**Korešpondenčný seminár z programovania**
**XXX. ročník, 2012/13**
**Katedra základov a vyučovania informatiky FMFI UK,**
**Mlynská Dolina, 842 48 Bratislava**

# Vzorové riešenia 1. kola zimnej časti

vzorák napísal Hermi

## 1. Za mreže s nimi!

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

Väčšinu bodov sa dalo získať aj za veľmi jednoduché riešenia. Ak sú IDčka detí malé, môžeme si napríklad urobiť pole boolovských premenných, označiť si v ňom, ktoré IDčka sme videli v prvom poli, potom odznačiť tie, ktoré sú aj v druhom, a jediné, čo nám ostane, je hľadaná odpoveď. Ak sú IDčka detí veľké, ale samotných detí je málo, môžeme postupne pre každý prvok prvého poľa prejsť celé druhé pole a zistiť, či sa nachádza aj v ňom. Na plný počet bodov bolo ale treba efektívnejšie riešenie – na to s boolovským poľom nám nestačila pamäť, to druhé zase na veľkých vstupoch mnohonásobne prekročí časový limit.

Pri riešení úlohy by nám výrazne pomohlo usporiadať obe postupnosti. Predpokladajme teda, že sme tak spravili a v každej postupnosti už máme čísla usporiadané od najmenšieho po najväčšie. Čo ďalej? Začneme postupne prechádzať naraz obe postupnosti, začínajúc od najmenších IDčiek. Vždy sa pozrieme na ďalšie IDčko v prvej postupnosti, a overíme, že je (aspoň raz) aj na aktuálne skúmanom mieste v druhej. Ak nie, našli sme, čo hľadáme. Ak áno, môžeme v oboch postupnostiach preskočiť všetky jeho výskyty a pokračovať ďalej.

Ak použijeme efektívny algoritmus na triedenie, dostaneme usporiadané postupnosti v čase $O(m \log m + n \log n)$. Následný prechod nimi a nájdenie nezhody už vieme spraviť v lineárnom čase.

Väčšina programovacích jazykov má na efektívne triedenie knižničnú funkciu. Implementácia tohto riešenia je potom veľmi jednoduchá.

### Listing programu (Python)

```
from sys import stdin
M, N = [ int(x) for x in stdin.readline().split() ]
A = [ int(x) for x in stdin.readline().split() ]
B = [ int(x) for x in stdin.readline().split() ]

A.sort()
B.sort()
a, b = 0, 0
while A[a] == B[b]:
    teraz = A[a]
    while a<len(A) and A[a] == teraz: a += 1
    while b<len(B) and B[b] == teraz: b += 1

print A[a]
```

Podobne efektívne riešenie vieme spraviť aj pomocou pokročilých dátových štruktúr, ako napríklad usporiadanej množiny (set), implementovanej pomocou vyvažovaného binárneho stromu. Kto už vie takého dátové štruktúry používať, tomu sa body získava ľahko:

### Listing programu (C++)

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

int main() {
        int M, N; cin >> M >> N;
        set<int> deti;
        while (M--) { int x; cin >> x; deti.insert(x); }
        while (N--) { int x; cin >> x; deti.erase(x); }
        cout << (*deti.begin()) << endl;
}
```

Najťažšie to mali programátori v Pascale, ktorý tých užitočných knižníc zatiaľ priveľa neponúka. Oplatí sa vedieť, že v dokumentácii FreePascalu sa dá nájsť aj súbor `qsort.pp`, v ktorom je implementované šikovné triedenie QuickSort. Ale každý dobrý programátor by mal aj sám vedieť efektívne triedenie naprogramovať. Ak to ty ešte nevieš, odporúčame ti navštíviť našu Liaheň (`https://liahen.ksp.sk/`), kde si môžeš prečítať podrobný popis viacerých dobrých algoritmov.

V našom pascalovskom programe sme implementovali triedenie HeapSort. Na triedenie teda použijeme špeciálnu stromovú dátovú štruktúru menom halda. V nej platí, že prvok v každom vrchole je väčší ako jeho

potomkovia. Z tejto vlastnosti vyplýva, že v koreni haldy bude vždy uložené najväčšie číslo. Pomocou haldy vieme ľahko usporiadať $n$ prvkov v čase $O(n \log n)$ tak, že vždy najväčší prvok z vrchu haldy vyberieme, dáme ho na koniec poľa a zvyšok poľa upraceme tak, aby ďalej spĺňal vlastnosť haldy.

### Listing programu (Pascal)

```
program Vzorak;
const
  MAX_N = 1000047;

var
  A,B: array[0..MAX_N] of longint;
  n,m,i,pa,pb,id: longint;

procedure vymen(var a, b: longint);
var
  temp: longint;
begin
  temp:=a; a:=b; b:=temp;
end;

procedure heapsort(var A: array of longint; size: longint);

  procedure prebubli(p: longint);
  var
    syn: longint;
  begin
    syn:=p*2;
    if syn<=size then begin
      if (A[syn+1]>A[syn]) and (syn<size) then inc(syn);
      if A[p] < A[syn] then begin
        vymen(A[syn],A[p]);
        prebubli(syn);
      end;
    end;
  end;

begin
  for i := size div 2 downto 1 do
    prebubli(i);
  for i := size downto 2 do begin
    vymen(A[i],A[1]);
    dec(size);
    prebubli(1);
  end;
end;

begin
  readln(N,M);
  for i := 1 to n do
    read(A[i]);
  for i := 1 to m do
    read(B[i]);
  heapsort(A, n);
  heapsort(B, m);
  pa:=1;
  pb:=1;
  while (pa<=n) and (pb<=m) do begin
    id:=A[pa];
    if (A[pa]<>B[pb]) then begin
      writeln(id);
      break;
    end;
    while (A[pa] = id) and (pa<n) do inc(pa);
    while (B[pb] = id) and (pb<m) do inc(pb);
  end;
end.
```

vzorák napísal Raf

## 2. Zahrajme si!

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

Keďže vyskúšať všetky možné spôsoby zahrania zadanej skladby by bolo príliš zdĺhavé, zamyslime sa nad niekoľkými intuitívne správnymi zásadami, ako hýbať rukou. V prvom rade ňou budeme hýbať len vtedy, ak z aktuálnej pozície nedosiahneme na kláves, ktorý chceme zahrať. A keď sa aj pohneme, bude to len o najmenšiu vzdialenosť, ktorú potrebujeme na dosiahnutie požadovaného klávesu.

Nech potrebujeme stlačiť kláves na pozícii $p$, pričom ruku máme položenú na klávesoch $x, x+1, \ldots, x+\ell-1$. Potom ak sa kláves nachádza naľavo od $x$ (teda $p < x$), tak ľavý koniec ruky treba posunúť na pozíciu $p$ a bude na to treba prejsť vzdialenosť $x - p$. Ak sa kláves nachádza napravo od ruky (teda $p > x + \ell - 1$), tak musíme ruku posunúť tak, aby daný kláves bol na pravom okraji jej rozsahu. Ľavý okraj ruky bude potom na pozícii $p - \ell + 1$ a prejdeme takto vzdialenosť $p - \ell + 1 - x$.

Ako ale ukázať, že týmto spôsobom nájdeme skutočne optimálne riešenie?

Vezmime si také optimálne riešenie $M$, ktoré sa od toho nášho (označme ho $R$) líši čo najneskôr (maximalizujeme poradové číslo noty, kvôli ktorej $M$ po prvýkrát vykoná iný pohyb ako $R$). Ak sú totožné, tešíme sa.

Inak sa bližšie pozrime na prvú nezhodu. Ruka je zatiaľ v oboch riešeniach na rovnakej pozícii, no teraz sa optimálne riešenie pohne inak ako to naše. Rozoberme si možnosti:

- $M$ sa pohlo (doľava alebo doprava), no $R$ stojí na mieste. Z toho vyplýva, že ruka dosiahne na požadovaný kláves. Upravme riešenie $M$ tak, aby najprv stlačilo kláves až potom vykonalo tento pohyb. Dostaneme tak riešenie $M'$, ktoré bude tiež optimálne, no od $R$ sa líši neskôr – čo je v spore s voľbou $M$.
- $R$ sa pohlo, no $M$ stojí na mieste. Nemôže nastať – hýbeme sa len vtedy, keď je to nutné.
- $M$ sa pohlo aj doľava, aj doprava – no toto je zjavný nezmysel.
- $M$ sa pohlo opačným smerom ako $R$. Ako chce potom dosiahnuť požadovaný kláves?
- $M$ aj $R$ sa pohli rovnakým smerom, no o rôznu vzdialenosť. Označme vzdialenosti, o ktoré sa pohli jednotlivé riešenie, ako $m$ a $r$. Zrejme platí $m > r$. Upravme $M$ tak, aby sa najprv pohlo iba o $r$ pozícií, potom stlačilo požadovaný kláves, potom sa pohlo o zvyšných $m - r$ pozícií a pokračovalo v pôvodnom pláne. Dostaneme tak iné optimálne riešenie $M'$, ktoré sa opäť líši od $R$ neskôr, čo je v spore s voľbou $M$.

Každá z rozobraných možností nás doviedla do sporu, takže $M = R$ a riešenie nájdené naším algoritmom je optimálne.

Aká bude časová zložitosť nášho algoritmu? Každú načítanú notu spracujeme v konštantnom čase, teda dokopy dosiahneme zložitosť $O(n)$. Riešenie s menšou zložitosťou zjavne neexistuje, lebo určite potrebujeme aspoň načítať vstup. Pamäťová zložitosť bude $O(1)$, lebo v každom kroku nám stačí pamätať si konštantný počet údajov.

Ešte poznámka k implementácii: Celková vzdialenosť, ktorú prejdeme rukou, môže presiahnuť rozsah 32-bitovej celočíselnej premennej, preto je potrebné použiť v Pascale typ `int64`, resp. v C++ `long long`.

### Listing programu (Pascal)

```
program klavir;
var n,k,l,x,pozicia,vysledok:int64;
i:longint;

begin
  readln(n,k,l);
  pozicia:=0; vysledok:=0;
  for i:=1 to n do begin
    read(x);
    if (x<pozicia) then begin
      {ak je nota nalavo od ruky}
      vysledok:=vysledok+pozicia-x;
      pozicia:=x;
    end
    else if (x>pozicia+l-1) then begin
      {ak je nota napravo od ruky}
      vysledok:=vysledok+x-(pozicia+l-1);
      pozicia:=pozicia+x-(pozicia+l-1);
    end;
    {ak je nota v rozsahu ruky, tak ruku neposuvame}
  end;
  writeln(vysledok);
end.
```

vzorák napísal Mio

## 3. Záhadné umenie

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

Treba si uvedomiť, že každý bod môže byť vrcholom trojuholníka pri pravom uhle. Navyše, ak je nejaký bod $A$ vrcholom pri pravom uhle, tak jeden zo zvyšných dvoch vrcholov trojuholníka musí mať rovnakú $x$-ovú súradnicu a druhý musí mať rovnakú $y$-ovú súradnicu ako $A$ (keďže odvesny majú byť rovnobežné so súradnicovými osami). Takže pre každý bod $A$ vieme vypočítať počet trojuholníkov, v ktorých je $A$ pri pravom uhle. Môžeme si totiž vybrať ľubovoľný bod s rovnakou $x$-ovou súradnicou (okrem samotného $A$) a k nemu ľubovoľný bod s rovnakou $y$-ovou súradnicou (samozrejme okrem $A$).

Nech $p_x(k)$ je počet bodov s $x$-ovou súradnicou rovnou $k$ a $p_y(\ell)$ je počet bodov s $y$-ovou súradnicou rovnou $\ell$. Potom počet trojuholníkov s $i$-tym bodom pri pravom uhle vypočítame takto:

$$t_i = (p_x(x_i) - 1) \cdot (p_y(y_i) - 1)$$

Výsledný počet bude súčet týchto hodnôt pre všetky body:

$$t = \sum_{i=1}^{n} t_i$$

Treba ešte vypočítať $p_x(x)$ a $p_y(y)$ pre všetky hodnoty $x$ a $y$. Jednoduchým spôsobom je mať polia[1] `PX[]` a `PY[]`, ktoré si na začiatku vynulujeme a pre každý bod $(x_i, y_i)$ zvýšime počítadlá `PX[`$x_i$`]` a `PY[`$y_i$`]` o jedna.

Pár slov k zložitosti. Veľkosť súradnicovej sústavy považujeme za konštantu, takže v zložitostiach ju nebudeme uvažovať. Každý bod najskôr započítame do `PX` a `PY` a potom preň spočítame počet trojuholníkov (v konštantnom čase). Keďže máme $n$ bodov, tak výsledná časová zložitosť je $O(n)$. Potrebujeme si pamätať len počty pre jednotlivé súradnice (čo sme označili za konštantu) a jednotlivé body, čiže pamäťová zložitosť bude $O(n)$.

Riešenia vo čase $O(n)$ môžu dostať plných 7 bodov, za časovú zložitosť $O(n^2)$ najviac 5 bodov. Okrem toho sa budú strhávať body za zlé odhady zložitostí, horšiu pamäťovú zložitosť alebo nedostatočný popis.

### Listing programu (Pascal)

```
program zahadne_umenie;
var n, i : longint;
    x, y : array [1..100000] of longint;
    PX, PY : array[1..100000] of longint;
    ret : int64;
begin
  readln( n );
  {vynulujem pole pre súradnice}
  for i := 1 to 100000 do begin
    PX[i] := 0;
    PY[i] := 0;
  end;
  for i := 1 to n do begin
    readln( x[i], y[i] );
    {započítam bod do počítadla x-ovej súradnice}
    PX[ x[i] ] := PX[ x[i] ] + 1;
    {započítam bod do počítadla y-ovej súradnice}
    PY[ y[i] ] := PY[ y[i] ] + 1;
  end;
  ret := 0;
  for i := 1 to n do begin
    {pripočítam počet trojuholníkov s i-tym vrcholom pri pravom uhle (výsledok
     sa nemusí zmestiť do 32 bitov)}
    ret := ret + int64(PX[ x[i] ]-1) * int64(PY[ y[i] ]-1);
  end;
  writeln( ret );
end.
```

vzorák napísal JaNo

## 4. Závody korytnačiek

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

Keď chceme naprogramovať niečo rýchle, môže nám pomôcť uvedomiť si, kde viaznu pomalé riešenia. V tomto príklade sú pomalé riešenia také, ktoré sa v každom ťahu dotknú všetkých korytnačiek, ktoré sa majú presunúť – teda pre každú korytnačku, ktorá sa v ťahu presúva, musia zmeniť nejaké hodnoty vo svojich dátových štruktúrach.

Keďže v každom ťahu sa môžu presunúť obrovské množstvá korytnačiek, znamená to obrovské množstvo operácií, čomu sa chceme vyhnúť.

Vzorové riešenie, ktoré sa pokúsime nájsť, môže v každom ťahu upraviť len zopár hodnôt bez ohľadu na to, koľko korytnačiek sa presúva.

Tým pádom si musíme nájsť nejakú vlastnosť, ktorá sa pri presunoch skoro vôbec nemení. Napríklad si skúsme pre každú korytnačku pamätať iba to, ktorá korytnačka je hneď pod ňou, prípadne ktoré políčko je hneď pod ňou. Môžeme si všimnúť, že v jednom ťahu sa táto hodnota zmení len pre jednu jedinú korytnačku – tú, ktorou hýbeme. Ostatné korytnačky budú mať pod sebou stále to isté.

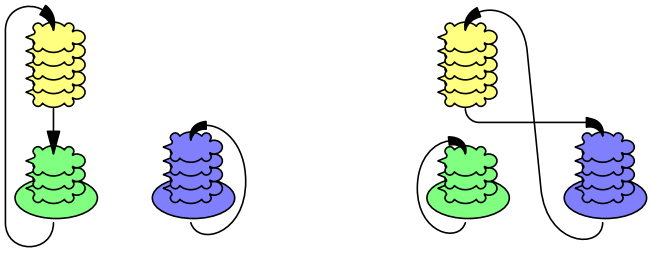
Ostáva nám vyriešiť zopár vecí na záver.

Pokiaľ presúvame korytnačku, potrebujeme vedieť, ktorá korytnačka bude pod ňou po presune. Samozrejme, bude to vrchná korytnačka cieľového políčka. Políčko síce poznáme, ale korytnačku na jeho vrchu bohužiaľ nie. Preto si budeme pre každé políčko pamätať najvrchnejšiu korytnačku (alebo niečo špeciálne, ak je políčko prázdne).

Táto hodnota sa v každom ťahu zmení najviac dvom políčkam, čo je stále dobre. Na začiatku teda povieme každej korytnačke, ktorá korytnačka/políčko je pod ňou.

Následne spravujeme všetky ťahy. Spracovanie jedného ťahu spočíva v presmerovaní troch šípok. Dvom políčkam sa zmení najvrchnejšia korytnačka a jednej korytnačke sa zmení korytnačka, ktorá je pod ňou. Lepšie to asi znázorní nasledovný obrázok, naľavo je pôvodný stav, napravo je stav po tom, ako sa 5 žltých korytnačiek presunie na 4 modré.

[1] Fajnšmekri mohli použiť hashovanie, ale v tomto príklade sa to nevyžadovalo, ani za to neboli body navyše.



Jeden ťah nás stojí čas $O(1)$.

Na konci iba pre každé políčko prejdeme korytnačky od hornej až dolu a vypíšeme ich v poradí zdola hore. Dokopy celý algoritmus spraví $O(K+P+T)$ krokov a jeho pamäťová zložitosť je $O(K+P)$.

Mimochodom, skúste sa zamyslieť, prečo na vstupe nebolo len číslo korytnačky a o koľko sa má posunúť, hoci týmto je ťah jednoznačne určený. Čo by sa stalo, keby sme vám políčko, na ktorom korytnačka je, nezadali? S týmto súvisí aj zaujímavosť tohoto príkladu, že na výrobu vstupov k tejto úlohe bol použitý zložitejší a rádovo pomalší program ako na riešenie.

### Listing programu (C++)

```
#include<cstdio>
#include<algorithm>
#include<stack>
using namespace std;
#define For(i, n) for(int i = 0; i<(n); ++i)
#define MAXN 500047

// prvých K prvkov je pre korytnačky,
//od MAXN vyššie je zapamätaná horná korytnačka pre políčko
int pod[2*MAXN];
int K,P,T,a,b,ktora, odkial, kolko, kam, horna;
stack<int> vystup;

int main() {
  scanf("%d%d%d",&K,&P,&T);
  // nastavíme pole 'pod'
  For(i,P) pod[MAXN+i] = MAXN+i;
  a = MAXN;
  For(i,K) {
    b = a;
    scanf("%d", &a);
    pod[a] = b;
  }
  pod[MAXN] = a;
  // T ťahov
  For(i, T) {
    scanf("%d%d%d",&ktora, &odkial, &kolko);
    odkial--;
    kam = odkial+kolko;
    // prešípkujeme
    horna = pod[MAXN+odkial];
    pod[MAXN+odkial] = pod[ktora];
    pod[ktora] = pod[MAXN+kam];
    pod[MAXN+kam] = horna;
  }
  // výpis výstupu
  For(i, P) {
    // aby bol výstup v opačnom poradí, nahádžeme čísla do stacku a povyberáme
    ktora = pod[MAXN+i];
    while(ktora<MAXN) {
      vystup.push(ktora);
      ktora = pod[ktora];
    }
    printf("%d",vystup.size());
    while(!vystup.empty()) {
      printf(" %d",vystup.top());
      vystup.pop();
    }
    printf("\n");
  }
}
```

## 5. Oko kukne, oko vidí (časť prvá)

opravovalo sa samo, vzorák napísal MišoF
(max. 0 b za popis, 16 b za program)

Bez dlhého okúňania poďme rovno na riešenie jednotlivých podúloh.

1. *Riadky, v ktorých je druhé písmeno „*`a`*“.*

   Tento regulárny výraz je veľmi jednoduchý: chceme vidieť začiatok riadku, ľubovoľný znak a znak „`a`“. Riešením je teda výraz „`^.a`“.

2. *Riadky, v ktorých sa písmeno „a“ nevyskytuje vôbec.*

   Ako vyzerajú takéto riadky? Od začiatku až po koniec sú tam samé znaky odlišné od „a“. Toto v reči regulárnych výrazov zapíšeme napríklad nasledovne: „`^[^a]*$`“.

3. *Riadky, v ktorých sa písmeno „a“ vyskytuje aspoň 3×.*

   V takomto riadku niekde máme „a“, potom ľubovoľne veľa znakov, potom druhé „a“, opäť ľubovoľne veľa znakov, a na koniec tretie „a“. Dostávame teda nasledovný výraz: „`a.*a.*a`“.

   Tu už začína byť pekne vidieť, v čom je sila regulárnych výrazov. My nemusíme písať žiadne cykly, ktoré niečo rátajú a kontrolujú. Len popíšeme, čo chceme nájsť, a necháme program (resp. pri iných použitiach regulárnych výrazov knižnicu), nech danú vzorku nájde, prípadne overí, že sa v danom riadku nenachádza.

   (Iným riešením bolo namiesto „`.`“ použiť „`[^a]`“. Takto by sme našli tri po sebe idúce výskyty „a“.)

4. *Riadky, v ktorých sa písmeno „a“ vyskytuje presne 3×.*

   Teraz už opäť musíme kontrolovať celý riadok. Nestačí predpísať, že sa v ňom trikrát vyskytne „a“, potrebujeme aj špecifikovať, že všetky ostatné znaky sa od „a“ líšia. Riešenie: „`^[^a]*a[^a]*a[^a]*a[^a]*$`“.

   Regulárne výrazy tiež umožňujú aj špecifikovať, že sa konkrétna vzorka má opakovať presne daný počet ráz. Toto rozšírenie sme v zadaní síce nepopísali, ale kto sa pustil do štúdia, určite ho objavil, a tak mohol napísať aj stručnejšie riešenie. Napríklad takéto: „`^([^a]*a){3}[^a]*$`“. Alebo takéto: „`^([^a]*a[^a]*){3}$`“.

5. *Riadky, v ktorých niektoré slovo začína písmenom „a“ alebo „A“.*

   Tu sa dalo použiť ďalšie dva syntaktické prvky popísané v zadaní: špeciálny znak pre hranicu slova a logický or. Riešenie: „`\ba|\bA`“. Prípadne to ide aj bez logického or-u: „`\b[aA]`“.

   Ak si nie sme istí prioritami operátorov, nič nepokazíme zátvorkami. Fungovalo by teda aj nasledovné: „`(\ba)|(\bA)`“.

   Pozor ale na konštrukciu typu „`[^a-zA-Z0-9_][aA]`“! Tá totiž zlyhá, ak je hľadané slovo na úplnom začiatku riadku. To potom treba dodatočne ošetriť. Touto cestou sa teda dopracovať k riešeniu dá tiež, ale je komplikovanejšie: „`[^a-zA-Z0-9_][aA]|^[aA]`“.

6. *Riadky, v ktorých sú aspoň dva znaky a prvý znak je rovnaký ako posledný.*

   Tu prvýkrát použijeme spätnú referenciu. Na začiatku riadku si znak označíme a na konci riadku budeme požadovať ten istý znak: „`^(.).*\1$`“.

7. *Riadky, v ktorých sa niektoré slovo zopakuje.*

   Tu už ide do tuhého. Potrebujeme si označiť nejaké slovo a potom vyžadovať, aby sa niekedy neskôr zopakovalo. Skúsme najskôr napísať vzorku, ktorej bude zodpovedať ľubovoľné jedno slovo.

   Prvý pokus by mohol vyzerať nejak takto: „`\b.*\b`“. Hranica slova, ľubovoľne veľa znakov, ďalšia hranica slova. V čom je problém? Je ich tam hneď viac. Prvým problémom je, že aj nula je ľubovoľne veľa. Tejto vzorke teda zodpovedá aj prázdny reťazec na hranici slova.

   Tak druhý pokus: „`\b..*\b`“ (teda najskôr „bodka“, potom „bodka s hviezdičkou“). Takto si vynútime, že vnútri vo vzorke je aspoň jeden znak. Ani toto však ešte nie je ono – totiž tejto vzorke bude zodpovedať napríklad aj úsek tvaru „koniec slova, medzera, začiatok slova“.

   Poučení z krízového vývoja to už radšej napíšeme explicitne – hranica slova, aspoň jeden znak tvoriaci slovo, hranica slova: „`\b[a-zA-Z0-9_][a-zA-Z0-9_]*\b`“.

   Opäť platí, že kto si pozrel viac syntaxe, vedel to isté zapísať aj stručnejšie. Napríklad môžeme použiť namiesto kvantifikátora `*` kvantifikátor `+` (ľubovoľne veľakrát, ale aspoň raz).

   Teraz už sme takmer hotoví. Vieme napísať vzorku pasujúcu na ľubovoľné slovo, tak si ju teda označme. Následne povolíme ľubovoľne veľa znakov a potom budeme vyžadovať zopakovanie reťazca, ktorý vzorke zodpovedal: „`\b([a-zA-Z0-9_]+)\b.*\1`“.

   Takmer správne, ale ešte to nie je ono. Totiž takýto regulárny výraz vieme napríklad nájsť aj v riadku „`pes spestrel`“. A už je jasné, kde je pes zakopaný – zabudli sme ešte požadovať, že aj okolo druhého výskytu musia byť hranice slova.

   Správne riešenie teda vyzerá napríklad nasledovne: „`\b([a-zA-Z0-9_]+)\b.*\b\1\b`“.

8. *Riadky, v ktorých sa vyskytuje (možno aj 1-znakové) slovo začínajúce aj končiace tým istým znakom.*

   Tu rozlíšime dve možnosti: buď má dané slovo aspoň dva znaky, alebo má práve jeden znak. Jednoznakové slová zjavne vyhovujú všetky. No a u viacznakových skombinujeme techniky z predchádzajúcich úloh: označíme si prvý znak slova a budeme vyžadovať, aby sa mu posledný rovnal.

   Výsledok môže vyzerať nasledovne: „`\b([a-zA-Z0-9_])[a-zA-Z0-9_]*\1\b|\b[a-zA-Z0-9_]\b`“.

A to už je naozaj všetko. Nabudúce sa môžete tešiť na príkaz `sed` – nebudeme už len vyhľadávať, ale aj nahrádzať.

vzorák napísal Žaba
(max. 13 b za popis, 7 b za program)

## 6. Odmena za lenivosť

Pri problémoch ako tento je veľmi dobré mať k dispozícii vhodný arzenál už predpripravených vecí, čo nám neposkytujú všetky jazyky. Napr. Pascal je dosť obmedzujúci. Preto je v súťažnom programovaní dobré ovládať C++ a niektoré nástroje, ktoré ponúka. Samozrejme to nie je podmienka, však aj samotný **tourist**[2] rieši niektoré súťaže v Delphi.

Poďme sa teraz vrhnúť na riešenie, kde si ukážeme, ktoréže to dátové štruktúry sú také nápomocné a ako ich správne používať.

### Prvé pozorovania

Pozrime sa na to, ako sa dá pre daný úsek dĺžky $\ell$ zistiť maximálna hodnota, ktorú vie dosiahnuť v absolútnej hodnote, ak môžeme najviac $k$ čísiel prenásobiť $-1$. Dôležité je, že môžeme zmeniť znamienko aj menej ako $k$ číslam. To znamená, že každému prvku úseku stačí zmeniť znamienko najviac raz (keďže $(-1)\cdot(-1)=1$).

Maximalizovať súčet úseku v absolútnej hodnote znamená buď maximalizovať súčet, alebo minimalizovať súčet. Pre rôzne úseky sa môže oplatiť odlišná stratégia, vždy preto vyskúšame obe. Ak sa napríklad snažíme súčet úseku minimalizovať, určite nebudeme meniť znamienko záporným číslam v tomto úseku – tým by sme si len poškodili. Najradšej by sme boli, ak by sa nám podarilo zmeniť znamienko všetkým kladným hodnotám. V prípade, že ich je viac ako $k$, uprednostníme $k$ najväčších kladných čísiel, keďže ich zmena na záporné nám pomôže najviac. Druhá možnosť, v ktorej sa snažíme súčet úseku maximalizovať, vyzerá analogicky.

Nech je teda súčet úseku $s$, súčet $k$ najmenších záporných čísiel $s_-$ a súčet $k$ najväčších kladných $s_+$. Potom v absolútnej hodnote najväčší dosiahnuteľný súčet tohto úseku je $\max(|s-2s_-|,|s-2s_+|)$. Všimnime si, že nestačí pozrieť sa len na znamienko čísla $s$ a podľa neho sa rozhodnúť o správnej stratégii. Napríklad pre úsek $(100,-1,-2,-3)$ a $k=1$ vieme dosiahnuť súčet 106, len keď zmeníme 100 na záporné číslo – aj keď je $s$ kladné.

### Magické krabičky

Keď prechádzame postupne všetky súvislé úseky dĺžky $\ell$, tak dva po sebe idúce úseky sa od seba veľmi nelíšia – jedno číslo pribudne a druhé odbudne. Načo teda zahadzovať všetky informácie čo už máme, stačí ich len nejak vhodne upraviť. Čo si vlastne potrebujeme pamätať?

Určite súčet celej postupnosti. To však nie je veľký problém. Vždy len pričítame nové číslo a odčítame staré. Ale tiež potrebujeme súčet $k$ najmenších a $k$ najväčších čísiel. (Nebudem to vždy písať, ale samozrejme myslím najmenšie len zo záporných a najväčšie len z kladných. A keď ich bude menej ako $k$, vezmeme všetky.) Tu však už nestačí pamätať si len ten súčet, lebo by sme ho nevedeli meniť. Potrebujeme totiž vedieť zistiť, či to číslo, ktoré odbudlo/pribudlo, náhodou nepatrí medzi tých $k$ vybraných.

Musíme si preto pamätať aj to, ktoré konkrétne čísla sú momentálne medzi $k$ najmenšími/najväčšími. Predstavme si zatiaľ teda, že máme čarovnú krabičku, do ktorej vieme čísla vkladať, vyberať ich, a ktorej sa vieme pýtať na súčet $k$ najmenších/najväčších čísiel v nej. Aby sme toho nechceli veľa, tak budeme mať jednu krabičku pre kladné a druhú pre záporné čísla.

Riešenie je teraz už veľmi priamočiare. Najskôr prejdeme prvých $\ell$ prvkov, pričom si zapamätáme ich súčet a jednotlivé čísla vložíme do správnych krabičiek. Teraz už len budeme robiť nasledovné. Podľa vzorca vyššie zistíme v absolútnej hodnote maximálny súčet tohto úseku. Pridáme ďalšie číslo v poradí (teda číslo na pozícii $\ell+1$) a odoberieme prvé číslo (a príslušne upravíme krabičky). Takto dostaneme úsek, ktorý je ďalší v poradí a tento postup môžeme opakovať až po posledné číslo. Medzitým si samozrejme budeme pamätať maximum z doteraz videných výsledkov a to vypíšeme ako odpoveď.

Stačí už len implementovať magické krabičky. Tie budú obsahovať nasledovné veci:

- súčet najmenších/najväčších $k$ čísiel (záleží od toho, v ktorej krabičke to je)

[2] Genadij Korotkjevič, najúspešnejší účastník Medzinárodnej olympiády v informatike

- množinu[3] týchto $k$ čísiel
- množinu zvyšných čísiel patriacich do tejto krabičky

Teraz vieme veľmi ľahko riešiť požiadavky na celú krabičku (povedzme, že pre tú s kladnými číslami a maximom, tá druhá bude fungovať analogicky). Keď pridáme nové číslo, vložíme ho najprv do prvej množiny. Ak sme týmto krokom presiahli jej veľkosť $k$, najmenšie z čísiel v nej presunieme do druhej množiny. Popri tom si ešte udržiavame správnu hodnotu súčtu čísiel v prvej množine. Keď chceme nejaký prvok odstrániť, tak sa pozrieme do oboch množín a z jednej ho odstránime. Ak nám týmto klesla veľkosť prvej množiny pod $k$ a druhá množina je neprázdna, presunieme najväčší prvok druhej množiny do prvej (a upravíme jej súčet). A keď chceme zistiť súčet $k$ najväčších kladných čísiel, stačí sa pozrieť na hodnotu, ktoré si pamätáme.

Už sme si povedali celú myšlienku tejto úlohy. Poslednou otvorenou otázkou zostalo, ako efektívne simulovať tie množiny. Avšak tí, čo poznajú čarovné slovíčko *set* a vedia ho aj používať, by teraz už mali mať presnú predstavu o tom, ako bude vyzerať riešenie – preto môžu preskočiť nasledujúcu pasáž. A my ostatní sa poďme vrhnúť do sveta binárnych vyhľadávacích stromov.

### Binárne vyhľadávacie stromy

Strom je dátová štruktúra na ukladanie prvkov, ktorá naozaj vyzerá ako taký strom, len rastie opačným smerom. Skladá sa z vrcholov a ukazovateľov. Vrcholy, do ktorých vedú ukazovatele z vrcholu $v$, nazývame jeho synmi a on im je otcom. Vrcholy, ktoré na nič neukazujú, sú listy. Na vrchu stromu je takzvaný koreň, čo je jediný vrchol, do ktorého nevedie žiaden ukazovateľ. Do ostatných vrcholov vedie práve jeden ukazovateľ a každý vrchol môže ukazovať na ľubovoľný počet iných vrcholov. Pri binárnom strome môže ukazovať na najviac dva iné vrcholy, pričom rozlišujeme medzi pravým a ľavým synom. V každom vrchole navyše môže byť uchovaná nejaká hodnota.

Teraz by sme chceli použiť binárny strom na ukladanie prvkov. Môžeme samozrejme do každého vrcholu umiestniť jeden prvok, ale aby to nestratilo význam a nestal sa z toho len divný spájaný zoznam, potrebujeme to spraviť nejak systematicky. Najčastejšie sa binárne vyhľadávacie stromy používajú na ukladanie čísiel, vysvetlíme si to teda na nich. Prvky budeme ukladať tak, že pre každý vrchol $v$ platí, že v jeho pravom synovi a v jeho podstrome (vrcholy pod ním) obsahujú prvky väčšie ako prvok vo $v$ a v ľavom synovi a v jeho podstrome sú prvky menšie ako prvok vo $v$.

Poďme sa teraz pozrieť na to, ako vieme do takéhoto stromu prvky pridávať, odoberať a zisťovať, či sa v ňom nachádzajú. Posledná operácia je najľahšia. Chceme zistiť či sa prvok $x$ nachádza v našom strome (teda v našej množine čísiel). Začnime tak, že sa pozrieme na prvok v koreni. Ak je to $x$, tak môžeme skončiť a povedať „áno“. Ak je $x$ menšie ako číslo v koreni, tak vieme, že $x$ sa môže nachádzať len naľavo od koreňa. V opačnom prípade sa $x$ môže nachádzať len v pravom podstrome. Tento postup opakujeme na vhodnom podstrome až dovtedy, kým $x$ nenájdeme (alebo nám zostane prázdny podstrom – vtedy sa $x$ v našej množine nenachádza).

Pridanie prvku je veľmi podobná operácia. Postupom popísaným vyššie buď zistíme, že $x$ sa už nachádza v strome, alebo nájdeme miesto, kde by sme $x$ očakávali, ale už tam nič nie je. Toto bude teda miesto, kam vložíme nový vrchol.

Najzložitejšie je odstrániť prvok zo stromu. Toto si treba rozdeliť na tri prípady.

1. Vrchol, ktorý chceme odstrániť, je list. Nič nám v tom nebráni, takže ho odstránime.
2. Vrchol $v$ má jedného syna $u$. Odstránime vrchol $v$ a vrchol $u$ bude odteraz synom otca vrcholu $v$.
3. Vrchol $v$ má oboch synov – ľavého $l$ a pravého $p$. V pravom podstrome nájdeme najmenší prvok $q$ (stačí, že od vrcholu $p$ pôjdeme doľava, kým sa bude dať). Prvok $q$ odstránime podľa druhého pravidla. Vrchol $v$ nahradíme vrcholom $q$.

Teraz už vieme robiť všetky tri operácie, ktoré vyžaduje naša množina. Otázkou zostáva, ako je to efektívne. Pri každej operácii ideme z koreňa stále dodola, to znamená, že najviac navštívime $h$ vrcholov, kde $h$ je hĺbka stromu (t.j. najdlhšia cesta z koreňa k nejakému listu). Každú operáciu teraz vieme vykonať v čase $O(h)$. Akú hĺbku však môže mať binárny strom s $n$ vrcholmi? Bohužiaľ, môže to byť čokoľvek medzi $\log n$ a $n$. Síce vykonávať tieto operácie v $O(\log n)$ by bolo super, $O(n)$ je už priveľa. Existujú síce spôsoby, ktoré zaručia vykonávanie týchto operácii v $O(\log n)$, avšak ich implementácia je trošku náročnejšia. Našťastie C++ to robí za nás.

### Set

[3]Prvky sa budú môcť v našich množinách opakovať. V skutočnosti teda pôjde o multimnožiny.

V C++ je už naimplementovaný takýto vyvažovaný vyhľadávací strom (t.j. jeho hĺbka je vždy $O(\log n)$) s názvom `set` a jeho použitie je veľmi jednoduché. Ukážme si teda základné funkcie, ktoré táto štruktúra pozná.

### Listing programu (C++)

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

int main() {
  set<int> S;
  S.insert(2); S.insert(10);
  S.insert(-5); S.insert(47);
  int a=S.size();
  set<int>::iterator it=S.begin();
  for(;it!=S.end(); it++) {cout << *it << endl;}
  if(S.find(7)==S.end()) cout << "Prvok sa v sete nenachadza.\n";
  else cout << "Prvok tu je.\n";
  S.erase(47);
  S.erase(S.begin());
  S.erase(S.begin(),S.find(10));
  S.lower_bound(15);
  S.upper_bound(1);
  multiset<int> M;
  M.insert(2); M.insert(2);
  M.insert(15); M.insert(15);
  M.erase(2);
  M.erase(M.find(15));
}
```

Tento kúsok kódu je zároveň rýchlym prehľadom najpoužívanejších funkcií. Samozrejme, všetko sa dá vygoogliť, ale rád by som zvlášť upozornil na pár zákerností. Prvým pár riadkom by nemal byť problém porozumieť. Najskôr vyberieme správnu knižnicu, ktorá našu štruktúru implementuje, v tomto prípade `<set>`. V prvom riadku funkcie `main` si potom zadefinujeme, ako sa bude náš `set` volať a aké hodnoty bude obsahovať (tu sa medze nekladú, zbehne všetko cez `int`y až po ďalšie `set`y – naozaj mať v kóde `set<set<int> >` je oveľa častejšie, ako by ste si mysleli). Následne funkcia `insert()` doňho vkladá prvky a funkcia `size()` vracia počet prvkov, ktoré sa v ňom nachádzajú.

Zaujímavé to však začína byť v riadku s magickým slovom `iterator`. Iterátor je taká magická šípka, ktorá ukazuje na nejaký prvok `set`u. Táto šípka sa dá aj posúvať, ale len pomocou inkrementátora `++` a dekrementátora `--`. Také základné iterátory, ktoré STL implementuje, sú `begin()` a `end()`. Funkcia `begin()` vracia iterátor na začiatok `set`u, teda na jeho prvý prvok. Funkcia `end()` vracia iterátor na koniec `set`u, teda **za** posledný prvok, preto si dajte pozor, aby ste sa nepýtali na jeho hodnotu (v lepšom prípade vám len vráti hlúposť).

Prvky v `set`e sú uskladnené utriedené v poradí, preto cyklus v tomto programe vypíše prvky, ktoré som do setu vložil utriedené od najmenšieho po najväčší. Všimnite si, že k hodnote prvku, na ktorý ukazuje daný iterátor, sa pristupuje cez symbol `*`. Za zmienku tiež stojí, že iterátory sa dajú porovnávať, pričom sa neporovnáva hodnota, na ktorú ukazujú, ale miesto v pamäti, kam smerujú. Preto funguje v cykle podmienka `it!=S.end()`. Iterátor sa bude zvyšovať, až kým nedosiahne koniec `set`u.

Ďalšie funkcie sú o dosť využívanejšie. Funkcia `find(x)` vracia iterátor na prvok s hodnotou `x`. Ak sa taká hodnota v `set`e nenachádza, vráti iterátor na koniec, teda `end()`. Príkaz `erase()` slúži na mazanie prvkov. Ako parameter si berie buď hodnotu, ktorú má zmazať, alebo iterátor na prvok, ktorý má zmazať. Keď dostane `erase()` ako parametre dva iterátory, vymaže všetky prvky od prvého iterátora vrátane až po druhý iterátor (ten však nezmaže). Funkcia `lower_bound(x)` vracia iterátor na prvý prvok s hodnotou rovnou alebo väčšou ako hodnota `x`; funkcia `upper_bound(x)` na prvý prvok s hodnotou väčšou ako `x`.

Ako taký ma však `set` jedno obmedzenie – každú hodnota sa v ňom môže nachádzať najviac raz. Ak by mal do seba vložiť hodnotu, ktorá sa v ňom nachádza, jednoducho toto vloženie ignoruje. V našej úlohe však potrebujeme mať možnosť uskladniť aj viacero rovnakých hodnôt, nikde totiž nie je povedané, že sa to nestane. Preto musíme použiť rozšírenie a tým je `multiset<>`. `multiset` je implementovaný v rovnakej knižnici ako `set`, používa tie isté funkcie, akurát vie v sebe obsahovať viacej rovnakých hodnôt. Existuje však jeden detail, na ktorý si musíte pri programovaní dať pozor. A to na funkciu `erase()`. Keď dostane `erase` ako vstupný parameter konkrétnu hodnotu `x`, vymaže **všetky** prvky s touto hodnotou. Avšak, keď jej dáte ako parameter iterátor, vymaže len jeden konkrétny prvok. Preto na vymazanie len jedného prvku nejakej hodnoty, použite najskôr funkciu `find()`, ktorá vám vráti iterátor len na jeden prvok.

Zaujímavou otázkou je, ako rýchlo sú tieto funkcie vykonávané. Vďaka tomu, že `set` je implementovaný ako vyvažovaný binárny strom, jeho hĺbka sa priveľmi nelíši od logaritmu počtu prvkov, preto sú funkcie `find()`, `erase()`, `insert()`, `lower_bound()` a `upper_bound()` spracované s časovou zložitosťou $O(\log n)$.

### Záver

Teraz by sme už mali presne vedieť, aké dátové štruktúry chceme použiť a aj to, ako ich použiť. Poslednou otvorenou otázkou je časová a pamäťová zložitosť. Pamäťová bude $O(n)$, keďže v najhoršom prípade (keď $\ell = n$)

si musíme pamätať všetky informácie zo vstupu. Časová zložitosť je $O(n \log n)$, lebo každý prvok raz vložíme a raz vyberieme z našich `set`ov, čo zaberie konštantný počet `set`ových operácií, z ktorých každá trvá $O(\log n)$.

Týmto vzorák končí, ja sa lúčim a vy, ktorí ste túto úlohu nevyriešili počas série, ste si snáď odniesli nejaké nové informácie a môžete si skúsiť túto úlohu nesúťažne naprogramovať.

## Listing programu (C++)

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

typedef long long ll;

int main() {
  int n,len;
  scanf("%d %d",&n,&len);
  vector<int> A; A.resize(n+1);
  A[0]=0;
  for(int i=0; i<n; i++) {
    scanf("%d",&A[i+1]);
  }
  int k;
  scanf("%d",&k);
  multiset<int> Z,K;
  multiset<int> Z1,K1;
  //napln prvy krat
  ll suc=0;
  for(int i=0; i<len; i++) {
    if(A[i]<0) Z1.insert(A[i]);
    else K1.insert(-A[i]);
    suc+=A[i];
  }
  ll sucz=0,suck=0;
  while(Z1.size()!=0 && Z.size()!=k) {sucz-=*Z1.begin(); Z.insert(-(*Z1.begin())); Z1.erase(Z1.begin());}
  while(K1.size()!=0 && K.size()!=k) {suck-=*K1.begin(); K.insert(-(*K1.begin())); K1.erase(K1.begin());}
  //prechadzaj cez prvky
  ll vys=0ll;
  for(int i=len; i<=n; i++) {
    //odstranim posledny
    if(A[i-len]<0) {
      int x=A[i-len];
      suc-=x;
      if(Z1.find(x)!=Z1.end()) {
        Z1.erase(Z1.find(x));
      } else {
        sucz+=x;
        Z.erase(Z.find(-x));
        while(Z1.size()!=0 && Z.size()!=k) {
          sucz-=*Z1.begin();
          Z.insert(-(*Z1.begin()));
          Z1.erase(Z1.begin());
        }
      }
    }
    else {
      int x=A[i-len];
      suc-=x;
      if(K1.find(-x)!=K1.end()) {
        K1.erase(K1.find(-x));
      } else {
        suck-=x;
        K.erase(K.find(x));
        while(K1.size()!=0 && K.size()!=k) {
          suck-=*K1.begin();
          K.insert(-(*K1.begin()));
          K1.erase(K1.begin());
        }
      }
    }
    //pridam novy
    if(A[i]<0) Z1.insert(A[i]);
    else K1.insert(-A[i]);
    suc+=A[i];
    if(Z.size()!=0) {
      Z1.insert(-(*Z.begin()));
      sucz-=*Z.begin();
      Z.erase(Z.begin());
    }
    if(K.size()!=0) {
      K1.insert(-(*K.begin()));
      suck-=*K.begin();
      K.erase(K.begin());
    }
    while(Z1.size()!=0 && Z.size()!=k) {
      sucz-=*Z1.begin();
      Z.insert(-(*Z1.begin()));
      Z1.erase(Z1.begin());
    }
    while(K1.size()!=0 && K.size()!=k) {
      suck-=*K1.begin();
      K.insert(-(*K1.begin()));
```

```
      K1.erase(K1.begin());
    }
    vys=max(vys,max(abs(suc+2*sucz),abs(suc-2*suck)));
  }
  cout << vys << endl;
  return 0;
}
```

vzorák napísal Usámec

## 7. Obodované preteky mravcov

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

Keď sa poriadne zamyslíme nad tým, ako sa bodujú jednotlivé cesty mravcov, tak zistíme, že zadanie chce, aby sme našli $k$-tu lexikograficky najmenšiu cestu z jedného rohu do druhého.

Myšlienka vzorového riešenia je pomerne jednoduchá. Pre každé políčko $(i, j)$ si najprv predpočítame hodnotu $C[i, j]$, ktorá nám hovorí, koľko ciest vedie z tohoto políčka do cieľa (pravého dolného rohu). Ako túto hodnotu spočítať si ukážeme neskôr. S použitím týchto hodnôt vieme našu cestu generovať pomerne priamočiaro. Nech hľadané poradie cesty je $p$ a stojíme na políčku $(y, x)$. Skúsime sa pohnúť dole a pozrieme sa na hodnotu $C[y + 1, x]$. Ak $C[y + 1, x] \geq p$, tak týmto smerom vedie dostatočne veľa ciest a pohneme sa smerom nadol. Ak naopak $C[y + 1, x] < p$, tak ciest smerom dolu je málo a musíme sa pohnúť doprava. Zároveň po tomto pohybe si znížime hodnotu $p$ o $C[y + 1, x]$ (lebo toľkoto ciest viedlo smerom nadol, kde sme nešli). Týmto spôsobom vieme v čase $O(r + c)$ nájsť jednu cestu. A všetky cesty v čase $O(q \cdot (r + c))$.

Ako spočítať hodnoty $C[i, j]$? Toto bude celkom jednoduchá dynamika. Je jasné, že $C[r - 1, c - 1] = 1$ (lebo z konca do konca máme práve jednu cestu). Zároveň pre všetky $y \geq r$ a $x \geq c$ platí $C[y, j] = C[x, i] = 0$ (spoza okraja máme 0 ciest do cieľa). Ak na políčku $(i, j)$ je prekážka, tak $C[i, j] = 0$. Všeobecnú hodnotu $C[i, j]$ spočítame veľmi jednoducho. Z daného políčka sa môžeme pohnúť buď dole alebo vpravo a teda $C[i, j] = C[i + 1, j] + C[i, j + 1]$. Toto vieme dvoma cyklami v sebe celé spočítať v čase $O(rc)$.

Ešte je tu jeden problém. Čísla v matici $C$ môžu byť celkom veľké. Na druhej strane položené otázky obsahujú ešte pomerne malé čísla. Z toho nám vyplýva, že si nemusíme pamätať konkrétne hodnoty v matici, pokiaľ sú dané čísla príliš veľké. Jednoducho ak hodnota spočítaného čísla prekročí istú hranicu (medzi $10^{17}$ a tým, čo znesie 64-bitové číslo, je stále veľká medzera), tak si poznačíme, že toto číslo je príliš veľké (potom treba nezabudnúť, že veľké číslo + hocijaké číslo je stále veľké číslo). V rozumných programovacích jazykoch sa toto rieši tak, že si vytvoríme vlastný typ a dodefinujeme mu potrebné operátory.

### Listing programu (C++)

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

using namespace std;

#define BIG 2000000000000000001ll

class bint {
 public:
  long long num;
  bool big;

  bint(long long x) : num(x), big(false) {
    Check();
  }

  bint() : num(0), big(false) {
  }

  void Check() {
    if (num > BIG) big = true;
  }

  bint operator=(long long x) {
    num = x;
    Check();
    return *this;
  }
};

bint operator+(const bint& a, const bint& b) {
  if (a.big) return bint(BIG+1);
  if (b.big) return bint(BIG+1);
  return bint(a.num + b.num);
}

bint operator-(const bint& a, const bint& b) {
  if (a.big) return bint(BIG+1);
  if (b.big) return bint(BIG+1);
  return bint(a.num - b.num);
}

bool operator<(const bint& a, const bint& b) {
```

```
  if (a.big) return false;
  if (b.big) return true;
  return a.num < b.num;
}

int mat[1010][1010];
bint dyn[1010][1010];
bint q[1010];
int R, C, Q;

int main() {
  scanf("%d %d %d", &R, &C, &Q);
  for (int i = 0; i < R; i++)
    for (int j = 0; j < C; j++)
      scanf("%d", &mat[i][j]);
  for (int i = 0; i < Q; i++) {
    long long a;
    scanf("%lld", &a);
    q[i] = a;
  }

  dyn[R-1][C-1] = 1;
  for (int i = R-1; i >= 0; i--) {
    for (int j = C-1; j >= 0; j--) {
      if (i == R-1 && j == C-1) continue;
      if (mat[i][j] == 1) continue;
      dyn[i][j] = dyn[i+1][j] + dyn[i][j+1];
    }
  }

  for (int i = 0; i < Q; i++) {
    if (dyn[0][0] < q[i]) {
      printf("neexistuje\n");
      continue;
    }
    int x = 0, y = 0;
    while (x != C-1 || y != R-1) {
      if (dyn[y+1][x] < q[i]) {
        q[i] = q[i] - dyn[y+1][x];
        x++;
        printf("P");
      } else {
        y++;
        printf("D");
      }
    }
    printf("\n");
  }
}
```

vzorák napísal Bob

## 8. Obdĺžnikové pečiatky

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

Hrozí, že v tomto vzoráku budem vo veľkej miere používať písmenká sádzané kurzívou a inú chrobač, preto sa zíde pripraviť sa na to hneď na začiatku.

Dĺžku reťazca $x$ budem označovať tradične $|x|$, zreťazenie $x$ a $y$ zase $xy$, prázdny reťazec bude $\varepsilon$. Jednotlivé znaky v reťazci budem číslovať od 1. Pre odlíšenie, konkrétne hodnoty písmen (literály) budem označovať monospaced fontom. Takže napríklad: $|\varepsilon\varepsilon| = |\varepsilon| = 0$, $x = \texttt{ab}$, $y = \texttt{c}$, $xy = \texttt{abc}$, $|xy| = 3$, $z = a_1 a_2 \ldots a_{|z|}$.

Text zadaný na vstupe označím $t$ a pečiatky $p_1, p_2, \ldots, p_n$. Pri odhadoch zložitosti budem často potrebovať nejako pomenovať dĺžku najdlhšej pečiatky. Použijem na to zápis $|p|$, ktorý je intuitívny, no nekonzistentný so zvyškom textu, lebo $p$ sa mi určite neskôr podarí zadefinovať inak. Dĺžka vstupu je potom $O(|t| + n \cdot |p|)$.

### Uchopenie problému

Pečiatky môžeme použiť koľkokrát chceme – optimálnym riešením je teda každú pečiatku odtlačiť na všetky jej výskyty v texte. Hotovo. Prečo bola potom táto úloha až osmička?

Všetkých výskytov $i$-tej pečiatky v texte môže byť totiž až $|t| - |p_i| + 1$, čo vzhľadom na limity v zadaní prepíšeme ako $O(|t|)$. V súčte pre všetky pečiatky môže byť výskytov až $O(n \cdot |t|)$ – a to ich ešte musíme nájsť a spracovať.

Takže sme nútení niektoré výskyty pečiatok v texte ignorovať. Nevadí, očividne nám stačí zapodievať sa iba tými najdlhšími. Presnejšie, pre každú pozíciu textu nájdeme najdlhšiu pečiatku, ktorej výskyt začína na tejto pozícii, a odtlačíme ju (ak existuje). Všetky kratšie pečiatky, ktoré by tam tiež sedeli, sú už teraz zbytočné – tá najdlhšia pečiatka ich celé prekryje.[4]

Takto sme si zmenšili počet odtlačkov na $O(|t|)$. Ich hľadaniu sa budeme venovať neskôr, teraz sa pozrime na to, ako z nich vypočítať výsledok našej úlohy.

[4] Rovnako dobre by sme mohli hľadať najdlhšie pečiatky, ktoré *končia* na jednotlivých pozíciách. Všetky postupy popísané v nasledujúcich odsekoch sa dajú jednoducho upraviť tak, aby fungovali pre takto otočenú formuláciu. Vo vzorovom programe budeme využívať práve tento variant (najdlhšia pečiatka končiaca na danej pozícii); neskôr uvidíme prečo.

### Zametanie

Označme $\ell_i$ dĺžku najdlhšej pečiatky, ktorá má výskyt začínajúci na $i$-tej pozícii $t$. V prípade, že taká pečiatka neexistuje, bude $\ell_i$ rovné 0. Chceme zistiť, koľko písmen textu je nepokrytých týmito najdlhšími pečiatkami.

Pozícia $i$ môže byť pokrytá jedine pečiatkami, ktoré začínajú naľavo od nej, prípadne priamo na nej. Najdlhšia pečiatka so začiatkom $j$ siaha až po $j + \ell_j - 1$, teda aby pokryla aj pozíciu $i$, musí platiť $j \leq i < j + \ell_j$. Takže nám stačí vypočítať $\max(1 + \ell_1, 2 + \ell_2, \ldots, i + \ell_i)$ a porovnať ho s $i$. Takto zistíme, či je $i$-te písmeno pokryté aspoň jednou pečiatkou.

Samozrejme, toto maximum nemusíme pre každé $i$ rátať vždy odznova. Budeme postupovať zľava doprava: vždy, keď pridáme novú pečiatku, maximum prepočítame v konštantnom čase. Môžeme si to tiež predstaviť ako zametanie intervalov (pečiatok), pričom maximum zodpovedá najbližšej pozícii, ktorá zatiaľ nie je pokrytá.

Dokopy nám táto časť riešenia zaberie $O(|t|)$ času.

### Ako hľadať výskyty pečiatok

Začnime s jednoduchým, ale pomalým algoritmom: Pre každú pozíciu $i$ a pečiatku $p_j$ overíme, či $p_j$ má výskyt začínajúci na $i$-tej pozícii textu. Ako? Lineárnym prechodom cez všetky písmená $p_j$. Toto môže v najhoršom prípade trvať až $O(n \cdot |t| \cdot |p|)$.

Aby sme vedeli overovať výskyt všetkých pečiatok naraz, použijeme písmenkový strom. Najprv do neho postupne vložíme každú z pečiatok, to zaberie $O(n \cdot |p|)$ času. Potom pre danú štartovaciu pozíciu $i$ nájdeme najdlhšiu vyhovujúcu pečiatku tak, že v písmenkovom strome budeme prechádzať z koreňa hranami zodpovedajúcimi $i$-temu, $(i+1)$-vému, $(i+2)$-hému, … písmenu textu, kým potrebné hrany existujú. Posledný vrchol na našej ceste, ktorý bol označený ako koniec nejakej pečiatky v písmenkovom strome, predstavuje práve hľadanú najdlhšiu pečiatku. Táto časť bude spolu pre všetky pozície trvať $O(|t| \cdot |p|)$.

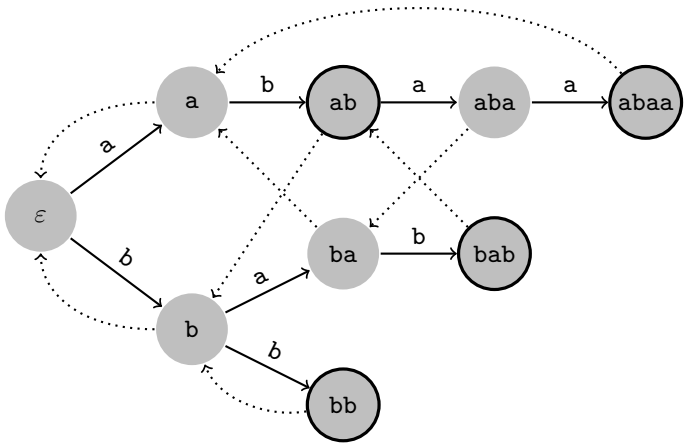
Máme riešenie s časovou zložitosťou $O(n \cdot |p| + |t| \cdot |p|)$. Pomaly sa približujeme k lineárnej zložitosti od veľkosti vstupu, na to však potrebujeme efektívnejšie využívať už získanú informáciu.

### Vytúnený písmenkáč

Povedzme, že prišiel dobrý ujo Xzibit a pridal do nášho písmenkového stromu sufixové linky. Čo to je a prečo sme mu vďační?

Každý vrchol stromu zodpovedá reťazcu, ktorý dostaneme čítaním písmen na ceste z koreňa stromu do toho vrcholu. Reťazce, ktoré majú takýto zodpovedajúci vrchol v strome, budeme volať *notorické*. Majme neprázdny notorický reťazec $x$. Najdlhší vlastný sufix[5] reťazca $x$, ktorý je tiež notorický, označme $f(x)$. Sufixová linka z vrcholu $x$ bude viesť práve do vrcholu $f(x)$.

Ukážme si to na príklade. Na nasledujúcom obrázku môžete vidieť písmenkový strom pre reťazce `ab`, `abaa`, `bab` a `bb`. Sufixové linky sú znázornené bodkovanými čiarami. Zvýraznené vrcholy zodpovedajú slovám, z ktorých sme strom budovali.




Tak si napríklad všimnime, že sufixová linka z vrcholu `abaa` vedie až do `a`, pretože pre dlhšie sufixy (`baa`, `aa`) v strome vrcholy nemáme. Taktiež je dobré uvedomiť si, že presúvaním sa po sufixových linkách postupne navštívime všetky notorické sufixy daného reťazca (napríklad `aba` → `ba` → `a` → $\varepsilon$).

[5] Každý reťazec je sám sebe sufixom. Nás však zaujímajú iba také sufixy, ktoré sú kratšie ako pôvodný reťazec.

Sufixové linky sú užitočné na hľadanie najhlbšieho vrcholu stromu,[6] ktorého reťazec končí na danej pozícii textu. Skúsme si to napríklad na texte `ababb`. Zo začiatku je to jednoduché: na prvej pozícii končí reťazec `a`, na druhej `ab`, na tretej `aba`. Všimnite si, že sme sa len pohodlne presúvali po vrcholoch stromu, stále idúc po hrane označenej nasledujúcim písmenom textu.

Vo vrchole `aba` však nastal problém: na štvrtej pozícii sa nachádza písmeno `b`, no takto označená hrana z `aba` nevedie. Treba cúvnuť – namiesto najhlbšieho vrcholu pre tretiu pozíciu textu (`aba`) skúsme zobrať druhý najhlbší vrchol ($f($`aba`$) =$ `ba`). A naozaj, z `ba` vedie hrana označená písmenom `b` do vrcholu `bab`. Teda najhlbším vrcholom s reťazcom končiacim na štvrtej pozícii textu je `bab`.

Na piatej pozícii textu nás čaká písmeno `b` a podobný problém. Keďže z aktuálneho vrcholu `bab` nevedie hrana na `b`, skúsime cúvnuť do $f($`bab`$) =$ `ab`. V ňom znova zlyhávame, cúvame preto až do $f($`ab`$) =$ `b`, kde konečne nachádzame správne označenú hranu a postupujeme ňou do vrcholu `bb`.

Skúsme náš postup zovšeobecniť a popísať formálnejšie. Označme $x_i$ najdlhší notorický reťazec, ktorý končí na $i$-tej pozícii textu. Písmená textu označme $c_1, c_2, \ldots, c_{|t|}$. Nech už poznáme $x_i$ a hľadáme, čomu sa rovná $x_{i+1}$.

Každý notorický reťazec končiaci na pozícii $i+1$ vieme skonštruovať tak, že vezmeme vhodný notorický reťazec končiaci na pozícii $i$ a pripojíme za neho písmeno $c_{i+1}$. Inými slovami: aj keď vymažeme posledný znak notorického reťazca, musíme dostať notorický reťazec. Jedinou výnimkou je $\varepsilon$, ten budeme musieť vyriešiť osobitne.

Aby sme našli $x_{i+1}$, budeme postupne skúšať predĺžiť písmenom $c_{i+1}$ každý z notorických reťazcov končiacich na $i$-tej pozícii. Začneme najdlhším ($x_i$) a v prípade neúspechu budeme postupovať k čoraz kratším ($f(x_i)$, $f(f(x_i))$, …), až kým neprídeme k $\varepsilon$. Len čo narazíme na taký reťazec $y$, že $yc_{i+1}$ je notorický, zastavíme hľadanie a prehlásime $x_{i+1} = yc_{i+1}$. Ak sa ani $\varepsilon c_{i+1} = c_{i+1}$ nenachádza v strome, potom $x_{i+1} = \varepsilon$.

Pozrime sa na časovú zložitosť hľadania $x_{i+1}$. Na prvý pohľad nám sufixové linky veľmi nepomohli – čo ak budeme musieť vyskúšať postupne všetky notorické sufixy $x_i$ (teda celú postupnosť $x_i, f(x_i), f(f(x_i)), \ldots, \varepsilon$)? V najhoršom prípade takto vykonáme $O(|p|)$ operácii, keďže hĺbka celého stromu je práve $|p|$ a každým prechodom cez sufixovú linku sa priblížime ku koreňu. V skutočnosti však takýchto „drahých“ hľadaní nemôže byť veľa. Dokopy totiž prejdeme sufixovými linkami v strome nahor najviac toľkokrát, koľkokrát prejdeme obyčajnými hranami nadol, a to sa rovná $|t|$. Vypočítať všetky $x_i$ nám preto spolu potrvá $O(|t|)$.

Hurá, máme efektívnu hračku, ktorou vieme pre každú pozíciu v texte určiť, aký najdlhší notorický reťazec sa tam končí. My by sme na vyriešenie pôvodnej úlohy potrebovali niečo podobné, akurát nás zaujímajú len tie reťazce, ktoré zodpovedajú pečiatkam – nie všetky ich prefixy.

To je už ľahké zistiť: Stačí sa z daného vrcholu pustiť sufixovými linkami, kým nenarazíme na taký vrchol, ktorý zodpovedá nejakej pečiatke. Zapíšeme si jeho hĺbku (teda dĺžku nájdenej pečiatky); prípadne 0, ak sme došli až do $\varepsilon$. Túto hodnotu pre vrchol $x$ si označme $\ell(x)$. Ak existuje pečiatka s textom $x$, potom $\ell(x) = |x|$; v opačnom prípade $\ell(x) = \ell(f(x))$. Tento vzťah nám umožňuje predpočítať si všetky hodnoty $\ell(x)$ postupne od koreňa k hlbším vrcholom v čase lineárnom od veľkosti stromu, teda $O(n \cdot |p|)$.

### Už len nájsť tie sufixové linky

Tak, tak. Nebudeme čakať na uja Xzibita, ale sufixové linky si do písmenkáča pridáme sami. Tupým prepísaním definície by sme vedeli nájsť $f(x)$ pre daný vrchol $x$ v čase $O(|x|^2)$ – pre čoraz kratšie sufixy $x$ by sme overovali, či sú notorické. To je zrejme príliš pomalé, skúsme preto aj teraz použiť rovnakú myšlienku ako pri hľadaní hodnôt $x_i$.

Majme vrchol $x$, z ktorého vedie hrana označená písmenom $c$ do vrcholu $xc$. Budeme hľadať najdlhší vlastný notorický sufix reťazca $xc$. Zjavne každý taký sufix sa dá skonštruovať tak, že za vhodný vlastný notorický sufix reťazca $x$ pridáme písmeno $c$. Budeme teda prechádzať postupne vrcholy $f(x), f(f(x)), \ldots, \varepsilon$ a snažiť sa presunúť sa z nich po hrane označenej $c$. Prvý vrchol, v ktorom sa nám to podarí, označme $y$ – potom bude platiť $f(xc) = yc$. V prípade, že ani z $\varepsilon$ nevedie hrana na $c$, máme $f(xc) = \varepsilon$.

Hodnoty $f(x)$ budeme počítať postupne od koreňa po vrstvách tak, aby v momente počítania $f(x)$ už boli známe hodnoty $f(y)$ pre všetky $y$ kratšie ako $x$. Na to nám poslúži prehľadávanie stromu do šírky spustené z koreňa.

S odhadom časovej zložitosti to teraz máme o trochu veselšie ako naposledy. Už nemôžeme jednoducho argumentovať, že „drahých“ výpočtov $f(x)$ bude v strome málo: Hoci bude $|f(xc)|$ len o jednotku viac ako $|f(x)|$, o túto jednotku potom možno cúvnuť v každom potomkovi vrcholu $xc$ v strome (a tých bude potenciálne vďaka

[6] Hĺbku vrcholu definujeme ako jeho vzdialenosť od koreňa. Vzhľadom na náš obrázok by možno bolo vhodnejším pomenovaním *najpravejší* vrchol stromu.

vetveniu veľa). Namiesto času lineárneho od veľkosti stromu skúsme preto dokázať časovú zložitosť lineárnu od súčtu dĺžok pečiatok (čo je stále $O(n \cdot |p|)$).

Každý presun po sufixovej linke pri výpočte $f(x)$ zaúčtujeme niektorej z pečiatok, ktoré začínajú na $x$ (určite aspoň jedna taká existuje, prečo inak by sme v strome mali vrchol $x$). No a teraz už len vytasíme starý argument: Jednej pečiatke sme mohli dokopy zaúčtovať najviac toľko pohybov po sufixovej linke, koľkokrát sme na jej ceste postúpili po obyčajnej hrane, čo je rovné dĺžke tejto pečiatky.

### Záver

Asi by už bolo načase prezradiť vám, že práve popísaný vytúnený písmenkáč som si nevymyslel ja ani Xzibit, ale pochádza od uja Aha a tety Corasickovej.[7] Ak chcete ďalšie informácie, hľadajte „Aho–Corasick string matching algorithm“. Tento algoritmus sa využíva na hľadanie viacerých vzoriek v texte – čo sme vlastne robili aj my v tejto úlohe.

Malá poznámka k implementácii: Na našej testovacej mašine pointre zaberajú 8 bytov, kvôli čomu môj vzorový program, ktorý narába s pamäťou veľmi rozšafne, nedostane plný počet bodov. Dá sa však upraviť (čítajte: zneprehľadniť) tak, aby si vrcholy stromu ukladal v poli, namiesto pointrov používal 4-bytové indexy do toho poľa, a aby nemíňal pamäť odkazmi na neexistujúcich synov vrcholov. Vhodná kombinácia týchto optimalizácii priniesla aj mne 15 bodov. Naozaj.

### Listing programu (C++)

```
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;

#define MAXT 300007
#define MAXP 5007
#define SIGMA 26

char text[MAXT], pattern[MAXP];
int L[MAXT];  // dĺžka najdlhšej pečiatky končiacej na danej pozícii textu

struct Node {
  Node* next[SIGMA];  // hrany písmenkáča
  Node* fail;         // sufixová linka
  int longest;        // dĺžka najdlhšej pečiatky, ktorá je sufixom tohto vrcholu

  Node() {
    for (int i = 0; i < SIGMA; ++i)
      next[i] = NULL;
    fail = NULL;
    longest = 0;
  }
};

Node root, earth;  // root je koreň stromu, earth je "nadkoreň" -- uľahčí to spracovanie epsilonu

void insert(char* s) {  // pridá reťazec s do písmenkového stromu
  Node* x = &root;
  int length = 0;
  while (*s) {
    if (!x->next[*s - 'a'])
      x->next[*s - 'a'] = new Node();
    x = x->next[*s - 'a'];
    ++s;
    ++length;
  }
  x->longest = length;
}

void build() {  // nájde sufixové linky
  root.fail = &earth;
  for (int i = 0; i < SIGMA; ++i)
    earth.next[i] = &root;
  queue<Node*> Q;
  Q.push(&root);
  while (!Q.empty()) {
    Node* x = Q.front(); Q.pop();
    for (int i = 0; i < SIGMA; ++i)
      if (x->next[i]) {
        Node* y = x->fail;
        while (!y->next[i])
          y = y->fail;
        x->next[i]->fail = y->next[i];
        if (!x->next[i]->longest)
          x->next[i]->longest = x->next[i]->fail->longest;
        Q.push(x->next[i]);
      }
  }

}
```

[7] To, že je teta Corasicková naozaj teta a nie ujo, viem len na základe ústneho podania, takže to berte s rezervou. Áno, volá sa Margaret... ale druhým menom John!

```
int main() {
  scanf("%s", text);
  int n, Tsize = strlen(text);
  scanf("%d", &n);
  for (int i = 0; i < n; ++i) {
    scanf("%s", pattern);
    insert(pattern);
  }
  build();

  Node* x = &root;
  for (int i = 0; i < Tsize; ++i) {
    while (!x->next[text[i] - 'a'])
      x = x->fail;
    x = x->next[text[i] - 'a'];
    L[i] = x->longest;
  }

  int b = Tsize - 1, res = 0;
  for (int i = Tsize - 1; i >= 0; --i) {
    b = min(b, i - L[i]);
    res += (b == i);
  }
  printf("%d\n", res);
}
```


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