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

Vzorové riešenia 2. kola zimnej časti

vzorák napísal Luxusko
(max. 7 b za popis, 3 b za program)

1. Zbedačený les

Zamyslime sa, ako určiť staršie z dvoch zvieratiek. Ak sú rovnakého druhu, je to ľahké – dlhšie žijúce je tučnejšie. Čo ale so zvieratkami rôznych druhov?

Správna odpoveď je: nedá sa určiť, najstaršie môžu byť obe – záleží to len od rýchlostí, akými tučnejú druhy našich zvieratiek, a tieto nepoznáme.

Teraz vidíme, že naši kandidáti budú najstarší zástupcovia každého druhu v lese. Kľúčové je uvedomiť si, že sa pýtame len na ich počet, nie na to, ktorí presne to sú. Ak existujú v lese nejaké zvieratká druhu X, jedno z nich musí byť najstaršie, a to jedno patrí medzi našich kandidátov. Pre správne vyriešenie úlohy teda potrebujeme vedieť povedať, **koľko rôznych druhov naozaj žije v lese**.

Šikovne využijeme, že druhy dostaneme očíslované od 1 po n a že n je najviac 10^6. Pre každý druh si poznačíme, či žije aspoň jedno zvieratko daného druhu v lese. Ľahko to zrealizujeme obyčajným poľom s aspoň n chlievikmi. Na začiatku si do každého chlievika poznačíme, že žiadne zvieratká takého druhu zatiaľ nemáme, a pri prechádzaní vstupnými údajmi budeme tieto hodnoty aktualizovať.

Dobré je sústrediť sa na to, čo potrebujeme zistiť, nepočítať nič navyše a nerobiť jednoduché veci komplikovane.

Časová aj pamäťová zložitosť nášho prístupu je lineárna od počtu zvieratiek, $O(n)$ – celý vstup raz prebehneme a potom ešte raz skontrolujeme, ktoré druhy v lese máme, a použijeme pri tom pár premenných a pole s chlievikom pre každý druh.

Listing programu (Pascal)

```
var n, druh, tucnota, i, druhov: longint;
    mame: array[1..1000000] of boolean;

begin
  ReadLn(n);
  for i := 1 to n do mame[i] := false;
  for i := 1 to n do begin
    ReadLn(druh, tucnota);
    mame[druh] := true;
  end;
  druhov := 0;
  for i := 1 to n do if mame[i] then Inc(druhov);
  WriteLn(druhov);
end.
```

vzorák napísala Maja
(max. 7 b za popis, 3 b za program)

2. Zradný rad na obed

Táto úloha sa dala riešiť viacerými spôsobmi. Priamočiare riešenie by vyzeralo asi takto: načítame si celý vstup do poľa veľkosti n a zapamätáme si, na ktorom mieste sa v tomto poli nachádza Luxusko. Potom si pole usporiadame a pozrieme sa, na ktorom mieste sa Luxusko nachádza teraz. Potom už len porovnáme tieto dve hodnoty a vieme, kam a o koľko sa Luxusko musí posunúť.

Na usporiadanie poľa sa dalo použiť rôzne efektívne algoritmy. Tie najjednoduchšie z nich (SelectSort, BubbleSort, InsertSort...) zvyknú na usporiadanie poľa veľkosti n potrebovať (v najhoršom prípade) až rádovo n^2 porovnaní. Ak ste použili takýto algoritmus, pravdepodobne vaše riešenie už na väčších vstupoch prekročilo časový limit.

O trochu lepší prístup (ale stále nie najlepší) je použiť triediaci algoritmus, ktorý vždy urobí $O(n \log n)$ porovnaní[1] prvkov (napr. HeapSort, vhodne implementovaný QuickSort, MergeSort, IntroSort[2] a mnohé iné). Takéto riešenie už malo stihnúť dobehnúť v časovom limite aj na najväčších vstupoch.

[1] čítaj: „nanajvýš rádovo $n \cdot \log(n)$ porovnaní“
[2] Ten používa napr. funkcia `sort()` v C++.

Efektívnejšie riešenie

Ako by sa dalo vyššie popísané priamočiare riešenie vylepšiť? Keďže nás zaujíma iba Luxuskova pozícia v usporiadanom poli, nezaujíma nás, kto a v akom poradí bude stáť pred ním – stačí nám iba počet tých ľudí! Ten vieme zistiť tak, že raz prejdeme pole, každé meno porovnáme s Luxuskom a zapamätáme si, koľko z nich bolo v abecede pred ním. Ak k tomu pripočítame 1 za samotného Luxuska, dostaneme jeho pozíciu v usporiadanom poli. Takéto riešenie urobí presne $n - 1$ porovnaní, jeho časová zložitosť je teda lineárna od n. A keďže vždy musíme načítať celý vstup a pozrieť sa na všetky mená v ňom, rádovo lepšiu časovú zložitosť dosiahnuť už nevieme – toto riešenie je teda z hľadiska časovej zložitosti optimálne.

Porovnávanie reťazcov

Šikovnejšie programovacie jazyky majú v sebe pre reťazce (`string`-y) priamo zadefinované porovnávanie, aké potrebujeme: porovnajú sa prvé znaky, v prípade ich rovnosti druhé, a tak ďalej, až kým sa nenájde rozdiel alebo jeden z reťazcov neskončí. Odborne sa to volá *lexikografické usporiadanie*. Nemusíme teda my pracne porovnávať dva reťazce znak po znaku, ale môžeme v programe jednoducho použiť napríklad operátor „`<`“. Pozrite si nižšie vzorovú implementáciu.

Ešte si treba uvedomiť, ako sa vlastne porovnávanie reťazcov prejaví na časovej zložitosti programu. Bez ohľadu na to, či porovnávanie implementujeme sami, alebo použijeme internú funkciu programovacieho jazyka, nemôžeme len tak vyhlásiť, že porovnanie dvoch reťazcov má konštantnú časovú zložitosť. Totiž ak porovnávame dva skoro identické reťazce, môže byť potrebný počet porovnaní znakov až priamo úmerný dĺžke porovnávaných reťazcov.

V našej úlohe bola dĺžka všetkých reťazcov obmedzená na 47 znakov, a teda na každé porovnanie dvoch reťazcov nám stačilo nanajvýš 47 porovnaní znakov. Je teda v poriadku túto konštantu zanedbať a tvrdiť, že časová zložitosť šikovného triedenia je $O(n \log n)$ a že časová zložitosť nášho vzorového riešenia je $O(n)$, teda že je lineárna od počtu reťazcov.

Ak by takéto obmedzenie v zadaní nebolo, mohli by sme prehlásiť, že časová zložitosť nášho optimálneho programu je $O(\ell n)$, kde ℓ je dĺžka najdlhšieho mena na vstupe.

(Presnejšie, najpomalšou časťou je načítanie vstupu, pri ktorom musíme postupne zo vstupu prečítať všetky písmená všetkých reťazcov. Porovnanie každého reťazca s reťazcom „`Luxusko`“ následne vieme spraviť na nanajvýš 7 porovnaní znakov.)

Pamäťová zložitosť

Obidve vyššie popísané riešenia majú pamäťovú zložitosť $O(n)$, keďže máme v pamäti pole obsahujúce všetky mená. (Ak by sme nemali obmedzenie na dĺžku mien, museli by sme aj pamäťovú zložitosť odhadnúť ako $O(\ell n)$.)

V optimálnom riešení však túto pamäťovú zložitosť vieme zlepšiť. Netreba si pamätať celé pole, stačí nám niekoľko premenných – na zapamätanie si pôvodnej polohy Luxuska a počtu ľudí, ktorí sú skôr v abecede ako on. Každý reťazec, ktorý nám príde na vstupe, môžeme hneď po spracovaní „zahodiť“. Takéto riešenie má pamäťovú zložitosť $O(1)$, čo už samozrejme vylepšiť nejde.

Listing programu (Pascal)

```
var i, n, poc, poz: longint;
    meno: string;

begin
  ReadLn(n);
  poc := 0;
  for i := 1 to n do begin
    ReadLn(meno);
    if meno <= 'Luxusko' then inc(poc);
    if meno  = 'Luxusko' then poz := i;
  end;
  if poz > poc then WriteLn('o ', poz-poc, ' dopredu')
    else if poc > poz then WriteLn ('o ', poc-poz, ' dozadu')
      else WriteLn('neposunie sa');
end.
```

3. Životné poistenie

vzorák napísal Bob
(max. 7 b za popis, 3 b za program)

Hoci nám úloha umožňuje uzatvárať zmluvy platné v ľubovoľnom intervale dní $[x, y]$, v tomto vzoráku si vystačíme len s takými, ktoré končia v posledný deň (teda budú platné v nejakom intervale $[x, n]$). Každé optimálne riešenie (sadu nových zmlúv) vieme totiž prerobiť na tento špeciálny tvar bez toho, aby sa počet nových zmlúv menil – jednoducho ich všetky predĺžime až do dňa n. Premyslite si, že ak máme ľubovoľnú množinu zmlúv, ktorá dosiahne monotónnosť požadovanú v zadaní, tak nám túto monotónnosť nepokazí, ak

niektorej zmluve zmeníme koniec platnosti na posledný deň – totiž len zväčšíme niekoľkým posledným dňom počet pokrytí o 1.

Pridanie zmluvy $[x, n]$ ovplyvní rozdiel len medzi dňami $x-1$ a x; rozdiely počtov platných zmlúv medzi ostatnými dvojicami po sebe idúcich dní sa nijako nezmenia. To znamená, že sa môžeme zaoberať každou dvojicou dní osobitne. Ak pre nejaké x platí $a_{x-1} > a_x$, musíme uzatvoriť aspoň $a_{x-1} - a_x$ nových zmlúv s platnosťou $[x, n]$, aby sme splnili monotónnosť medzi dňami $x-1$ a x. No a práve toľko ich aj uzatvoríme, čím dosiahneme minimálny počet nových zmlúv.

Pre názornosť si to ukážeme na príklade. Nech je na vstupe postupnosť $(2, 5, 3, 7, 4)$. Vidíme, že $5 > 3$, čiže na tretí deň máme menej zmlúv ako na druhý. Pridáme teda $5 - 3 = 2$ zmluvy na interval dní $[3, 5]$. Tým dosiahneme nové počty pokrytí: $(2, 5, 5, 9, 6)$, čím už je druhý a tretí deň v poriadku. Všimnite si, že problém medzi štvrtým a piatym dňom sa nám nezmenil: ako predtým $(7 - 4 = 3)$, tak aj teraz $(9 - 6 = 3)$ budeme ešte potrebovať ďalšie 3 zmluvy pokrývajúce len samotný piaty deň.

Implementácia tejto myšlienky je veľmi jednoduchá: Postupne načítavame vstup a vyhodnocujeme rozdiel počtu starých zmlúv pre každú dvojicu po sebe idúcich dní. Ak by mal počet zmlúv klesnúť, pripočítame si k výsledku potrebný počet nových zmlúv. Postačí nám teda konštantná pamäť. Časová zložitosť algoritmu je $O(n)$, teda lineárna od počtu dní.

Listing programu (Pascal)

```
var n, i, a, last: longint;
    res: int64;

BEGIN
  ReadLn(n);
  last := 0;
  res := 0;
  for i := 1 to n do begin
    Read(a);
    if a < last then
      res := res + last - a;
    last := a;
  end;
  WriteLn(res);
END.
```

vzorák napísal Syseľ

4. Zvieratká

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

Najskôr si musíme uvedomiť, čo potrebujeme spočítať. Pre každé políčko potrebujeme rozhodnúť, či doň zasahuje fľak, ktorý ostal po nejakej vši. Tento nápad vedie k najjednoduchšiemu, ale pomalému riešeniu:

Vytvoríme si dvojrozmerné pole rozmerov plachty. Potom budeme prechádzať všetky vši na vstupe a pre každú si do plachty zakreslíme každé políčko fľaku, čo vytvorí. Ak však budú vši na plachte husto, toto riešenie bude mať zložitosť až $\Theta(rsn^2)$.

Na fľaky sa však môžeme pozrieť aj z pohľadu políčka. Na to, aby bol na nejakom políčku fľak, musí byť neďaleko nejaká voš. Presnejšie, musela byť od nášho políčka v oboch súradniciach vzdialená najviac o n, teda musela ležať vo štvorci so stranou $2n + 1$ a so stredom v našom políčku.

Ak sa teda v tomto štvorci žiadna voš nenachádza, políčko je čisté. Stačí však iba jedna voš vo štvorci a už naše políčko zašpiní. A ako zistíme, či sa vo štvorci nachádza nejaká voš? Pre každý štvorec $(2n + 1) \times (2n + 1)$ si spočítame, koľko vší sa v ňom nachádza.

Tu predstavím druhé neoptimálne (avšak už rýchlejšie) riešenie. Pre každý riadok vstupu si na každom súvislom úseku dĺžky $2n + 1$ spočítame vši. To vieme urobiť v čase lineárnom od dĺžky riadka tak, že úsek posúvame v riadku po jednom políčku a všímame si, či do neho nevošla alebo z neho nevyšla nejaká voš. Potom už pre každé políčko výstupu sčítame $2n + 1$ takýchto úsekov a máme súčet celého štvorca.

Možno sa vám to nezdá, ale už sme skoro na konci riešenia. Presne tú istú fintu, akú sme použili na rátanie počtu vší v úseku, vieme totiž použiť aj na spočítanie $2n+1$ úsekov pod sebou. Posúvame virtuálny štvorec dole plachtou a vždy len pripočítame nový riadok (súčty v úsekoch širokých $2n + 1$ sme už vypočítali) a odpočítame posledný.[3] Po vypočítaní počtu vší v danom štvorci budeme vedieť, či je jeho stredné políčko zafarbené.

Ostáva nám teda určiť časovú a pamäťovú zložitosť. Zakaždým prechádzame celú plachtu a pre každé políčko spravíme nejaké konštantné operácie a zapamätáme si niečo. Teda časová aj pamäťová zložitosť algoritmu je $\Theta(rs)$.

[3] Zaujímavé je, že keď dvojrozmerné pole prechádzame po riadkoch (prvá súradnica sa mení pomalšie), bude to v praxi na počítačoch kvôli cache pamäti o konštantu rýchlejšie oproti prechádzaniu po stĺpcoch. Preto budeme posúvať dole naraz celý pás s štvorcov $(2n + 1) \times (2n + 1)$.

Poznámka

Trochu iným prístupom sa dala dosiahnuť aj pamäťová zložitosť $\Theta(s)$, zamyslite sa, ako by ste to spravili. Na 15 bodov stačí $\Theta(rs)$, ale v prípade dobrého riešenia s pamäťovou zložitosťou $\Theta(s)$ môžeme udeliť jeden bonusový bod.

Listing programu (C++)

```
#include<iostream>

using namespace std;

string plachta[2000]; // Obraz plachty
int sumy_usekov[2000][2000]; // Počet vší v úsekoch dĺžky 2N+1 okolo políčka
int stvorce[2000]; // Aktuálny počet vší vo štvorci okolo daného stĺpca

int main(){
        int N, R, S; // Premenné zo zadania
        int odisiel; // Index políčka/riadku, ktoré práve opustilo úsek/štvorec
        int prisiel; // Index políčka/riadku, ktoré práve vošlo do úseku/štvorca
        int usek; // Aktuálny počet vší v úseku

        // AKO VYZERÁ PLACHTA? VSTUP NÁM POVIE...
        cin >> R >> S >> N;
        for(int r = 0; r < R; r++){
                cin >> plachta[r];
        }

        // VÝPOČET POČTU VŠÍ V ÚSEKOCH
        // Každý riadok zvlášť
        for(int r = 0; r < R; r++){
                // Na začiatku je úsek prázdny
                usek = 0;
                // Pridáme do úseku celú pravú časť (až po stred)
                for(int s = 0; (s < N) && (s < S); s++){
                        if(plachta[r][s] == '#') usek++;
                }
                // Posúvame stred úseku "s" po plachte
                for(int s = 0; s < S; s++){
                        prisiel = s + N; // Pravý okraj - práve prišiel
                        odisiel = s - N - 1; // Pred ľavým okrajom - odišiel
                        // Odišla voš?
                        if( (odisiel >= 0) && (plachta[r][odisiel] == '#') ) usek--;
                        // Prišla voš?
                        if( (prisiel < S) && (plachta[r][prisiel] == '#') ) usek++;
                        // Zapíšeme si súčet vší
                        sumy_usekov[r][s] = usek;
                }
        }
        // VÝPOČET POČTU VŠÍ V ŠTVORCOCH sumy_usekov[r][s]A VYTVORENIE PLACHTY
        // Na začiatku je každý štvorec prázdny
        for(int s = 0; s < S; s++) stvorce[s] = 0;
        // Pridáme do štvorcov N spodných riadkov - až po ich stredy
        for(int r = 0; (r < N) && (r < R); r++){
                for(int s = 0; s < S; s++) stvorce[s] += sumy_usekov[r][s];
        }
        // Posúvame stredy štvorcov "r" po plachte
        for(int r = 0; r < R; r++){
                prisiel = r + N; // Spodný riadok - práve prišiel
                odisiel = r - N - 1; // Riadok nad vrchným - práve nás opustil
                // Rátame pre stred v každom stĺpci
                for(int s = 0; s < S; s++){
                        // Koľko vší nás opustilo?
                        if( odisiel >= 0 ) stvorce[s] -= sumy_usekov[odisiel][s];
                        // Koľko vší sme privítali?
                        if( prisiel < R ) stvorce[s] += sumy_usekov[prisiel][s];
                        // Vypočítame ako bude stred štvorca vyzerať
                        if( stvorce[s] > 0) plachta[r][s] = '#';
                        else plachta[r][s] = '.';
                }
        }

        // PLACHTA JE HOTOVÁ - STAČÍ JU LEN VYPÍSAŤ!
        for(int r = 0; r < R; r++){
                cout << plachta[r] << endl;
        }
        return 0;
}
```

vzorák napísal MišoF

5. Oko za oko, zub za zub (časť druhá)

(max. 0 b za popis, 16 b za program)

Tak čo, ako sa vám páči `sed`? Samozrejme, v 99% situácií v praxi sa zaobídeme s obyčajným príkazom „`s/stare/nove/`“, ale oplatí sa vedieť, že jazykom regulárnych výrazov vieme ľahko popísať aj omnoho komplikovanejšie zmeny.

Poďme sa teda pozrieť na riešenia súťažných podúloh.

1. *Zmeniť všetky malé samohlásky (`aeiouy`) na zavináče (`@`).*

Malé písmená zodpovedajú vzorke „`[aeiouy]`“. Všetky jej výskyty v každom riadku chceme zmeniť na zavináče. To spravíme jednoducho príkazom „`s/[aeiouy]/@/g`“.

2. *Každé prirodzené číslo deliteľné piatimi nahradiť reťazcom* `BAC`. *(Za „prirodzené číslo“ považujeme každý najdlhší podreťazec tvorený len ciframi. Aj* `047` *a dokonca aj* `00` *sú teda prirodzené čísla.)*

   Prirodzené čísla deliteľné piatimi sú jednoducho čísla, ktoré končia nulou alebo päťkou. Takže hľadáme úsek ľubovoľných čísel, potom 0 alebo 5, a za ňou buď koniec riadka, alebo znak iný ako cifru. Najprehľadnejšie bude napísať dva príkazy, ktoré `sed` postupne oba použije:

   ```
   s/[0-9]*[05]([^0-9])/BAC\1/g
   s/[0-9]*[05]$/BAC/g
   ```

   Všimnite si, že v prvom príkaze sme použili označenie časti starého textu a referenciu naň v novom – chceme totiž skontrolovať, že za číslom nasleduje iný znak, ale ten iný znak nechceme zmazať.

   Tiež si všimnite, že nie je potrebné kontrolovať, čo je *pred* číslom. Tu totiž využijeme, že `sed` sám od seba nájde najskôr začínajúci výskyt vzorky.

3. *Každému riadku, ktorý obsahuje reťazec „*`zadanie`*“, doplniť na koniec tri výkričníky.*

   Nová časť syntaxe, ktorú potrebujeme, je vedieť príkaz použiť len na niektoré riadky. Riešenie vyzerá nasledovne: „`/zadanie/␣s/$/!!!/`“. (Znak „␣“predstavuje medzeru.)

4. *Zmazať všetky prázdne riadky. (T.j., vypísať len riadky obsahujúce aspoň jeden znak. Aj medzera je znak.)*

   Opäť použijeme syntax z riešenia predchádzajúcej podúlohy: chceme použiť príkaz len na riadky, ktoré zodpovedajú vzorke „`^$`“. Len tentoraz nepôjde o príkaz `s///` (substitute), ale o príkaz `d` (delete).

   Celé riešenie: „`/^$/␣d`“.

5. *Z každého riadka, v ktorom sú aspoň dve slová, zmazať druhé slovo. Prípadné medzery musia ostať nedotknuté. Teda napr. riadok „*`krtko␣␣kopal␣jamu`*“ sa má zmeniť na „*`krtko␣␣␣jamu`*“, zatiaľ čo riadok „*`␣␣jedna␣`*“ má ostať nezmenený.*

   Opäť využijeme, že `sed` nájde najľavejší výskyt vzorky, a navyše každému kvantifikátoru (`*`) zodpovedá najdlhší možný úsek. Stačí nám teda bez akejkoľvek ďalšej kontroly nájsť najľavejší výskyt vzorky (znaky prvého slova, znaky netvoriace slovo, znaky druhého slova) a z nej nechať len prvé dve časti.

   V syntaxi programu `sed` to vyzerá nasledovne: „`s/([a-zA-Z0-9_]+[^a-zA-Z0-9_]+)[a-zA-Z0-9_]+/\1/`“.

6. *Pre každý riadok vstupu, ktorý je tvaru „*`X:␣X`*“ (pričom* `X` *je ľubovoľný, možno aj prázdny reťazec), vypísať riadok obsahujúci* `X`*. Pre iné riadky vstupu nevypísať nič.*

   Stačia nám dva postupne vykonané príkazy. Prvým zmažeme všetky riadky, ktoré nie sú tvaru zo zadania. No a keď nám už ostali len riadky tvaru „`X:␣X`“, čo s nimi? Jednoducho v každom z nich zmažeme dvojbodku a všetko, čo za ňou nasleduje. Celé riešenie:

   ```
   /^(.*): \1$/ !d
   s/:.*//
   ```

   V zadaní sme naznačili, že v kombinácii s príkazmi `seq` a `factor` vieme ľahko zostrojiť zoznam prvočísel. Dokonca to ide ešte o niečo stručnejšie – ak vieme, že vstupom pre `sed` je výstup programu `factor` a tomu na vstupe nedáme číslo 1, stačí nám aj stručnejší regulárny výraz:

   ```
   seq 2 100 | factor | sed '/ .* / d ; s/:.*//'
   ```

   (Najskôr zmažeme všetky riadky s aspoň dvomi medzerami, a potom z toho čo ostalo zmažeme dvojbodku a všetko za ňou.)

7. *Za každé prirodzené číslo (viď podúlohu 2), za ktorým nasleduje medzera a prirodzené číslo, doplniť čiarku. Teda napr. riadok „*`1␣2␣3␣jahoda␣4␣␣5`*“ sa má zmeniť na „*`1,␣2,␣3␣jahoda␣4␣␣5`*“.*

   Táto zmena je v skutočnosti ľahšia ako sa zdá. Totiž vôbec nemusíme kontrolovať celé prirodzené čísla. Úplne stačí zmeniť každú postupnosť znakov „cifra medzera cifra“ na postupnosť „cifra čiarka medzera cifra“.

Toto by malo ísť ľahko, nie? Malo by stačiť nasledovné: „`s/([0-9]) ([0-9])/\1, \2/g`“. Lenže nestačí.

A to už prečo? Veď sme použili prepínač `g`, má to zmeniť všetky výskyty na riadku!

Preto, že sme trochu zavádzali. Keď `sed` nájde nejakú vzorku a zmení ju, pokračuje v hľadaní ďalej až od miesta, kde prvá nájdená vzorka skončila. Ak teda použijeme vyššie uvedenú vzorku na riadok „1␣2␣3␣4␣5␣6␣7“, stane sa nasledovné: Najskôr `sed` nájde výskyt „1␣2“a zmení ho na „`1,␣2`“. Teraz mu na spracovanie zostalo „␣3␣4␣5␣6␣7“. Tam nájde ďalší výskyt hľadanej vzorky: „3␣4“. Ten zmení... a tak ďalej. Na konci teda dostaneme takýto výstup: „`1,␣2␣3,␣4␣5,␣6␣7`“.

Čo s týmto problémom? Riešenie je našťastie veľmi jednoduché: teraz už len stačí znovu použiť presne ten istý príkaz. Tentokrát prvý výskyt hľadanej vzorky bude „2␣3“, druhý „4␣5“a tretí „6␣7“.

Celé riešenie teda vyzerá nasledovne:

```
s/([0-9]) ([0-9])/\1, \2/g
s/([0-9]) ([0-9])/\1, \2/g
```

8. *Zoznam prirodzených čísel vyzerá nasledovne: dvojbodka, medzera, prirodzené číslo, medzera, prirodzené číslo, a tak ďalej. Za posledným prirodzeným číslom v zozname medzera byť môže, ale nemusí. Môže tam byť aj koniec riadka či iný znak namiesto medzery.*

   Chceli by sme krajšie zoznamy prirodzených čísel: také, kde za každým číslom okrem posledného máme najskôr čiarku a až potom medzeru. Teda napr riadok „`cisla:␣1␣2␣3␣4␣5␣a␣6␣7`*“ by sa mal zmeniť na riadok „*`cisla:␣1,␣2,␣3,␣4,␣5␣a␣6␣7`*“. Nezabudnite, že takých zoznamov môže byť v jednom riadku textu aj viac.*

   Tentoraz už nestačí aplikovať nejakú vzorku lokálne. Vždy, keď pridávame čiarku, musí to byť do zoznamu čísel začínajúceho dvojbodkou. Aby sme sa príliš nezamotali, vydáme sa cestou, ktorá síce povedie k riešeniu nie-práve-najrýchlejšiemu, ale zato jednoduchému. Najskôr napíšeme vzorku, ktorá nájde jeden zoznam čísel, ktorý ešte čiarky nemá, a pridá doň jednu jedinú čiarku (a to na poslednú možnú pozíciu). No a túto vzorku budeme potom na každý riadok znova a znova používať, až kým nenastane situácia, že už naša vzorka nikam nepasuje.

   Ako to bude celé vyzerať? Najskôr ošetríme špeciálny prípad: zoznam končiaci koncom riadka. Hľadáme teda dvojbodku, zoznam čísel oddelených medzerami, a za posledným koniec riadka. Ak toto nájdeme, pred posledné číslo vložíme čiarku: „`s/(:( [0-9]+)+) ([0-9]+)$/\1, \3/`“.

   No a teraz všeobecný prípad: dvojbodka, za ňou zoznam čísel oddelených medzerami, a za posledným z nich už je niečo iné. To „niečo iné“ môže byť znak iný ako cifry a medzera. Ale netreba zabudnúť ani na situáciu, kedy je tam ešte jedna medzera. Aby zoznam nepokračoval, za tou musí byť buď priamo koniec riadka, alebo nejaká nie-cifra. A to už sú všetky možné prípady. Dostávame teda nasledovný príkaz: „`s/(:( [0-9]+)+) ([0-9]+)( $| [^0-9]|[^0-9])/\1, \3\4/`“.

   Keď si to celé dáme dokopy a zabalíme do správneho cyklu, dostávame výsledné riešenie:

```
s/(:( [0-9]+)+) ([0-9]+)$/\1, \3/
:again
s/(:( [0-9]+)+) ([0-9]+)( $| [^0-9]|[^0-9])/\1, \3\4/
t again
```

   Keď tento „program“ spustíme na riadku „`x:␣1␣2␣3,␣y:␣4␣5␣6`“, postupne pridá čiarky pred cifry 6, 3, 2 a nakoniec 5.

   A či by to šlo aj bez toho cyklu? Skúste sa zamyslieť, nie je to ale ľahká otázka.

Tak, a tým sme úspešne ukončili druhý diel nášho mini-seriálu o regulárnych výrazoch. Pýtate sa, s čím ďalším sa stretnete v letných dvoch sériách? Nechajte sa prekvapiť...

6. Oprava kódu

vzorák napísal Usámec
(max. 10 b za popis, 10 b za program)

Úvodom riešenia spravíme jednu veľmi očividnú vec: pre každý stĺpec si spočítame, koľko je tam čiernych pixelov, a výsledok si uložíme do poľa $P[0, \ldots, s-1]$. Toto vieme spraviť v čase $O(rs)$.

Po krátkom zamyslení je zjavné, že riešenie tejto úlohy pôjde buď cez rekurziu s memoizáciou, alebo cez dynamické programovanie. Oba prístupy sú viac-menej ekvivalentné, my použijeme dynamické programovanie.

Budeme si počítať nasledujúce hodnoty: $C[0, \ldots, s]$ a $B[0, \ldots, s]$. $C[i]$ pre $i > 0$ vyjadruje, koľko najmenej pixelov musíme zmeniť, aby prvých $i - 1$ stĺpcov tvorilo platný čiarový kód končiaci čiernou čiarou. Ak sa to vôbec nedá dosiahnuť, $C[i]$ bude rovné nekonečnu. $B[i]$ definujeme podobne s tým rozdielom, že čiarový kód musí končiť bielou čiarou. Špeciálne si zvolíme $B[0] = C[0] = 0$. Naše riešenie potom bude minimum z $B[s]$ a $C[s]$.

Ako spočítať príslušnú hodnotu $C[i]$? Ide to celkom jednoducho: prejdeme všetky možné dĺžky d poslednej čiernej čiary od x do y a pre konkrétnu dĺžku d spočítame, koľko políčok musíme zmeniť, aby sme dostali čiernu čiaru; nech je výsledok z_d. Keď spočítame $z_d + B[i - d]$, tak vieme, koľko políčok musíme zmeniť, aby posledná čiara bola čierna a mala dĺžku d. Takto prejdeme všetky prípustné dĺžky d a vyberieme najlepšie riešenie pre $C[i]$. Jedno počítanie $C[i]$ vieme urobiť v čase $O(y)$. Malá poznámka: hodnotu z_d nemusíme počítať stále od začiatku, ale vieme ju z hodnoty z_{d-1} odvodiť v konštantnom čase (stačí pripočítať príslušnú hodnotu použitím poľa P).

Hodnoty $B[i]$ spočítame presne rovnako.

Celkovo teda máme riešenie, ktoré beží v čase $O(s(r+y))$. Pamäťové nároky sú $O(s)$, prípadne $O(rs)$, pokiaľ sme príliš leniví pri načítaní.

Pre náročnejších riešiteľov: Dokážete vymyslieť algoritmus, ktorý dostane skomprimovaný vstup (pole P a parametre x, y) a vypočíta výsledok tejto úlohy v čase $O(s)$?

Listing programu (C++)

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

#define For(i,n) for(int i=0; i<(n); i++)
#define INF 1000000000
int r, s, x, y;
int P[1047], B[1047], C[1047];

int main() {
  scanf("%d %d %d %d", &r, &s, &x, &y);
  For(i, r)
    For(j, s) {
      char c;
      scanf("%c ", &c);
      if (c == '#') P[j]++;
    }
  For(i, s + 1) B[i] = C[i] = INF;
  B[0] = C[0] = 0;
  for (int i = 1; i <= s; i++) {
    int zd = 0;
    For(j, min(i, x - 1)) zd += P[i - j - 1];
    for (int j = x - 1; j < y; j++) {
      if (i - j - 1 < 0) continue;
      zd += P[i - j - 1];
      B[i] = min(B[i], zd + C[i - j - 1]);
      C[i] = min(C[i], (j + 1) * r - zd + B[i - j - 1]);
    }
  }
  printf("%d\n", min(C[s], B[s]));
}
```

vzorák napísal Žaba

7. Obchod s číslami

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

Dúfam, že sa vám táto úloha páčila. Mne osobne veľmi a ak sa vám ju nebodaj nepodarilo vyriešiť a už sa s ňou nechcete ďalej trápiť, rýchlo pokračujte v čítaní.

Prvé čo začneme robiť je, že si skúsime generovať nejaké postupnosti a sledovať čo sa deje. Problém však je, že keď sa na to pozeráme odpredu, možností, ako tá postupnosť môže pokračovať, je hrozne veľa a nevieme veľmi určiť, ktorá by pre nás bola výhodná. Pozrime sa teda na celý problém od konca. Skôr ako začneme, zavedieme si nejaké označenie. Slovom stav budeme označovať dvojicu čísiel bez ohľadu na to, ktoré je hore a ktoré je dole. Keďže musíme začať stlačením tlačidla `H`, po prvom kroku budeme v symetrickom stave $(1, 1)$. Ak sa z neho nejakou postupnosťou dostaneme do pozície (x, y), tak tým, že invertujeme stláčania, sa dostaneme do stavu (y, x). Tieto dva stavy sú teda ekvivalentné vzhľadom na ich dosiahnuteľnosť. Rozdiel môže byť v počte rovnakých susedných dvojíc. V tom prípade si nám však stačí vybrať tú postupnosť, ktorá po prvotnom `H` pokračuje stlačením `D`, aby sme si nevytvorili zlú dvojicu naviac. Ďalšia dôležitá vlastnosť je, že keď vychádzame zo stavu $(1, 1)$, všetky čísla, ktoré zapíšeme, budú kladné.

Vieme, že na konci sme zapísali číslo r. V druhom riadku je číslo x, o ktorom vieme povedať, že $x \leq r$. Prečo? No r vzniklo ako sčítanie predchádzajúcich dvoch riadkov a keďže riadok, v ktorom je x, sa nezmenil, tak r vzniklo ako súčet x s niečím iným. Ale počkať. Teraz vieme dokonca aj ako vyzeral stav pred posledným

stlačením. V jednom riadku je číslo x a v druhom musí byť číslo $r-x$, lebo súčet týchto čísel musí byť r. Super, a vieme povedať aj ako vyzeral stav ešte pred tým? Vieme – rovnakým postupom.

Ak si teda povieme, aké číslo je x, tak vieme presne zrekonštruovať postupnosť. Stačí vždy od väčšieho čísla odčítať menšie a dostaneme stav pred tým. Ak sa týmto postupom dostanem do stavu $(1,1)$ na práve $n-1$ odčítaní, máme nejakú validnú postupnosť. Toto by však ešte bolo pomalé, lebo potrebujeme vyskúšať r možností ako vyzerá x a pre každú vygenerovať postupnosť, ktorá je dlhá n. Teda zatiaľ máme riešenie v čase $O(rn)$. To je však dosť pomalé. Potrebujeme niečo vylepšiť.

Pozrime sa na stav (a,b), pričom b bude oveľa väčšie ako a. Stavy pred tým vyzerali nasledovne: $(a,b-a)$, $(a,b-2a),\ldots,(a,b-ka)$. Postupne odčítavame od b číslo a, až kým nám nezostane číslo menšie ako a. Uvedomme si však, že to je vlastne $b \bmod a$ a $k=\lfloor b/a \rfloor$. A celé toto niečo hrozne pripomína. Postupne modulujeme väčšie číslo menším, však to je Euklidov algoritmus na hľadanie najväčšieho spoločného deliteľa. A to je celé, stačí nám už len trochu upraviť tento algoritmus. Budeme si v ňom rátať, aká dlhá je tá postupnosť, a tiež koľko rovnakých prvkov za sebou ide. Vždy, keď máme stav (a,b), tak si pridáme k k dĺžke postupnosti a $k-1$ k počtu zlých dvojíc.

Nakoniec si už len dogenerujeme postupnosť končiacu stavom (r,x) pre také x, ktorému prislúcha validná postupnosť s minimálnym počtom zlých dvojíc. Pri generovaní postupnosti si kvôli šetreniu pamäti pamätáme len jednotlivé k, čo znamená počet rovnakých stlačení za sebou. Tým pádom si pamätáme rádovo toľko čísiel, koľko krokov spraví náš Euklidov algoritmus, čo je rádovo $\log r$.

Keďže Euklidov algoritmus má zložitosť $O(\log r)$[4] a použijeme ho najviac r-krát, časovú zložitosť máme $O(r \log r)$. Pamätať si potrebujeme akurát postupnosť, čo bude $O(\log r)$.

Listing programu (C++)

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

int n, r;
vector<int> vys, A;

int suma(vector<int> P) {
  int vys=0;
  for(int i=0; i<P.size(); i++) vys+=P[i];
  return vys;
}

void solve(int a, int b) {
  if(a==0) return;
  if(a==1) {
    if(b>1) A.push_back(b-1);
    A.push_back(1);
    if(A.size()>vys.size() && suma(A)==n) vys=A;
  } else {
    A.push_back(b/a);
    solve(b%a,a);
  }
}

int main() {
  scanf("%d %d", &n, &r);
  for(int i=1; i<=r; i++) {
    A.clear();
    solve(i,r);
  }
  if(vys.size()==0) printf("Neexistuje\n");
  else {
    int p=0;
    for(int i=vys.size()-1; i>=0; i--)
    {
      for(int j=0; j<vys[i]; j++)
        if(p) printf("D");
        else printf("H");
      p++; p%=2;
    }
    printf("\n");
  }
}
```

vzorák napísal Jano

8. Ochutené lahôdky

(max. 13 b za popis, 12 b za program)

Pohodlne sa usaďte a buďte pripravení. Pripravení na to, že uvidíte veci, ktoré sa aj vo svete algoritmov zjavujú len málokedy. Veru s časovou zložitosťou závislou od odmocniny n sa nestretnete každý deň a len zriedkavo potrebujete okresávať aj pamäťovú zložitosť. Uvidíte, ako je dôležité odpovedať na otázky v správnom

[4] Ak $a \le b$, potom $b \bmod a < b/2$. Väčší z argumentov Euklidovho algoritmu sa preto po dvoch iteráciách určite zmenší aspoň o polovicu. To znamená, že pre vstup (a,b) vykonáme celkovo najviac $2\log_2 b$ iterácií.

poradí, ako sa vydať zdanlivo nepriechodným chodníkom, aby ste sa dostali do cieľa. Zažijete nejeden trik a prehryziete sa zopár písmenkami s dolnými indexami. Dobrú chuť.

Riešenie s odmocninou v zložitosti

Pomerne jednoduchý prístup k tejto úlohe je taký, že postupne prechádzame otázky (v poradí v akom ich máme na vstupe) a pre každú otázku sa snažíme čo najrýchlejšie zistiť odpoveď – koľko je v úseku od a po b takých čísel x, že sa tam vyskytujú práve x-krát – toto si nazvime Podmienka o počte výskytov, skrátene POPV.

Keď už sme pri označovaní, vstupné pole (predstavujúce celú Knihu jednoduchých receptov) si označme A. Interval od a po b vrátane si označíme $[a, b]$ a počet čísel spĺňajúcich POPV v intervale $[a, b]$ (teda odpoveď na otázku „a b“) označíme $p(a, b)$. (Indexujeme od nuly, takže $a, b \in \{0, \ldots, n-1\}$.)

Zistiť $p(a, b)$ môžeme napríklad tak, že prejdeme celý úsek $[a, b]$, spočítame si výskyty každého čísla v ňom a overíme, koľko z nich spĺňa POPV. Aby sme na počítanie výskytov nemuseli mať pole veľkosti 10^9, budeme automaticky ignorovať všetky čísla väčšie ako n – museli by sa vyskytnúť viac ako n-krát v poli veľkosti n, čo nejde.

Aha! Tu by mohlo skrsnúť isté podozrenie, že rôznych veľkých čísel spĺňajúcich POPV bude akosi málo, pretože sa tam musia vyskytovať akosi veľakrát a na to spotrebujú akosi veľa políčok. A nielen veľkých – ani tých malých veľa nebude. Presnejšie, keď si zoberieme všetky čísla, ktoré spĺňajú POPV aspoň v jednom intervale (a teda nám môžu ovplyvniť výstup programu) tak ich nie je viac, ako $\lceil\sqrt{2n}\rceil$.

Ak totiž nejaké x aspoň v jednom intervale spĺňa POPV, tak sa x v celom poli A musí vyskytovať aspoň x-krát. Ak by aj čísla spĺňajúce POPV boli najmenšie možné, musela by v poli A byť aspoň jedna 1, aspoň dve 2, aspoň tri 3, a tak ďalej.

Ak by teda bolo v poli A viac ako $s = \lceil\sqrt{2n}\rceil$ rôznych čísel, ktoré v nejakom intervale spĺňajú POPV, bol by celkový počet ich výskytov aspoň $1 + 2 + \cdots + (s+1) = (s+1)(s+2)/2 > n$, čo sa nám ale do n-prvkového poľa nezmestí.

A z tohto sa dá hneď vyrobiť rýchle funkčné riešenie – najprv si zistím, ktoré čísla x môžu niekedy splniť POPV – musia sa vyskytnúť presne x-krát v nejakom intervale, takže aspoň x-krát v celom poli. Prejdeme teda celé pole a pre každé číslo od 1 po n si v poli spočítame, koľko má výskytov. Pomocou tejto informácie následne ľahko nájdeme všetkých $O(\sqrt{n})$ čísel, ktoré niekde spĺňajú POPV. Toto celé vieme spraviť v čase $O(n)$.

Okrem toho si pre každé číslo x spĺňajúce POPV spravíme pole, označme ho $S[x]$, také, že $S[x][i]$ bude počet výskytov čísla x na prvých i pozíciách poľa A. (Špeciálne teda $S[x][0] = 0$.) Pre konkrétne x toto vieme spraviť v lineárnom čase, celkovo nám teda výroba týchto polí zaberie čas $O(n\sqrt{n})$.

Vďaka poľu $S[x]$ budeme vedieť pre ľubovoľný interval $[a, b]$ a číslo x zistiť v konštantnom čase, či x v danom intervale spĺňa POPV. Na to nám stačí skontrolovať, či $S[x][b] - S[x][a-1] = x$. Ak teda dostaneme nejakú otázku $[a, b]$, postupne prejdeme cez všetkých $O(\sqrt{n})$ čísel x, ktoré niekde spĺňajú POPV, a o každom z nich v konštantnom čase zistíme, či spĺňa POPV aj v tomto konkrétnom intervale. Na každú otázku teda vieme odpovedať v čase $O(\sqrt{n})$.

Celý algoritmus bude bežať v čase $O(n + n\sqrt{n} + q\sqrt{n}) = O((n+q)\sqrt{n})$. Pamäťová zložitosť bude $O(n\sqrt{n})$. Oj, lenže tá pamäťová zložitosť neznie dobre, lebo už pri $n = 150\,000$ nám samotné pole S zaberie vyše 320 MB, čo je nad pamäťový limit.

To sa dá okresať napríklad tým, že pole S nevytvárame celé naraz, ale vždy si ho spočítame len pre jedno z čísel x a potom prejdeme všetky otázky. V prípade, že v nejakej otázke x splní POPV, zvýšime počítadlo pre danú otázku o 1. Na konci len vypíšeme hodnoty počítadiel. Časová zložitosť sa nezmení, ostane $O((n+q)\sqrt{n})$, a pamäťová klesne na $O(n+q)$.

Teraz sa môžete pokochať implementáciou tohoto riešenia, a chvíľku si oddýchnuť, kým sa pustíme do vzoráku – áno, toto ešte na plný počet nestačilo :).

Listing programu (C++)

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

int N,Q; // počet čísel, počet otázok
int A[MAXN]; // čísla na vstupe
int P[MAXN], S[MAXN];
int qa[MAXN], qb[MAXN]; // otazka - od do
vector<int> V;
```

```
int main(){
    scanf("%d %d",&N,&Q);
    For(i,N) scanf("%d", A+i);

    For(i,N) P[i+1] = 0;
    For(i,N) if (A[i]<=N) P[A[i]]++;
    // v P[i] je teraz počet výskytov i v poli A
    For(i,N) if (P[i+1] > i) V.push_back(i+1);

    //načítam si otázky
    For(q,Q) {scanf("%d%d",qa+q,qb+q); qa[q]--;}
    For(q,Q) P[q] = 0; // do P budem ukladať odpovede

    For(v,V.size()) {
        // v S[i] bude počet výskytov 'v' v A[0] až A[i-1]
        For(i, N) S[i+1] = S[i] + (A[i]==V[v]);
        // v ktorých otázkach mám zarátať aj číslo v?
        For(q, Q) P[q] += (S[qb[q]] - S[qa[q]] == V[v]);
    }

    For(q,Q) printf("%d\n", P[q]);
}
```

O malý chlp lepšie riešenie (stále s odmocninou v zložitosti, ale trochu ináč)

Teraz sa poďme vrhnúť na myšlienku, z ktorej časom vypadne vzorové riešenie. Podobne, ako keď sme zlepšovali pamäťovú zložitosť v predchádzajúcom riešení, aj tu využijeme skutočnosť, že si môžeme najprv celý vstup načítať a až potom vypísať výstup. Na otázky teda nemusíme odpovedať v zadanom poradí, môžeme si zvoliť nejaké iné, vhodnejšie. Najlepšie to bude robiť nejako systematicky – napríklad si otázky usporiadame podľa konca intervalu, na ktorý sa pýtajú. Teda najskôr zodpovieme tie, pre ktoré interval končí na pozícii 0, potom všetky končiace pozíciou 1, a tak ďalej.

Aby nejaké číslo x splnilo v nejakom intervale POPV, musí mať v danom intervale presne x výskytov. Zoberme nejaké fixné x a označme V usporiadaný zoznam indexov do poľa A, na ktorých sa hodnota x nachádza.

Nech napríklad $x = 3$ a $V = (2, 4, 7, 10, 11)$. Pre ktoré úseky $[a, b]$ spĺňa číslo x POPV? V prvom rade zjavne koniec úseku musí byť na pozícii 7 alebo viac, inak by sme mali menej ako 3 výskyty x. Ak $b = 7$, musí a byť medzi 0 a 2. Takýto úsek potom obsahuje výskyty na pozíciách 2, 4 a 7. To isté platí aj pre $b = 8$ a $b = 9$. Akonáhle ale $b = 10$, už musí a byť medzi 3 a 4, vrátane – ak by bolo menšie, mali by sme výskytov x priveľa, ak by bolo väčšie, mali by sme ich primálo.

Nech $v = |V|$ je počet všetkých výskytov čísla x, teda nech sú výskyty x na pozíciách $V[0]$ až $V[v-1]$. Nech navyše $V[-1] = -1$ a $V[v] = n$ (akoby boli ešte dva výskyty čísla x tesne pred začiatkom a tesne za koncom poľa A). Potom vieme vyššie popísané pozorovanie zovšeobecniť nasledovne:

- Aby x spĺňalo POPV v intervale $[a, b]$, musí byť $v \geq x$ a $V[x-1] \leq b$.
- Ak pre nejaké $i \geq x-1$ platí $V[i] \leq b < V[i+1]$, tak musí byť $V[i-x] < a \leq V[i-x+1]$.

V našom riešení budeme postupne zvyšovať b a pre každé x, ktoré sme už aspoň raz videli, si budeme pamätať, či už môže toto x v nejakom intervale spĺňať POPV, a ak áno, v akom intervale musí ležať a, aby sa tak stalo.

Takto si v každom okamihu pamätáme nejakú množinu intervalov (o ktorej vieme, že ich tam je nanajvýš $\lceil\sqrt{2n}\rceil$, lebo viac rôznych x nemôže POPV spĺňať). Ak teraz máme nejakú otázku $[a, b]$, zodpovedať ju vieme tak, že spočítame, v koľkých rôznych intervaloch toto konkrétne a leží. Ak by sme to spravili priamočiaro (v cykle prejdeme všetky a spočítame), dostali by sme riešenie s časovou zložitosťou $O(n+q\sqrt{n})$, čo je o chlp lepšie ako naše predchádzajúce riešenie.

Príklad: Nech sme už z postupnosti A spracovali nasledovnú časť: $(2, 4, 3, 47, 3, 3, 5, 1, 3, 2)$. Teda aktuálne máme $b = 9$. Pre interval $[a, b]$ bude hodnota 1 spĺňať POPV ak $a \in [0, 7]$, hodnota 2 ak $a \in [0, 0]$, hodnota 3 ak $a \in [3, 4]$ a žiadna iná hodnota POPV spĺňať nemôže. V danej chvíli si teda pamätáme intervaly $[0, 7]$, $[0, 0]$ a $[3, 4]$. Ak teraz dostaneme napr. otázku $[4, 9]$, odpoveďou na ňu je číslo 2, lebo $a = 4$ leží v dvoch z intervalov, ktoré si pamätáme.

Vzorové riešenie (s logaritmom v zložitosti)

Vyššie popísané riešenie vieme zrýchliť použitím vhodnej dátovej štruktúry. Skôr, než si popíšeme jej implementáciu, zamyslime sa nad tým, aké operácie vlastne budeme od nej potrebovať.

Čo presne sa stane s našou množinou intervalov, keď zväčšíme b o jedna? Pre nové b dostávame nový výskyt hodnoty $x = A[b]$. Zjavne sa teda nezmenia intervaly týkajúce sa hodnôt iných ako x: naďalej budú pre ne fungovať tie isté a. Napr. keby sme vo vyššie uvedenom príklade zväčšili b na 10 a bolo $A[10] = 2$, naďalej bude platiť, že hodnota 3 spĺňa POPV práve vtedy, keď $a \in [3, 4]$.

Jediné, čo sa môže zmeniť, je interval, v ktorom má ležať a, aby práve táto hodnota x, ktorej nový výskyt máme na pozícii b, spĺňala POPV. Potrebujeme teda z našej dátovej štruktúry starý interval odstrániť a nový interval, v ktorom ležia dobré hodnoty a, do nej pridať. (Ak už samozrejme bolo dosť výskytov x. Môže sa stať, že sa neudeje vôbec nič, prípadne aj to, že len prvýkrát pridáme do našej dátovej štruktúry interval zodpovedajúci hodnote x a nič z nej nezmažeme.)

Navyše budeme potrebovať používať našu dátovú štruktúru na odpovedanie na otázky: keď dostaneme konkrétne a, potrebujeme vedieť, v koľkých spomedzi aktuálne pamätaných intervalov leží.

Zistiť, v koľkých intervaloch dané číslo a leží, vieme napríklad tak, že spočítame, koľko intervalov začína na pozícii a alebo skoršej, a od toho odčítame intervaly, ktoré končia na pozícii ostro skoršej ako a. (Všimnite si, že nám pri tom nezáleží na tom, ktorý koniec patrí ku ktorému začiatku.)

Môžeme teda použiť dve polia: v jednom si budeme pre každú pozíciu pamätať, koľko z našich intervalov tam začína, v druhom zase, koľko ich tam končí. Navyše budeme pre každé pole potrebovať vedieť povedať, koľko prvkov je dokopy na pozíciách 0 až x, pre dané x. Preto nad poľom postavíme intervalový strom. (Alebo, rovnako dobre, môžeme použiť Fenwickov strom, v našich končinách ľudovo nazývaný fínsky strom.) Síce nám tak každá zmena poľa bude trvať $O(\log n)$ krokov, ale zato budeme vedieť aj povedať súčet ľubovoľného úseku poľa v čase $O(\log n)$.

O tom ako intervalový strom funguje, sa dočítate napr. vo vzorovom riešení 10. úlohy 1. letného kola 27. ročníka KSP v sekcii Intervalový strom. `http://www.ksp.sk/wiki/uploads/Zadania/vzoraky273.pdf`

Celé to ide implementovať ešte o niečo kompaktnejšie. Napríklad nepotrebujeme tie polia dve, stačí nám jedno, do ktorého si budeme začiatky intervalov značiť ako $+1$ a konce intervalov ako -1. (Presnejšie, keď si chceme zapamätať nový interval $[a, b]$, pripočítame 1 na pozíciu a, kde začína, a odpočítame 1 od pozície $b+1$, ktorá doň ako prvá nepatrí. Ak chceme tento interval neskôr odstrániť, jednoducho urobíme opačné dve zmeny.)

Samotná implementácia algoritmu už vôbec nie je zložitá. Rozdeliť si otázky podľa konca intervalu vieme už popri načítavaní, stačí si pre každý koniec vytvoriť vector. Všetkých n vektorov bude mať dokopy q prvkov. Podobne si vieme do vectorov načítať aj pozície výskytov V, pre každé x zvlášť. Vieme to spraviť v čase $O(n+q)$ a zaberie nám to pamäť $O(n+q)$.

Následne prejdeme cez všetkých n fáz, pre každú najprv updatneme náš intervalový strom a potom zodpovieme na príslušné otázky. Keďže v každej fáze najviac 1 interval odstraňujeme a najviac 1 pridávame, update údajov v intervalovom strome vždy prebehne v čase $O(\log n)$. Podobne aj zodpovedanie každej otázky nám zaberie $O(\log n)$ kvôli výpočtu súčtu úseku v našom intervalovom strome. Intervalovému stromu stačí $O(n)$ pamäte.

Celkový čas behu programu bude teda $O((n+q)\log n)$. Program potrebuje $O(n+q)$ pamäte.

Nižšie uvedená ukážková implementácia nezodpovedá presne vyššie uvedenému popisu, ale je s ním ekvivalentná. Používa -1 pre začiatky intervalov, $+1$ pre ich konce a následne počíta súčet opačnej časti intervalu. Ak si pôvodné pole (teda listy nášho intervalového stromu) označíme P, tak v našej implementácii po spracovaní prvku $A[b]$ pre každé a platí, že $P[a] + P[a+1] + \cdots + P[b] = p(a, b)$, teda že počet čísel, ktoré spĺňajú POPV pre interval $[a, b]$ je rovný počtu intervalov, ktoré nezačnú, ale končia od pozície a ďalej.

Listing programu (C++)

```
#include<cstdio>
#include<algorithm>
#include<vector>
using namespace std;
#define For(i, n) for(int i = 0; i<(n); ++i)
#define MAXN (1<<19)
typedef long long ll;
typedef pair<int, int> pii;

int n,q, A[MAXN], otz_a[MAXN], otz_b[MAXN], odpoved[MAXN];
vector<int> konci[MAXN];
vector<int> V[MAXN];

// intervalový strom
int I[2*MAXN]; // P[i] = I[MAXN+i]
void add(int x, int v){
    x+=MAXN;
    while(x>0) I[x]+=v, x/=2;
}
int get(int x, int v = 1, int l = 0, int r = MAXN) {
    if (x<l) return 0;
    if (x>=r-1) return I[v];
    return get(x,2*v,l,(l+r)/2) + get(x,2*v+1,(l+r)/2,r);
}

int main(){
    // načítanie vstupu
    scanf("%d %d",&n,&q);
```

```
    For(i,n) scanf("%d", A+i);
    For(i,q) {
        scanf("%d%d",otz_a+i, otz_b+i);
        // rozdelenie otázok podľa konca intervalu
        konci[otz_b[i]].push_back(i);
    }

    For(i,n){
        int x = A[i], v;
        if (x<=n){ // veľké čísla ignorujeme
            // zapamätávanie pozícií výskytov
            V[x].push_back(i+1);
            v = V[x].size();
            // update poľa P
            if (v >= x) add(V[x][v-x], 1);
            if (v > x) add(V[x][v-x-1], -2);
            if (v > x+1) add(V[x][v-x-2], 1);
        }
        For(j, konci[i+1].size()) // odpovieme na otázky končiace i+1
            // odpoved[] = súčet ( P[otz_a[]] az P[otz_b[]] )
            odpoved[konci[i+1][j]] = get(otz_b[konci[i+1][j]]) - get(otz_a[konci[i+1][j]]-1);
    }

    For(i,q) printf("%d\n", odpoved[i]);
}
```


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