesty sa môžu ľubovoľne spájať, celé to teda vyzerá vo všeobecnosti tak, že pre každý vrchol každého cyklu máme v našom grafe jeden strom, ktorý sa v ňom na daný cyklus pripája.

V našom riešení sa najskôr postaráme o stromy a potom o cykly (teda o to, čo z nich ostane, kým sa k nim dopracujeme). Teda, úplne najskôr sa pozrieme, kam sa vieme z našej planéty dostať bez pridávania hrán, a tieto vrcholy označíme ako spracované. A teraz už tie sľubované stromy.

Pozorovanie druhé: V prvom rade si treba uvedomiť, že akonáhle máme nejaký vrchol, ktorý neleží na žiadnom cykle, tak budeme mať vrcholy, do ktorých žiadna hrana nevedie – totiž máme len n hrán a ak do nejakého vrcholu vchádzajú dve, niekde inde musí byť vrchol, do ktorého nevchádza žiadna.

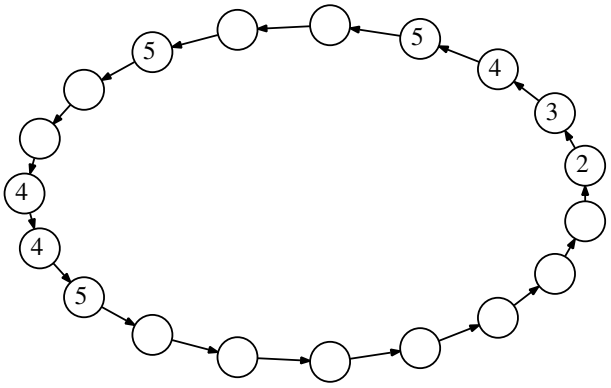
Pozorovanie tretie: Do každého vrcholu, do ktorého nevchádza žiadna hrana, musíme novú hranu pridať, inak sa tam nedostaneme vôbec, tobôž nie na najviac k krokov. Každým pridaním hrany sa nám rozšírila množina vrcholov, kam sa vieme dostatočne rýchlo dostať.

Pozorovanie štvrté: Do vrcholov, do ktorých sa vieme dostatočne rýchlo dostať, nemá zmysel pridávať ďalšie hrany. Dôkaz tohto tvrdenia opäť využíva štruktúru nášho grafu. (Vo všeobecných grafoch toto tvrdenie neplatí, rozmyslite si, ako by vyzeral protipríklad.) Totiž nech sa do v vieme dostať na nanajvýš k krokov. Majme nejaké

optimálne riešenie, v ktorom sme pridali hranu do v. Namiesto nej môžeme pridať hranu do nasledovníka v a nič tým nepokazíme. Opakovaním tejto úvahy sa časom dostaneme do jednej z dvoch možných situácií. Prvá možnosť je, že presunieme hranu na nejaký vrchol, ktorý by sme bez nej nevedeli na k krokov dosiahnuť, a skončíme. Druhá možnosť je, že sa zacyklíme. Táto možnosť ale vlastne nastať nemôže – vtedy totiž všetky vrcholy dosiahnuteľné z v vieme navštíviť na najviac k krokov aj bez pomoci tejto hrany, čo je spor s optimálnosťou riešenia, s ktorým sme začínali.

Tretie a štvrté pozorovanie dokopy nám vlastne kompletne vyriešia stromovú časť grafu. Zoberieme všetky listy stromov, pridáme do nich hrany, označíme za spracované všetky vrcholy ktoré už sú dosiahnuteľné. Ak nám v nespracovanej časti grafu vznikli nejaké nové listy (t.j. vrcholy, kam nevedie hrana zo žiadneho nespracovaného vrcholu), pridáme nové hrany aj do tých. A tak dokola, až kým nám neostanú už len (čiastočne spracované) cykly.

Keď skončíme spracovanie stromov, taký všeobecný cyklus môže vyzerať nejak tak ako na nasledovnom obrázku. V obrázku uvažujeme $k = 5$, čísla vo vrcholoch sú počty krokov, na ktoré sa do nich vieme dostať. Do prázdnych vrcholov sa ešte nevieme dostať dostatočne rýchlo.



Všimnite si postupnosť „4,4,5“ vľavo. To nie je preklep, tá vznikla tak, že sa na cyklus vieme dostať z dvoch rôznych stromov. Do prostredného vrcholu sa teda vieme dostať z predchádzajúceho vrcholu cyklu piatym krokom, alebo z nejakého vrcholu v jeho strome štvrtým krokom.

Teraz ešte potrebujeme pridať hrany do nejakých vrcholov na cykle. Tu stále platí úvaha, že sa neoplatí pridávať hranu do vrcholu, do ktorého sa vieme dostať. Ale neplatí už úvaha, že môžeme pridať hrany do všetkých listov – tie vrcholy, do ktorých momentálne treba $k + 1$ krokov, už totiž nie sú listami v strome. Na našom ukážkovom cykle si môžeme všimnúť, že obe dvojice vrcholov okolo osamoteného vrcholu s číslom 5 vieme obslúžiť tak, že pridáme len jednu novú hranu.

Pohodlné riešenie by bolo teraz spracovať cyklus s n vrcholmi v čase $O(n^2)$: keď si totiž zvolíme jednu hranu, môžeme postupne ísť po cykle a vždy, keď narazíme na vrchol, do ktorého sa ešte nevieme dostať, pridáme ďalšiu. Štandardnými fintami („čo viem spraviť na 2^k krokov?“) sa toto riešenie dá zlepšiť na $O(n \log n)$, čo už je v praxi dostatočne rýchle.

Pre poriadok ale chceme aj teoreticky optimálne, teda lineárne riešenie. V ňom budeme robiť to isté, len trochu šikovnejšie. Použijeme dve zlepšenia: budeme skúšať menej začiatkov a budeme ich skúšať efektívnejšie.

Skúšanie menej začiatkov: Poďme po cykle a rátajme si prázdne vrcholy. Keď ich narátame k, prestaneme. Do aspoň jedného z týchto vrcholov musí ísť v optimálnom riešení nová hrana – ak by nešla do žiadneho, tak k-ty nájdený vrchol nutne ostane prázdny (t.j. nebudeme doň vedieť prísť na k krokov. No a keď dotyčný vrchol použijeme vo vyššie popísanom postupe ako začiatok, zjavne zostrojíme optimálne riešenie. Preto namiesto n možných začiatkov stačí skúšať nejakých, takto zvolených, k začiatkov. (V špeciálnom prípade, keď máme na celom cykle menej ako k prázdnych vrcholov, vyskúšame jednoducho všetky.)

Efektívnejšie skúšanie konkrétneho začiatku: Pred tým, ako čokoľvek začneme skúšať, si niečo šikovne predpočítame. Konkrétne, pre každý vrchol x chceme odpoveď na nasledovnú otázku: „Ak máme cyklus v jeho súčasnej podobe a pridáme hranu do x, ktorý bude prvý prázdny vrchol na cykle (v smere od x)?“ Toto predpočítanie vieme spraviť v čase $O(n)$ na jeden prechod cyklom. V rámci prechodu si udržiavame dva pointre (na x a na vrchol, ktorý je preň odpoveďou) a pamätáme si ich aktuálnu vzdialenosť. Vždy, keď posunieme x, posúvame aj odpoveď, až kým súčasne neplatí (odpoveď je prázdny vrchol) a (vzdialenosť x a odpovede je aspoň k).

Pomocou takto predpočítaných údajov vieme pridávanie hrán pre konkrétny začiatok odsimulovať rýchlejšie. Konkrétnejšie, každé pridanie hrany vieme realizovať v konštantnom čase. No a keďže každá nová hrana pokryje k vrcholov cyklu, pridaných hrán bude najviac $\lceil n/k \rceil$.

Tak si to zosumarizujme. Najskôr predpočítame pomocné údaje v čase $O(n)$. Potom skúšame k možných začiatkov a pre každý z nich urobíme najviac $\lceil n/k \rceil$ krokov, čo je dokopy opäť $O(n)$. Teda celé naše riešenie má

lineárnu časovú zložitosť, čo je zjavne optimálne.

Na záver si môžete pozrieť implementáciu vzorového riešenia z istej staršej CEOI, z ktorej táto úloha pochádza.

Listing programu (C++)

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;

#define MAXN 1000000
const int inf = 1000000000;

int n, k;
int link[MAXN];
int dead[MAXN];
vector<int> adj[MAXN];

vector< vector<int> > ciklusi;

int result = 0;

void nadji_cikluse() {
   vector<int> bio( n, 0 );
   int time = 0;

   for( int i = 0; i < n; ++i ) {
      if( bio[i] ) continue;
      int a, b;

      bio[i] = ++time;
      for( a = link[i]; !bio[a]; a = link[a] )
         bio[a] = ++time;

      if( bio[a] < bio[i] ) continue;

      ciklusi.push_back( vector<int> (1,a) );
      for( b = link[a]; b != a; b = link[b] )
         ciklusi.back().push_back( b );
   }
}

int rec( int a ) {
   int kill = 0;
   for( vector<int>::iterator it = adj[a].begin(); it != adj[a].end(); ++it )
      kill = max(kill, rec( *it ));
   if( a == 0 ) kill = k+1;

   dead[a] = 1;
   if( kill == 0 ) {
      kill = k;
      ++result;
   }
   return kill-1;
}

int main( void ) {
   scanf( "%d%d", &n, &k );
   for( int i = 0; i < n; ++i ) { int b; scanf("%d", &b); link[i] = b-1; adj[b-1].push_back(i); }

   nadji_cikluse();

   for( vector< vector<int> >::iterator it = ciklusi.begin(); it != ciklusi.end(); ++it ) {
      vector<int> &ciklus = *it;
      int m = ciklus.size();
      {
         int kill = 0;
         int prev = ciklus[m-1];
         for( int i = 0; i < 2*m; ++i ) {
            int curr = ciklus[i%m];

            if( i < m ) {
               for( vector<int>::iterator it = adj[curr].begin(); it != adj[curr].end(); ++it ) {
                  if( *it == prev ) continue;
                  kill = max(kill, rec( *it ));
               }
               if( curr == 0 ) kill = k+1;
            }
            if( kill-- > 0 ) dead[curr] = 1;

            prev = curr;
         }
      }
      vector<int> next( m, inf );
      {
         int prev = 0;
         for( int i = 2*m-1; i >= 0; --i ) {
            int curr = i%m;

            next[curr] = next[prev]+1;
            if( !dead[ciklus[curr]] ) next[curr] = 0;

            prev = curr;
         }
      }
```

```
      if( next[0] >= inf ) continue;
      {
         int best = 1000000;
         for( int start = 0; start < min(k,m); ++start ) {
            int cost = 0;
            int i = start + next[start];
            while( i - start < m ) {
               ++cost;
               i += k;
               i += next[i%m];
            }
            best = min(best, cost);
         }
         result += best;
      }
   }
   printf( "%d\n", result );
   return 0;
}
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09