Vzorové riešenia 2. kola letnej časti

Feri (sebestyen.frantisek.michal@gmail.com)

1. Utrápený Marcel

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

Našou úlohou bolo pomôcť Samkovi a Emkovi zistiť, koľko predmetov treba vložiť do prvej tašky. Ukážeme si najprv pomalšie riešenie, ktoré potom vylepšíme. Zadanie hovorí, že Marcel do určitého momentu dáva predmety iba do prvej tašky a od tohto momentu ich dáva iba do druhej. Prvý predmet, ktorý Marcel vloží do druhej tašky, budeme v ďalšej časti vzoráku nazývať *zmrzlina*.

Pomalšie riešenie

Vyskúšame všetky možnosti pre zmrzlinu. Pre každú možnosť vypočítame súčet hmotností predmetov, ktoré sú nablokované pred zmrzlinou a súčet hmotností zvyšných predmetov. Čím menší je rozdiel týchto čísel, tým lepšie sú tašky vyvážené. Kedže všetkých predmetov (možností pre zmrzlinu) je n a pri každej možnosti musíme sčítať n hmotností, výsledná časová zložitosť bude $O(n^2)$, pamäťová je $O(n)$.

Listing programu (Python)

```
n = int(input())
predmety = list(map(int, input().split())) # nacitame predmety do pola a skonvertujeme na int-y
best_index = 0 # (zatial) najlepsia volba zmrzliny
najlepsie_vyvazenie = sum(predmety) # ako velmi su tasky vyvazene pri (zatial) najlepsej volbe

for i in range(n + 1): # n + 1, lebo chceme overit pripad kedy su vsetky predmety v prvej taske
    prva, druha = sum(predmety[:i]), sum(predmety[i:]) # scitame hmotnosti predmetov v prvej a druhej taske
    vyvazenie = abs(prva - druha) # a spravime ich rozdiel
    if not vyvazenie > najlepsie_vyvazenie: # overime, ci sme nenasli lepsie vyvazenie
        najlepsie_vyvazenie = vyvazenie
        best_index = i

print(best_index)
```

Vzorové riešenie

Čo robíme navyše? Stačí si všimnúť dve veci:

1. Najlepší výsledok dostaneme, ak bude hmotnosť prvej tašky najbližšie k polovici z celkovej hmotnosti všetkých predmetov.
2. Nemusíme pre každú možnú pozíciu zmrzliny počítať hmotnosť prvej tašky samostatne. Rozdiel medzi taškou, ktorá obsahuje prvých x prvkov a taškou, ktorá obsahuje prvých $x+1$ prvkov je len jeden predmet. Tiež si všimnime, že ak vieme hmotnosti oboch tašiek (nazvime tieto hmotnosti A a B) a hmotnosť predmetu p, ktorý ideme dať z druhej do prvej tašky, tak hmotnosti týchto tašiek sa zmenia nasledovne: prvej taške stúpne hmotnosť na $A + p$ a druhej klesne na $B - p$.

Čo teda spravíme? Najprv si spočítame súčet hmotností všetkých predmetov. Na začiatku si predstavujeme, že všetky predmety sú v druhej taške. Potom postupne prechádzame cez všetky predmety zľava doprava a zisťujeme, či sa nám oplatí dať predmet z druhej tašky do prvej. Kým sa to oplatí, dávame predmety do prvej tašky až nastane situácia, že pridaním ďalšieho predmetu sa už nič nezlepší. Vtedy môžeme skončiť, lebo pridávaním ďalších predmetov do prvej tašky sa situácia vždy len zhorší. Výsledná časová zložitosť bude $O(n)$, pamäťová ostáva $O(n)$.

Listing programu (C++)

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

int main() {
  int n, hmotnost_celkovo = 0; // pocet predmetov, celkovy sucet hmotnosti predmetov
  cin >> n;
  vector<int> predmety(n);
  for (int i = 0; i < n; ++i) {
```

```
    cin >> predmety[i];
    hmotnost_celkovo += predmety[i];
  }

  int prva_taska = 0, i;
  for (i = 0; i < n; ++i) {
    // ekvivalentne napisane ako: prva_taska - (hmotnost_celkovo - prva_taska)
    int stare_vyvazenie = prva_taska * 2 - hmotnost_celkovo;
    // podobna uvaha akurat pridame predmet do prvej tasky
    int nove_vyvazanie = (prva_taska + predmety[i]) * 2 - hmotnost_celkovo;
    // ak sme pridanim predmetu do tasky nic nezlepsili, vypisme odpoved
    if (abs(nove_vyvazanie) > abs(stare_vyvazenie)) {
      cout << i << endl;
      break;
    }
    prva_taska += predmety[i]; // pridajme predmet do prvej tasky
  }

  if (i == n) { // nezabudnut na pripad, ked su vsetky predmety v prvej taske
    cout << n << endl;
  }

  return 0;
}
```

Iný typ vzorového riešenia

Ukážeme si ešte jedno riešenie, založené na prefixových sumách[1].
Všimnime si, že v našom pomalšom riešení zbytočne sčitujeme hmotnosti predmetov pre každú voľbu zmrzliny. To, čo nás zaujíma, je súčet prvých x predmetov (pre $x \leq n$) a súčet hmotností zvyšných predmetov. Tieto súčty vieme získať v konštantnom čase použítím práve spomínaných prefixových súm. Výsledná časová zložitosť bude $O(n)$, pamäťová ostáva $O(n)$ rovnako ako v prvom vzorovom riešení.

Listing programu (Python)

```python
n = int(input())
predmety = list(map(int, input().split())) # nacitame predmety do pola a skonvertujeme na int-y
best_index = 0 # (zatial) najlepsia volba zmrzliny
najlepsie_vyvazenie = sum(predmety) # ako velmi su tasky vyvazene pri (zatial) najlepsej volbe
prefixove_sumy = [0] # oplati sa nam spravit zarazka, lebo sucet prvych 0 predmetov je 0
for i in range(n): prefixove_sumy.append(predmety[i] + prefixove_sumy[i]) # spocitame prefixove sucty

def sucet(od, po):
    return prefixove_sumy[po] - prefixove_sumy[od] # kedze mame zarazku, tak pocitanie je jednoduche

for i in range(n + 1): # n + 1, lebo chceme overit pripad kedy su vsetky predmety v prvej taske
    prva, druha = sucet(0, i), sucet(i, n) # spravime sucet danych usekov v konstantnom case
    vyvazenie = abs(prva - druha) # potom spravime ich rozdiel
    if not vyvazenie > najlepsie_vyvazenie: # overime, ci sme nenasli lepsie vyvazenie
        najlepsie_vyvazenie = vyvazenie
        best_index = i

print(best_index) # vypiseme odpoved
```

2. O Žer Celaskon

Sandyna (sandyna@ksp.sk)
(max. 12 b za popis, 8 b za program)

Aby sme našli sedadlo pre Zuzku, pre každé voľné sedadlo zistíme vzdialenosť od najbližšieho chorého. Z týchto vzdialeností potom vyberieme tú najlepšiu.

Prvé riešenie

Najjednoduchší spôsob na zisťovanie vzdialeností je, keď pre každé voľné sedadlo prejdeme zoznam všetkých chorých a určíme, ktorý z nich je najbližšie.

Toto riešenie má časovú zložitosť $O(v \cdot c)$. Pamäťová zložitosť bude $O(v + c)$.

Lepšie riešenie

Pri spracovaní jedného voľného sedadla nás ale väčšina chorých nezaujíma. Dôležití sú len tí, ktorí sú najbližsie napravo a naľavo od daného voľného sedadla. Čakáreň si vieme predstaviť ako úseky voľných sedadiel (a sedadiel obsadených zdravými ľuďmi), ktoré sú oddelené chorými. Každý takýto úsek má spoločných najbližších chorých.

Na hľadanie najbližších chorých sa nám oplatí si dáta usporiadať. Vieme potom použiť princíp dvoch bežcov, kedy si v zozname chorých aj v zozname voľných miest pamätáme index sedadla, s ktorým práve pracujeme. V zozname voľných sedadiel si budeme pamätať index práve spracúvaného sedadla a v zozname chorých index

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

najbližšieho chorého napravo od tohto sedadla (rovnako dobre by fungovalo pamätať si index najbližšieho chorého naľavo). Keď v zozname voľných sedadiel presiahneme pozíciu aktuálneho chorého, posunieme sa na ďalší úsek – medzi ďalších dvoch chorých.

V rámci jedného úseku budeme určovať vzdialenosť najbližšieho chorého ako menšiu zo vzdialeností k jednému z chorých na kraji úseku.

Takéto riešenie bude mať časovú zložitosť $O(v \log v + c \log c)$, pretože zoznamy chorých a voľných sedadiel si potrebujeme usporiadať. Prechádzanie týchto zoznamov a určovanie vzdialeností spravíme raz – časová zložitosť je $O(v + c)$ a toto nám celkovú časovú zložitosť nezhorší.

Pamäťová zložitosť bude $O(v + c)$, pretože si potrebujeme pamätať oba zoznamy.

Listing programu (Python)

```
#nacitanie vstupu
v, c = [int(x) for x in input().split()]
volne = [int(x) for x in input().split()]
chore = [int(x) for x in input().split()]

#pridame si choreho do nekonecna na zaciatok a na koniec
chore.append(-100000000)
chore.append(100000000)

#usporiadanie poli
volne = sorted(volne)
chore = sorted(chore)

maximalna_vzdialenost = 0
sucasna_vzdialenos_lava = 0
sucasna_vzdialenos_prava = 0
sucasny_volny_index = 0
#toto bude lavy chory
sucasny_chory_index = 0
vysledne_sedadlo = 0

while (sucasny_chory_index < len(chore) - 1 and sucasny_volny_index < len(volne)):
        if (volne[sucasny_volny_index] > chore[sucasny_chory_index + 1]):
                sucasny_chory_index += 1
        else:
                sucasna_vzdialenost_lava = volne[sucasny_volny_index] - chore[sucasny_chory_index]
                sucasna_vzdialenost_prava = chore[sucasny_chory_index + 1] - volne[sucasny_volny_index]
                if maximalna_vzdialenost < min(sucasna_vzdialenost_lava, sucasna_vzdialenost_prava):
                        vysledne_sedadlo = volne[sucasny_volny_index]
                        maximalna_vzdialenost = min(sucasna_vzdialenost_lava, sucasna_vzdialenost_prava)
                sucasny_volny_index += 1

print(vysledne_sedadlo)
```

Erik (riko@ksp.sk)

3. O Žabovej krutovláde

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

Na začiatok chceme upozorniť, že v tomto vzoráku indexujeme reťazce od nuly, teda reťazec dĺžky n začína indexom 0 a končí indexom $n - 1$.

Počítanie s veľkých číslami

V tejto úlohe budeme často počítať s veľmi veľkými číslami (najväčšie z nich budú rádovo miliónciferné). Počítanie s veľkými číslami je problematické: v mnohých programovacích jazykoch (Java, Pascal, C++) sa nám takéto veľké čísla nezmestia do premennej a aj v jazykoch s neobmedzene veľkými celočíselnými premennými (Python) je výpočet s takýmito veľkými číslami pomalý.

Konečný výsledok (súčet všetkých podreťazcov) však nepotrebujeme vypočítať presne – stačí nám vypočítať jeho zvyšok po delení $10^9 + 7$. Ak pre nejaké celé čísla a, b a kladné celé číslo m chceme vypočítať hodnotu

$$(a + b) \bmod m,$$

nič sa s výsledkom nestane, ak a a b dopredu vymodulujeme m. Formálne povedané, platí:

$$(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m.$$

Podobné tvrdenie platí aj s násobením namiesto sčítania:

$$(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m.$$

Keďže sa všetky naše výpočty budú skladať iba zo sčítaní a násobení, môžeme aj každý medzivýsledok našich výpočtov modulovať číslom $10^9 + 7$. To nám zaručí, že nebudeme musieť narábať s veľkými číslami, nakoľko budú medzivýsledky zaručene menšie ako m.

Hrubá sila

Najjednoduchšie riešenie je postupne prejsť všetky súvislé podreťazce, premeniť si každý z nich zo stringu na celé číslo a to pripočítať k celkovému výsledku.

Pri premene podreťazca na celé číslo môžeme použiť nasledujúci postup. Na začiatku premeníme prvú (najľavejšiu) cifru podreťazca na celé číslo, ktoré si uložíme do premennej. Potom pokračujeme v čítamí nášho podreťazca zľava doprava a pri každej ďalšej cifre najprv prenasobíme našu premennú desiatimi, potom k nej pripočítáme túto cifru podreťazca a nakoniec premennú zmodulujeme $10^9 + 7$.

Každý súvislý podreťazec je jednoznačne určený svojím začiatkom a svojím koncom. Prejdeme teda všetky možné začiatky (v poradí zľava doprava) a pre každý začiatok prejdeme všetky možné konce (tiež v poradí zľava doprava). Dokopy teda musíme spracovať $O(n^2)$ podreťazcov (kde n je dĺžka vstupného reťazca). Pri spracovávaní jedného podreťazca spravíme $O(n)$ operácii, keďže začíname jednociferným číslom a postupne to navyšujeme až na potenciálne n-ciferné číslo. Výsledná časová zložitosť je preto $O(n^3)$. Čo sa týka pamäťovej zložitosti, tak tá je $O(n)$. Máme totiž len jeden reťazec dĺžky n a nejaké pomocné premenné, ktorých je konštantný počet.

Listing programu (C++)

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

#define MOD 1000000007

int main() {
  string str;
  cin >> str;
  int n = str.size();

  long long res = 0;
  for (int l = 0; l < n; l++) {
    for (int r = l; r < n; r++) {
      long long sub = 0;
      for (int i = l; i <= r; i++) {
        sub *= 10;
        sub += str[i] - '0';
        sub %= MOD;
      }
      res += sub;
      res %= MOD;
    }
  }
  cout << res << "\n";

  return 0;
}
```

Trochu porozmýšľajme

Ak trocha porozmýšľame, tak zistíme, že sme veľa vecí počítali zbytočne. Napríklad, ak máme reťazec 12345, tak sme začali podreťazcami 1, 12, 123 atď. Hodnotu každého podreťazca (modulo $10^9 + 7$) sme počítali odznovu, čo je zbytočné. Ak sme už vypočítali hodnotu podreťazca 123 a chceme k celkovému výsledku pripočítať podreťazec 1234, nebudeme to počítať znova od jednotky, ale vynásobíme predchádzajúce číslo 10, pripočítame 4 a zmodulujeme $10^9 + 7$.

Týmto sa nám časová zložitosť zníži na $O(n^2)$, keďže zbytočne nepočítame každé číslo odznova. A pre každý začiatok spracujeme všetky konce v čase $O(n)$. Pamäťová zložitosť ostáva $O(n)$.

Listing programu (C++)

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

#define MOD 1000000007

int main() {
  string str;
  cin >> str;
  int n = str.size();

  long long res = 0;
  for (int l = 0; l < n; l++) {
    long long sub = 0;
    for (int r = l; r < n; r++) {
      sub *= 10;
      sub += str[r] - '0';
      sub %= MOD;
      res += sub;
      res %= MOD;
    }
  }
  cout << res << "\n";
```

```
  return 0;
}
```

Ide to aj lepšie

Vzorové riešenie už vyžaduje trošku iný pohľad na vec. Nebudeme sa snažiť čo najoptimálnejšie prejsť všetky súvislé podreťazce, ale zistíme, akou hodnotou daná cifra na indexe i v reťazci prispieva do konečného výsledku, a tú započítame.

Pozrime sa na príklad, kedy nejaká cifra na i-tej pozícií má pre rôzne súvislé podreťazce, v ktorých sa nachádza, rôzne hodnoty. Napríklad pre reťazec `1532` má cifra `5` v podreťazci `532` hodnotu 500, v podreťazci `53` hodnotu 50 atď.

Všimnime si, že prvá cifra v prípade reťazca `1532` je `1` a môže mať najvyššiu hodnotu 1000. A čím prechádzame na cifry viac vpravo, tým sa nám najvyššia hodnota zmenšuje. Pre cifru `5` máme maximum 500, pre cifru `3` máme maximum 30 atď. Taktiež si ale musíme všimnúť, že keď prechádzame na cifry viac vpravo, tak sa nám aj zvyšuje výskyt každej hodnoty o 1. Vo vyššie uvedenom príklade pre cifru `1` máme hodnoty 1, 10, 100, 1000, všetky iba raz. Pre cifru `5` máme už len hodnoty 500, 50 a 5, ale každú dvakrát, lebo hodnotu 500 máme v podreťazcoch `1532` a `532`, hodnotu 50 v podreťazcoch `153` a `53`, hodnotu 5 v podreťazcoch `15` a `5`. Teraz vieme, že čím ideme viac vpravo tak sa nám maximum najväčšieho možného čísla zmenšuje, ale počet výskytov sa zväčší o 1 pre každú hodnotu.

Takže pre cifru c na pozícii i vypočítame jej prispenie do celkového výsledku ako

$$(i+1) \cdot \underbrace{111\ldots11}_{n-i} \cdot c.$$

Pri výpočte sa nám tým pádom zíde pole veľkosti n, kde na i-tom indexe bude číslo tvorené $(i+1)$ jednotkami. To číslo ale musíme tiež zmodulovať 10^9+7, keďže najvyššie možné môže byť až n-ciferné. Pole vytvoríme jednoducho tak, že definujeme prvý prvok (na indexe 0) ako 1 a ďalšie už počítame ako

$$(10 \cdot \text{predchádzajúci prvok} + 1) \bmod 10^9+7.$$

Zhrnieme si, čo presne robíme. Najprv si vytvoríme pole dĺžky n, kde na i-tom indexe poľa máme číslo tvorené $(i+1)$ jednotkami. Potom prechádzame cyklom celý reťazec a pre každú cifru s pomocou tohto poľa vypočítame jej prispenie do celkového výsledku ako

$$(i+1) \cdot \underbrace{111\ldots11}_{n-i} \cdot c.$$

Medzivýsledky, samozrejme, vždy zmodulujeme 10^9+7.

Časová zložitosť je v tomto prípade len $O(n)$, keďže pomocné pole vieme vytvoriť v čase $O(n)$ a okrem toho stačí len raz prejsť celý reťazec. Pamäťová zložitosť je rovnaká ako pri predchádzajúcich riešeniach, čiže $O(n)$.

Listing programu (C++)

```
#include <iostream>
#include <string>
using namespace std;
#define modulo 1000000007

int main() {

        long long sum=0;
        string cislo;
        cin >> cislo;
        long long pole[cislo.length()];
        pole[0]=1;
        for(int i=1;i<cislo.length();i++)
                pole[i] = (10*pole[i-1]+1) % modulo;

        for(int i=0;i<cislo.length();i++){
                int cifra = ((int)cislo[i])-48;

                sum=(sum+((i+1)*cifra*pole[cislo.length()-1-i])) % modulo;

        }

        cout<<sum % modulo<<endl;
        return 0;
}
```

Úloha ešte má aj riešenie s pamäťovou zložitosťou $O(1)$, za ktoré sa udeľoval jeden bonusový bod. Neuvádzame ho tu, ale záujemcov podporujeme v tom, aby sa naň pokúsili prísť :)

4. Dievčatá sú náročné

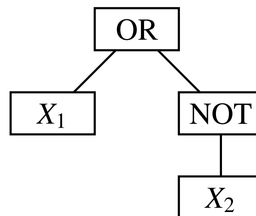
Prefix (prefix@ksp.sk)
(max. 12 b za popis, 8 b za program)

Našou úlohou bolo zistiť, koľkými spôsobmi vieme doplniť premenné tak, aby bol zadaný logický výraz pravdivý. Pri takýchto úlohach je fajn sa ako prvé zamyslieť nad tým, ako sa dá tento výraz reprezentovať v pamäti.

Ako výraz reprezentovať

Výraz obsahuje mnoho zátvoriek a v nich ďalšie výrazy. Každú logickú spojku alebo premennú si vieme reprezentovať ako vrchol v strome. (Ak vám slovíčko "strom" v súvislosti s informatikou nič nehovorí, odporúčam prečítať si časť o stromoch tu[2].) V tomto strome má každý vrchol buď 0, 1 alebo 2 synov—vrcholy reprezentujúce výrazy (*operandy*), ktoré sú v jeho zátvorkách.

Napríklad `OR(X_1, NOT(X_2))` reprezentujeme vrcholom pre `OR`, ktorý má ako synov vrcholy `X_1` a `NOT`. Tento `NOT` má syna `X_2`.



Otázka je, ako z textovej reprezentácie výrazu dostaneme takýto strom? Budeme ho zostrojovať rekurzívne, od koreňa k listom. Predstavme si, že text čítame zľava doprava. Na začiatku sa určite nachádza jedna zo značiek `AND`, `OR`, `NOT` alebo `X`. Podľa toho, ktorý z týchto prípadov nastal, vieme, čo máme ďalej v texte očakávať:

- Ak sme prečítali `AND` alebo `OR`, bude nasledovať `(`, výraz, `,`, výraz, `)`. Takže stačí rekurzívne načítať prvý operand, prečítať čiarku `,`, načítať druhý operand a prečítať koncovú zátvorku `)`. Vrátime vrchol príslušného typu, ktorého synovia sú načítané operandy.
- Ak sme prečítali `NOT`, postupujeme podobne. Jediný rozdiel je v tom, že máme len jeden operand a nemusíme prečítať čiarku.
- Ak sme prečítali `X`, vrátime vrchol reprezentujúci premennú.

Keď máme načítaný vstup a uložený v strome, vieme sa vrhnúť na prvé riešenie.

Ako vieme nájsť ohodnotenie výrazu

Všimnime si, že ak máme vyhodnotených všetkých synov nejakého vrcholu, tak vieme v konštantnom čase zistiť ohodnotenie tohto vrcholu. Toto tvrdenie nie je nejako závratné, spočíva len v tom, že spočítame hodnotu logickej funkcie v tomto vrchole. Z tohto tvrdenia ale vyplýva dôležitejšie—predstavme si, že poznáme hodnoty všetkých listov (premenných). Potom vieme v lineárnom čase zistiť hodnotu celého výrazu.

Spravíme to takto: ak chceme vyhodnotiť nejaký vrchol, vyhodnotíme najprv všetkých jeho synov a následne v konštantnom čase vyhodnotíme logickú funkciu z hodnôt synov. Vieme, že listy už máme ohodnotené, takže ak to budeme vyhodnocovať rekurziou, má kde zastať. Takémuto algoritmu to bude trvať $O(n)$, kde n bude počet vrcholov. Toto môžeme jednoducho odhadnúť ako dĺžku vstupu. Pre každý vrchol spravíme iba konštantný počet operácii.

Nás zaujíma počet takých ohodnotení, pri ktorých je vstupný výraz pravdivý. Môžeme vyskúšať všetky možné ohodnotenia, pre každé zistiť hodnotu celého výrazu a spočítať tie, ktoré sú pravdivé. Všetkých možností ohodnotenia máme najviac 2^n a teda časová zložitosť riešenia je $O(n \cdot 2^n)$.

Týmto riešením mohol váš program získať 2 body.

Vzorové riešenie

Skúsme si náš problém vyriešiť pre najmenšie možné stromy.

Pre strom skladajúci sa z jedného `X` máme práve jednu možnosť ako môže byť pravdivý: ak za `X` dosadíme 1.

Pokiaľ máme strom `NOT(X)`, je práve jedna možnosť ako môže byť pravdivý: ak `X` je nepravdivé. Všeobecne, počet možností ako môže byť `NOT(výraz)` pravdivý je počet možností, ako môže byť `výraz` nepravdivý. Pre strom `OR(X,X)` máme tri dosadenia, ako získať pravdivý výraz: $(0,1)$, $(1,0)$ a $(1,1)$. Pre strom `AND(X,X)` je zasa len jedno: $(1,1)$.

[2] https://www.ksp.sk/kucharka/grafy_uvod/

Pri vyhodnocovaní `NOT` sme si všimli zaujímavú vec: ak by sme vedeli, koľkými spôsobmi môžu byť jeho synovia nepravdivý, vedeli by sme hneď povedať, koľkými spôsobmi môže byť `NOT` pravdivý. Myšlienka, ktorá nás môže napadnúť je, či sa toto nedá použiť aj pre ostatné typy vrcholov.

Poďme teda skúsiť vyhodnotiť `AND(výraz1, výraz2)` s tým, že pre `výraz1` aj `výraz2` vieme, koľkými spôsobmi môžu byť pravdivé. Označme tieto počty postupne p_1 a p_2. Hľadáme počet možností, v ktorých sú aj `výraz1` aj `výraz2` pravdivé. Tí, čo už mali trocha kombinatoriky hneď vidia, že to je $p_1 \cdot p_2$—pre každú možnosť, v ktorej je `výraz1` pravdivý, existuje p_2 možností, v ktorých je aj `výraz2` pravdivý.

Kebyže vieme, koľkými spôsobmi možu byť `výraz1` a `výraz2` nepravdivé, vieme zistiť dokonca aj to, koľkými spôsobmi môže byť `AND(výraz1, výraz2)` nepravdivý. Označme si teda n_1 a n_2 počty nepravdivých možností pre vyrazy 1 a 2. `AND(výraz1, výraz2)` je nepravdivý v troch prípadoch

- `výraz1` je nepravdivý a `výraz2` je nepravdivý: $n_1 \cdot n_2$ možností
- `výraz1` je pravdivý a `výraz2` je nepravdivý: $p_1 \cdot n_2$
- `výraz1` je nepravdivý a `výraz2` je pravdivý: $n_1 \cdot p_2$

Teda `AND` je nepravdivý v toľkoto prípadoch:

$$n_1 \cdot n_2 + p_1 \cdot n_2 + n_1 \cdot p_2.$$

Rovnakým spôsobom vieme vymyslieť, ako z počtov možností detí vrcholu `OR` zistiť počty možností samotného vrcholu `OR`.

Takže ak vieme, koľkými spôsobmi vedia byť deti nejakého vrcholu pravdivé a koľkými spôsobmi vedia byť nepravdivé, vieme zistiť aj pre tento samotný vrchol rovnakú informáciu. Použijeme pri tom len pomocou pár násobení v prípade `AND` a `OR`, a výmenou hodnôt v prípade `NOT`. A o listoch stromu vieme, že je práve jedna možnosť ako môžu byť pravdivé, a práve jedna ako môžu byť nepravdivé.

Tu si zoberieme myšlienku z brute forcu—keď sme chceli zistiť vyhodnotenie stromu s nejakými hodnotami vrcholov, rekurzívne sme ho prešli a pre každý vrchol sme z jeho synov zistili jeho hodnotu. V našom riešení môžme spraviť to isté, len pre každý vrchol zo synov zistíme, koľkými spôsobmi môže byť pravdivý a koľkými nepravdivý. Riešenie bude potom počet pravdivých spôsobov pre koreň.

Časová zložitosť je $O(n)$, keďže pre každý vrchol v konštantom čase zistíme jeho hodnotu. Pamäťová je $O(n)$, nakoľko si pamätáme celý strom.

Listing programu (C++)

```cpp
#include <string>
#include <cstdio>
#include <cstdlib>
#include <iostream>

const long long MOD = 1000000007L;
using namespace std;

typedef long long ll;

enum typ{
    PREM,
    OR,
    AND,
    NOT
};

struct Node{
    typ t;
    Node * c[2];
    ll fa = 0;
    ll tr = 0;
};

Node * parse(const string &s, int & pos){
    Node * x = new Node;
    if(s[pos] == 'X'){
       x-> t = PREM;
       pos += 1;
    }
    else if(s[pos] == 'A'){
       x->t = AND;
       pos += 4;
       x->c[0] = parse(s,pos);
       pos+=1;
       x->c[1] = parse(s,pos);
       pos +=1;
    }
    else if(s[pos] == 'O'){
       x->t = OR;
       pos += 3;
       x->c[0] = parse(s,pos);
       pos+=1;
```

```
        x->c[1] = parse(s,pos);
        pos +=1;

    }
    else if(s[pos] == 'N') {
        x->t = NOT;
        pos += 4;
        x->c[0] = parse(s,pos);
        pos +=1;
    }
    return x;
}

void pocetMoznosti(Node * x){
    switch(x->t){
        case PREM:
            x->tr = 1;
            x->fa = 1;
            break;
        case OR:
            pocetMoznosti(x->c[0]);
            pocetMoznosti(x->c[1]);
            x->fa = x->c[0]->fa * x->c[1]->fa;
            x->tr = x->c[0]->tr*(x->c[1]->tr + x->c[1]->fa) + x->c[0]->fa *x->c[1]->tr;
            break;
        case AND:
            pocetMoznosti(x->c[0]);
            pocetMoznosti(x->c[1]);
            x->tr = x->c[0]->tr * x->c[1]->tr;
            x->fa = x->c[0]->fa * (x->c[1]->tr + x->c[1]->fa) + x->c[0]->tr*x->c[1]->fa;
                        x->tr %= MOD;
                        x->fa %= MOD;
            break;
        case NOT:
            pocetMoznosti(x->c[0]);
            x->tr = x->c[0]->fa;
            x->fa = x->c[0]->tr;
    }
        x->tr %= MOD;
        x->fa %= MOD;
}


int main(){
    string s;
    cin >> s;

    int parser=0;
    Node * x = parse(s,parser);
    pocetMoznosti(x);
    printf("%lld\n",x->tr);
}
```

5. Insektológia v Slovakistane

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

Stará dobrá hrubá sila

Ako inak, úloha sa dá riešiť aj hrubou silou. Pre každú otázku prejdeme zadaný interval, každé číslo delíme daným prvočíslom kým to len ide, a k odpovedi pripočítame rozdiel pôvodného a výsledného čísla.

Aká je časová zložitosť? Pre každú otázku prejdeme najviac n čísel a každé z nich niekoľkokrát vydelíme– najviac ale logaritmicky veľakrát, a keďže všetky čísla v poli sú nanajvýš p, budeme deliť $O(\log p)$-krát.

Každá otázka nám teda zaberá $O(n \log p)$ času, a teda q otázok zodpovedáme v $O(qn \log p)$, a tešíme sa, že prejdeme prvou sadou. Otázky môžeme riešiť po jednej, a teda si okrem zamorení polí pamätáme len konštantne veľa premenných; pamäťová zložitosť je teda $O(n)$.

Listing programu (C++)

```
#include <iostream>

using namespace std;

int main()
{
        int n,q,p;
        cin >> n >> q >> p;
        int polia[n];
        for(int i=0;i<n;++i)
                cin >> polia[i];

        while(q--)
        {
                int l,r,p;
                cin >> l >> r >> p;
                int res = 0;
                for(int i=l-1;i<r;++i)
                {
```

```
                        int x = polia[i];
                        while(x%p==0)
                                x /= p;
                        res += polia[i]-x;
                }
                cout << res << "\n";
        }
}
```

Všetko si predpočítame

Chceli by sme zrýchliť odpovedanie na otázky, nakoľko $O(n \log p)$ si okrem prvej sady nemôžeme nikde dovoliť.

Ako by vyzerala úloha, keby každá otázka bola o rovnakom prvočísle x? Každá kontaminácia poľa v_i by (ak je v intervale otázky) prispela nejakou hodnotou x_i, ktorá sa nikdy nezmení. Každá otázka by teda bola priamočiaro 'súčet čísel v intervale', ktoré vieme v $O(1)$ zodpovedať pomocou prefixových súčtov[3].

Môžeme si teda predpočítať prefixový súčet hodnôt x_i pre každé prvočíslo x menšie rovné p. Takýto prístup je dosť efektívny na druhú sadu, kde p je malé.

Pre ilustráciu, hodnoty x v príkladovom vstupe by vyzerali takto:

		index				
		1	2	3	4	5
prvočísla	2	5	15	15	35	25
	3	0	0	20	0	0
	5	8	16	24	32	48
	7	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮
	97	0	0	0	0	0

A k nim prislúchajúce prefixové súčty:

		index					
		0	1	2	3	4	5
prvočísla	2	0	5	20	35	70	95
	3	0	0	0	20	20	20
	5	0	8	24	48	80	128
	7	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	97	0	0	0	0	0	0

Aká je teda časová zložitosť? V čase $O(p^{3/2})$ vieme nájsť všetky prvočísla do p—iba pre každé číslo $2, 3, \dots, p$ overíme, či má deliteľa menšieho/rovného jeho odmocnine. Potom pre každé prvočíslo vypočítame hodnoty x. Všetkých hodnôt x je $O(np)$[4]. Na výpočet každej z hodnôt x potrebujeme $O(\log p)$-krát deliť, teda toto predpočítanie nám potrvá $O(np \log p)$. Otázky už zodpovedáme v $O(1)$. Časová zložitosť je teda $O(np \log p + q)$.

Pamäťová zložitosť je $O(np)$, nakoľko si pamätáme prefixové súčty hodnôt x.

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;

vector<int> najdi_prvocisla(int MAX)
{
    vector<int> prvocisla;
    for(int i=2;i<=MAX;++i)
    {
        bool ok = true;
        for(int j=2;ok && j*j<=i;++j)
            if(i%j==0)
                ok = false;

        if(ok) prvocisla.push_back(i);
```

[3] https://www.ksp.sk/kucharka/prefixove_sumy/

[4] Dá sa dokázať, že prvočísel v $1, \dots, p$ je $O(\frac{p}{\log p})$, a tým sa dá odhad vylepšiť na $O(n \cdot \frac{p}{\log p})$. Záujemcovia môžu navštíviť https://en.wikipedia.org/wiki/Prime_number_theorem

```
    }
    return prvocisla;
}

int main()
{

    int n,q,p;
    cin >> n >> q >> p;

    if(n<=1500 && q<=1500)
    {
        //riesenie hrubou silou
        return 0;
    }

    vector<int> prvocisla = najdi_prvocisla(p);
    vector<int> ocislovanie(p+1);
    for(int i=0;i<prvocisla.size();++i)
        ocislovanie[ prvocisla[i] ] = i;

    vector<vector<int>> prefix(prvocisla.size(),{{0}});
    for(int i=0;i<n;++i)
    {
        int v;
        cin >> v;
        for(int j=0;j<prvocisla.size();++j)
        {
            int tmp = v;
            while(tmp%prvocisla[j] == 0)
                tmp /= prvocisla[j];
            prefix[j].push_back(v-tmp+prefix[j][i]);
        }
    }

    while(q--)
    {
        int l,r,pi;
        cin >> l >> r >> pi;
        pi = ocislovanie[pi];
        cout << prefix[pi][r] - prefix[pi][l-1] << "\n";
    }
}
```

Predpočítame len nenulové prvky

Vo vyššieuvedenom riešení už odpovedáme na otázky dosť rýchlo, pridlho nám však trvá predpočítanie prefixových polí. Máme v nich priveľa prvkov, potrebujeme ich ale všetky?

Musíme si uvedomiť, že tak, ako v čísle veľkosti p môže byť prvočíslo umocnené na najviac $\log p$, tak isto v ňom môže byť najviac $\log p$ **rôznych** prvočísel. Príkladne pre $p = 10^6$ máme 78498[5] prvočísel, avšak v konkrétnom čísle do 10^6 ich vie najviac byť len 7[6]! Pre každé z n čísel nám teda reálne najviac $\log p$ hodnôt x dodalo nejakú podstatnú informáciu, zvyšné sú nutne nuly. Stačí nám teda uvažovať len týchto $n \log p$ hodnôt, a pre každú si navyše pamätáme, ktorému indexu zodpovedá.

Pre ilustráciu, hodnoty x v príkladovom vstupe by sa skomprimovali takto:

prvočísla					
2	1 : 5	2 : 15	3 : 15	4 : 35	5 : 25
3	3 : 20				
5	1 : 8	2 : 16	3 : 24	4 : 32	5 : 48

(Prvé číslo je index i, druhé je hodnota x_i.) Keď máme (komprimované) hodnoty x, ľahko k nim zostrojíme (komprimované) prefixové súčty. V príkladovom vstupe be sme dostali:

prvočísla						
2	0 : 0	1 : 5	2 : 20	3 : 35	4 : 70	5 : 95
3	0 : 0	3 : 20				
5	0 : 0	1 : 8	2 : 24	3 : 48	4 : 80	5 : 128

Žiadne ďalšie prvočísla v číslach na vstupe neboli, teda zvyšné prefixové súčty sú (efektívne) prázdne.

Keď už poznáme prefixové súčty, na každú otázku vieme odpovedať nasledovne. Ak chceme súčet intervalu $\langle l_i, r_i \rangle$, nájdeme v príslušnom prefixovom poli prvé $L_i \geq l_i$ a posledné $R_i \leq r_i$, a odpovieme súčtom v $\langle L_i, R_i \rangle$ (ktorý je rovnaký). Tieto dve hodnoty vieme binárne vyhľadať v $O(\log n)$, zodpovedanie všetkých otázok nám teda bude trvať $O(q \log n)$.

Ostáva teda len jedna otázka: ako rýchlo si vieme hodnoty x predpočítať?

[5] http://www.wolframalpha.com/input/?i=pi+function(10%5E6)

[6] $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 = 510\,510$. Použili sme tu čo najmenšie prvočísla, a ôsme už do milióna nezmestíme.

Budeme postupovať podobne, ako pri Eratostenovom site[7]. Upravíme ho tak, aby nám navyše pre každé číslo našlo najmenšie prvočíslo, ktoré ho delí—vždy, keď zistíme, že nejaké q je prvočíslo, navštívime všetky násobky q. Všetky násobky, ktoré sme doteraz nenavštívili, určite nie sú deliteľné žiadnym prvočíslom menším ako q, ich najmenší prvočíselný deliteľ je teda q.

Tak, ako pri štandardnom Eratostenovom site, stačí nám navštíviť len tie násobky, ktoré sú väčšie alebo rovné ako q^2. Menšie násobky sú totiž deliteľné aj nejakým prvočíslom menším ako q, ktoré sme už spracovali—a teda už poznáme najmenšieho prvočíselného deliteľa takého násobku.

Toto spravíme v čase $O(p \log \log p)$ a pamäti $O(p)$.

Keď teraz chceme zistiť prvočíselný rozklad nejakého v_i, budeme ho zakaždým deliť jeho najmenším prvočíselným deliteľom, až kým nedosiahne hodnotu 1. Pritom si pamätáme, ktorými prvočíslami sme ho vydelili a koľkokrát. Každé zistenie rozkladu nám bude trvať $O(\log p)$.

Z rozkladu v_i následne zostrojíme komprimovanú tabuľku hodnôt x v čase $O(n \log p)$, a zostrojíme komprimované prefixové súčty v čase lineárnom od veľkosti tabuľky, čiže tiež $O(n \log p)$. Nakoniec v $O(q \log n)$ zodpovieme otázky.

Spolu s Eratostenovým sitom dostávame časovú zložitosť $O(p \log \log p + n \log p + q \log n)$. Pamäťová zložitosť je $O(p + n \log p)$, nakoľko si pamätáme pole s najmenšími prvočíselnými deliteľmi, a prefixové súčty hodnôt x.

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;

#define NEZNAMY -1

int p;
vector<int> ocislovanie;
vector<int> min_prvodelitel;
vector<int> prvocisla;

vector<vector<pair<int,long long> > > prefixy;

void Eratosten()
{
    for(int i=2;i<=p;++i)
    {
        if(min_prvodelitel[i] != NEZNAMY)
            continue;

        prvocisla.push_back(i);
        ocislovanie[i] = prvocisla.size()-1;
        min_prvodelitel[i] = i;

        if((long long)i*i >= p) continue;

        for(int k=i*i;k<=p;k+=i)
            if (min_prvodelitel[k] == NEZNAMY)
                min_prvodelitel[k] = i;
    }

    prefixy.resize(prvocisla.size(),{{0,0}});
}

int main()
{
        ios::sync_with_stdio(false);
        cin.tie(0);
        cout.tie(0);

    int n,q;
    cin >> n >> q >> p;

    min_prvodelitel.resize(p+1, NEZNAMY);
    ocislovanie.resize(p+1);

    // najdeme prvocisla <= p
    // a pre cisla <= p ich najmensich prvociselnych delitelov
    Eratosten();

    for(int i=0;i<n;++i)
    {
        int v,tmp;
        cin >> v;
        tmp = v;

        // najdeme prvociselnych delitelov v
        vector<int> prvodelitelia;
        while(tmp != 1)
        {
            int pdel = min_prvodelitel[tmp];
            if (prvodelitelia.empty() || pdel != prvodelitelia.back())
                prvodelitelia.push_back(pdel);
            tmp /= pdel;
        }
```

[7] https://www.ksp.sk/kucharka/eratosten/

```
        // pre kazdeho prvociselneho delitela v najdeme prislusnu hodnotu x
        for(int k=0;k<(int)prvodelitelia.size();++k)
        {
            int pdel = prvodelitelia[k];
            tmp = v;
            while(tmp % pdel == 0)
                tmp /= pdel;

            // pridame prvok do prefixoveho suctu s indexom cisla, a kolko urody zachrani
            prefixy[ocislovanie[pdel]].push_back({i+1,v-tmp});
        }
    }

    for(int i=0;i<prvocisla.size();++i)
    {
        for(int k=1;k<prefixy[i].size();++k)
            prefixy[i][k].second += prefixy[i][k-1].second;
    }

    while(q--)
    {
        int l,r,p;
        cin >> l >> r >> p;
        p = ocislovanie[p];

        //L = posledny prefix s indexom < l, R = posledny prefix s indexom <= r
        int L,R;


        int low=0,high=prefixy[p].size();
        // binarne vyhladame posledny prefix s indexom < l
        while(high-low>1)
        {
            int mid = (high+low)/2;

            if(prefixy[p][mid].first < l)
                low = mid;
            else high = mid;
        }

        L = low;

        low=0,high=prefixy[p].size();
        // binarne vyhladame posledny prefix s indexom <= r
        while(high-low>1)
        {
            int mid = (high+low)/2;

            if(prefixy[p][mid].first <= r)
                low = mid;
            else high = mid;
        }

        R = low;

        cout << prefixy[p][R].second - prefixy[p][L].second << "\n";
    }
}
```

6. Šialená jazda

Dávid (davidb@ksp.sk)
(max. 12 b za popis, 8 b za program)

Pred čítaním vzorového riešenia odporúčame prečítať si v KSP Kuchárke články o geometrii, konkrétne článok o skalárnom súčine[8] a článok o konvexnom obale[9].

Konvexný obal

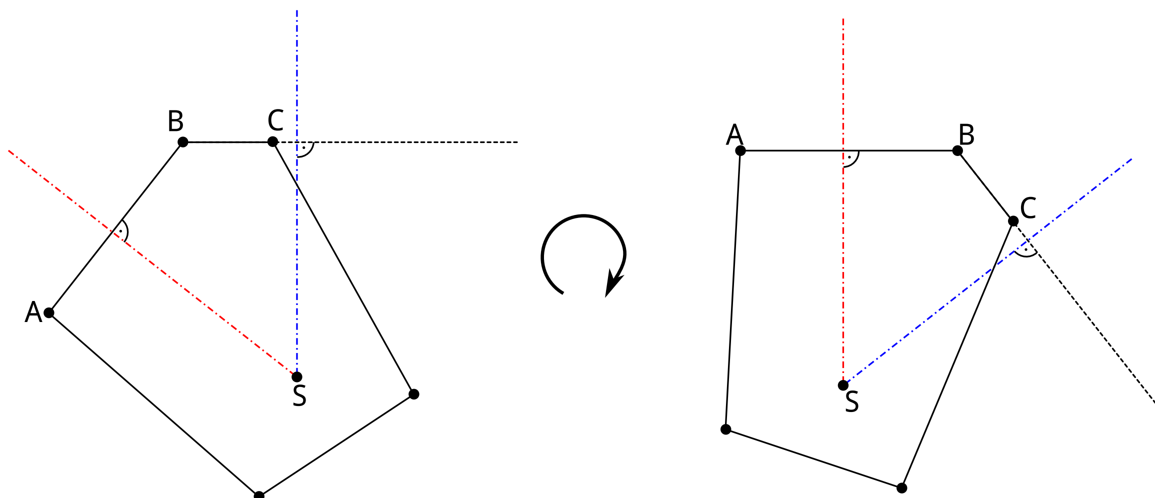
Prvé pozorovanie potrebné na vyriešenie tejto úlohy je nasledujúce: na to, aby sa mohla kabínka dostať nad všetky ostatné, musí ležať na konvexnom obale všetkých kabíniek. Ak neleží, znamená to, že v každom momente je nad ňou nejaká hrana konvexného obalu. Aspoň jeden koniec tejto hrany je vtedy vyššie ako daná kabínka. Na začiatku si teda môžeme zostrojiť konvexný obal všetkých kabíniek a tie kabínky, ktoré nie sú jeho vrcholmi, môžeme rovno zahodiť.

Najlepší bod

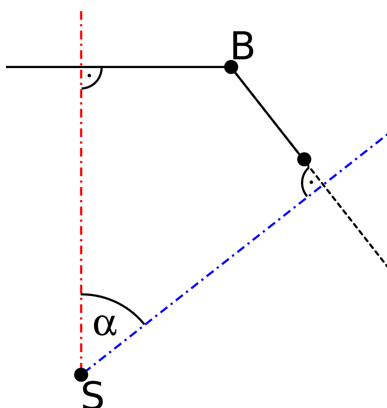
Keď už máme konvexný obal, stačí nám o každom jeho vrchole zistiť, ako dlho bude najvyššie. Vezmime si ľubovoľný vrchol B z nášho konvexného obalu. Vrchol, ktorý je na obale pred vrcholom B označme A a nasledujúci vrchol na obale označme C. Stred, okolo ktorého sa celý kolotoč točí, označme S. Bod B začína byť najvyššie vtedy, keď je hrana BC vo vodorovnej polohe, a prestáva vtedy, keď sa do vodorovnej polohy dostane hrana AB. Uhol, o ktorý sa kolotoč otočí medzi týmito dvoma stavmi, je uhol ktorý zvierajú kolmice na tieto

[8] https://www.ksp.sk/kucharka/skalarny_a_vektorovy_sucin/
[9] https://www.ksp.sk/kucharka/konvexny_obal

strany prechádzajúce bodom S (na obrázku červená a modrá priamka). Kolotoč sa otáča konštantnou *uhlovou* rýchlosťou, veľkosť tohto uhla je teda priamo úmerná času, počas ktorého bude bod B najvyššie. Najlepší je teda bod, ktorý má tento uhol maximálny.



Uhol, ktorý sa snažíme maximalizovať, si označme α. Zo súčtu vnútorných uhlov štvoruholníka na nasledujúcom obrázku vyplýva, že uhol $|\measuredangle ABC|$ je rovný $180° - \alpha$. Preto nám stačí nájsť taký vrchol konvexného obalu, pri ktorom je najmenší vnútorný uhol. Vnútorné uhly konvexného mnohouholníka sú vždy menšie než $180°$. Z toho vyplýva, že veľkosť vnútorného uhla je jednoznačne určená jeho kosínusom. Teraz nám stačí nájsť vrchol, pri ktorom je vnútorný uhol s najväčším kosínusom. A kosínus vnútorného uhla ľahko vyrátame pomocou skalárneho súčinu.



Časová a pamäťová zložitosť

Časová zložitosť výpočtu konvexného obalu je $O(n \log n)$. Po tomto výpočte už len v lineárnom čase prejdeme body z konvexného obalu a nájdeme najlepší z nich. Pamäťová zložitosť je $O(n)$.

Listing programu (C++)

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <cmath>

using namespace std;

struct Point {
    long long x, y;

    bool operator < (Point p){
        if(x == p.x)
            return(y < p.y);
        else
            return(x < p.x);
    }

    Point(long long a, long long b){
        x = a;
```

```
        y = b;
    }

    Point(){
        x = 0;
        y = 0;
    }
};

// kladné ak sa OAB točí doľava
long long cross(Point O, Point A, Point B){
    return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

// dot product, ktory v sebe skrýva kosínus uhla
long long dot(Point O, Point A, Point B){
    return (A.x - O.x) * (B.x - O.x) + (A.y - O.y) * (B.y - O.y);
}

double length(Point O, Point A){
    return sqrt(pow(A.x - O.x, 2) + pow(A.y - O.y, 2));
}

bool smallerAngle(int i, int j, vector<Point> &hull){
    Point a1 = hull[i-1];
    Point c1 = hull[i];
    Point b1 = hull[i+1];

    Point a2 = hull[j-1];
    Point c2 = hull[j];
    Point b2 = hull[j+1];

    // porovnáme kosínus uhla a1,c1,b1 a a2,c2,b2 (delenie upravíme na násobenie)
    // (a1.b1)/(|a1|*|b1|) > (a2.b2)/(|a2|*|b2|)
    // (a1.b1)*(|a2|*|b2|) > (a2.b2)*(|a1|*|b1|)
    double l = dot(c1, a1, b1) * (length(c2, a2) * length(c2, b2));
    double r = dot(c2, a2, b2) * (length(c1, a1) * length(c1, b1));

    if(l > r || (l == r && c1 < c2))
        return true;
    else
        return false;
}

// vráti body na konvexnom obale v ľavo točivom smere (prvý bod je rovnaký ako posledný)
vector<Point> convex_hull(vector<Point> &points){
    vector<Point> hull;

    // zoradíme body od ľava
    sort(points.begin(), points.end());

    // Dolná časť obalu
    for(int i = 0; i < points.size(); i++){
        while(hull.size() >= 2 && cross(hull[hull.size()-2], hull[hull.size()-1], points[i]) <= 0)
            hull.pop_back();
        hull.push_back(points[i]);
    }

    // horná časť obalu
    int z = hull.size() + 1; //začiatok dolnej časti
    for(int i = points.size() - 2; i >= 0; i--){ // -2 aby tam nebol krajný bod dva krát
        while(hull.size() >= z && cross(hull[hull.size()-2], hull[hull.size()-1], points[i]) <= 0)
            hull.pop_back();
        hull.push_back(points[i]);
    }

    return hull;
}

int main(){
    int n;
    cin >> n;
    vector<Point> points;

    for(int i = 0; i < n; i++){
        long long a, b;
        cin >> a >> b;
        points.push_back(Point(a, b));
    }

    if(n == 1){
        cout << points[0].x << " " << points[0].y << endl;
        return 0;
    }

    if(n==2){
        sort(points.begin(), points.end());
        cout << points[0].x << " " << points[0].y << endl;
        return 0;
    }

    vector<Point> hull = convex_hull(points);
    hull.push_back(hull[1]);

    int where = 1;
    for(int i = 1; i < hull.size() - 1; i++)
        if(smallerAngle(i, where, hull)) where = i;
```

```
    cout << hull[where].x << "␣" << hull[where].y << endl;
}
```

7. Lamborghini? Bicykel!

Mišo (faiface@ksp.sk)
(max. 12 b za popis, 8 b za program)

Každému skúsenému riešiteľovi pravdepodobne napadlo neoptimálne riešenie tejto úlohy hneď po prečítaní zadania. K optimálnemu riešeniu potom už ostáva si len uvedomiť jednu skutočnosť a je to tam.

Poďme sa pozrieť na obe riešenia.

Neoptimálne riešenie – Dijkstra

Askar sa musí dostať zo svojho domu do budovy Konferencie o Sprevinilej Planéte, čo najrýchlejšou cestou. Najznámejším algoritmom na hľadanie najrýchlejšej cesty je starý dobrý Dijkstra. Tu si o ňom môžete prečítať.[10]

V našom prípade ho však nemôžeme aplikovať úplne priamočiaro – Askar môže totiž ísť pešo aj na bicykli. Tento problém sa dá ľahko vyriešiť: pôvodný graf si trochu prerobíme. Každý vrchol i nahradíme dvoma vrcholmi: i_p (pešo) a i_b (bicykel). Vrchol i_b reprezentuje situáciu, keď je Askar vo vrchole i a má so sebou bicykel, vrchol i_p zasa situáciu, keď je Askar vo vrchole i bez bicykla. Ak v pôvodnom grafe bola hrana medzi vrcholmi i a j, tak v novom grafe bude hrana $i_p - j_p$ s dĺžkou k a hrana $i_b - j_b$ s dĺžkou 1. Čiže, ak bol Askar pešo, tak môže pešo prejsť do druhého vrcholu a podobne, ak bol na bicykli, tak na ňom môže pokračovať. V prípade, že vrchol i je bike-sharing stanica, tak pridáme ešte dve ďalšie hrany: $i_p - j_b$ s dĺžkou 1 (Askar po tejto hrane už prejde na bicykli) a $i_b - j_p$ s dĺžkou k (Askar zosadol z bicykla a túto hranu prejde pešo).

Takto upravený graf má $2n$ vrcholov a nanajvýš $6m$ hrán (v najhoršom prípade, ak by skoro všade boli bike-sharing stanice, budeme mať šesť hrán za každú pôvodnú hranu).

V skutočnej implementácii nemusíme ani konštruovať nový graf. Stačí si v halde okrem čísla vrcholu pamätať aj to, či je Askar pešo alebo na bicykli a podľa toho akumulovať vzdialenosti.

Časová zložitosť tohoto riešenia je $O(m \log m)$ a pamäťová $O(n + m)$.

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;

struct dijkstra
{
    int stav,vrchol;
    long long cena;
};

struct cmp
{
    bool operator()(const dijkstra&a,const dijkstra&b) const
    {
        return a.cena > b.cena;
    }
};

vector<bool> bicykel;
vector<vector<int> > graf;
vector<long long> dist[2];

int main()
{
        ios::sync_with_stdio(false);
        cin.tie(0);
        cout.tie(0);

    int n,coef,m;
    cin >> n >> coef >> m;

    graf.resize(n);

    while(m--)
    {
        int x,y;
        cin >> x >> y;
        --x,--y;
        graf[x].push_back(y);
        graf[y].push_back(x);
    }

    bicykel.resize(n,false);
    int b;
    cin >> b;
    while(b--)
    {
```

[10] https://www.ksp.sk/kucharka/dijkstra/

```
        int c;
        cin >> c;
        bicykel[c-1] = true;
    }

    for(int i=0;i<2;++i)
        dist[i].resize(n,(long long)n*coef);

    dist[0][0] = 0;
    priority_queue<dijkstra,vector<dijkstra>,cmp> pq;

    pq.push({0,0,0});

    while(!pq.empty())
    {
        dijkstra d = pq.top();
        pq.pop();
        if(dist[d.stav][d.vrchol] < d.cena) continue;

        for(int i=0;i<graf[d.vrchol].size();++i)
        {
            int kam = graf[d.vrchol][i];
            long long nova_cena = d.cena + 1 + (coef-1)*(d.stav==false);

            if(dist[d.stav][kam] > nova_cena)
            {
                dist[d.stav][kam] = nova_cena;
                pq.push({d.stav,kam,nova_cena});
            }
        }

        if(bicykel[d.vrchol])
        {
            bool novy_stav = !d.stav;
            if(dist[novy_stav][d.vrchol] > d.cena)
            {
                dist[novy_stav][d.vrchol] = d.cena;
                pq.push({novy_stav,d.vrchol,d.cena});
            }
        }
    }

    cout << dist[0][n-1] << "\n";
}
```

Optimálne riešenie

Intuícia nám napovedá, že predchádzajúce riešenie nie je optimálne. Dijkstra funguje na hocijaké grafy, ale náš graf nie je hocijaký: všetky jeho hrany sú dĺžky 1 alebo k. Toto sa musí dať využiť.

A naozaj to ide. Všeobecnú logaritmickú haldu z predchádzajúceho riešenia vieme nahradiť dvoma frontami. Do prvej fronty budeme vkladať vrcholy, keď je Askar na bicykli a do druhej keď je pešo. Ako si o chvíľu ukážeme, obe fronty ostanú vždy usporiadané podľa vzdialenosti, a teda keď chceme vybrať nasledujúci najbližší vrchol (tak, ako to robíme v Dijkstrovi), stačí nám vybrať menší (bližší) z vrcholov na začiatku oboch front.

Tak ako sme si už povedali, obe fronty musia vždy ostať usporiadané, aby naše riešenie fungovalo – inak by sme sa nevedeli spoľahnúť, že nasledujúci vrchol sa nachádza na začiatku niektorej z nich. Dôkaz je veľmi ľahký. Predpokladajme, že doteraz sú fronty usporiadané. Z dvoch začiatkov teda vyberieme menší vrchol, povedzme, že tento vrchol je vo vzdialenosti 15. V prípade, že z neho ideme pokračovať peši, do fronty pre pešie vrcholy vložíme vrchol so vzdialenosťou $15 + k$. Aby fronta ostala usporiadaná, nesmel sa v nej nachádzať žiaden vzdialenejší vrchol. Mohol sa? Nemohol: keďže Dijkstrov algoritmus navštivuje vrcholy podľa vzdialenosti (a doteraz boli fronty usporiadané, takže bežal korektne), všetky vrcholy, ktoré sme doteraz navštívili, boli vo vzdialenosti 15 alebo menšej. Najvzdialenejší vrchol, ktorý sme doteraz mohli do pešej fronty pridať, mohol mať vzdialenosť najviac $15 + k$, čo nie je viac, než náš naposledy pridaný vrchol. Fronta teda ostane usporiadaná. Dôkaz pre druhú frontu je úplne rovnaký.

Fronty nám pomohli vyhnúť sa logaritmu, časová zložitosť tohoto riešenia je teda $O(n + m + s) = O(m)$ a pamäťová je rovnaká. Keďže na vyriešenie úlohy určite musíme minimálne načítať vstup, lepšiu asymptotickú časovú zložitosť sa už dosiahnuť nedá.

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;

struct dijkstra
{
    int stav,vrchol;
    long long cena;
};

vector<bool> bicykel;
vector<vector<int> > graf;
vector<long long> dist[2];
```

```
queue<dijkstra> q[2];

bool qempty()
{
    return q[0].empty() && q[1].empty();
}

dijkstra qtop()
{
    if(q[0].empty() || q[1].empty())
    {
        int i = q[0].empty();
        return q[i].front();
    }

    int i = (q[0].front().cena > q[1].front().cena);
    return q[i].front();
}

void qpop()
{
    if(q[0].empty() || q[1].empty())
    {
        int i = q[0].empty();
        q[i].pop();
        return;
    }

    int i = (q[0].front().cena > q[1].front().cena);
    q[i].pop();
}

void qpush(dijkstra d,int i)
{
    q[i].push(d);
}

int main()
{
        ios::sync_with_stdio(false);
        cin.tie(0);
        cout.tie(0);

    int n,coef,m;
    cin >> n >> coef >> m;

    graf.resize(n);

    while(m--)
    {
        int x,y;
        cin >> x >> y;
        --x,--y;
        graf[x].push_back(y);
        graf[y].push_back(x);
    }

    bicykel.resize(n,false);
    int b;
    cin >> b;
    while(b--)
    {
        int c;
        cin >> c;
        bicykel[c-1] = true;
    }

    for(int i=0;i<2;++i)
        dist[i].resize(n,(long long)n*coef);

    dist[0][0] = 0;

    qpush({0,0,0},0);

    while(!qempty())
    {
        dijkstra d = qtop();
        qpop();
        if(dist[d.stav][d.vrchol] < d.cena) continue;

        for(int i=0;i<graf[d.vrchol].size();++i)
        {
            int kam = graf[d.vrchol][i];
            long long nova_cena = d.cena + 1 + (coef-1)*(d.stav==false);

            if(dist[d.stav][kam] > nova_cena)
            {
                dist[d.stav][kam] = nova_cena;
                qpush({d.stav,kam,nova_cena},d.stav);
            }
        }

        if(bicykel[d.vrchol])
        {
            bool novy_stav = !d.stav;
            if(dist[novy_stav][d.vrchol] > d.cena)
            {
```

```
                dist[novy_stav][d.vrchol] = d.cena;
                qpush({novy_stav,d.vrchol,d.cena},novy_stav);
            }
        }
    }

    cout << dist[0][n-1] << "\n";
}
```

MišoF (misof@ksp.sk)

8. Antény

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

Vzorové riešenie tejto úlohy bude mať vtipnú časovú zložitosť $O((n+q)\sqrt{n}\log f)$, kde n je počet antén, q je počet otázok a f je rozsah hodnôt pre frekvencie antén. Ako takáto veselá časová zložitosť vznikne? Uvidíme časom.

Jednoduchšia úloha

Zabudnime na to, že máme nejaké pole a nejaké otázky, a zamyslime sa nad jednoduchšou úlohou, ktorá vyzerá nasledovne: Dané je číslo δ. Začíname s prázdnou (multi)množinou. Následne dostaneme sadu príkazov, pričom každý je buď tvaru "pridaj toto číslo" alebo "odstráň toto skôr pridané číslo". Všetky čísla sú celé z rozsahu od 0 po f. Po každej operácii musíme odpovedať, koľko dvojíc čísel v našej množine má vzdialenosť aspoň δ.

Ako túto úlohu efektívne riešiť? Existujú mnohé dátové štruktúry, pomocou ktorých vieme každú otázku zodpovedať v logaritmickom čase. Napr. pomocou vhodného vyvažovaného binárneho stromu by sme pre danú postupnosť n príkazov vedeli každý príkaz spracovať v čase $O(\log n)$, a to dokonca bez ohľadu na veľkosť f.

Existujú však aj riešenia omnoho jednoduchšie na implementáciu. Asi najľahšie je použiť Fenwickov ("fínsky") strom, resp. súčtový intervalový strom[11]. Pre každé číslo od 0 po f si budeme v strome pamätať, *koľkokrát* sa už v našej množine nachádza. Strom nám následne umožní v čase $O(\log f)$ o ľubovoľnom intervale zistiť, koľko prvkov našej množiny v ňom leží.

No a toto využijeme nasledovne: Ak napr. dostaneme príkaz pridať nový prvok x, najskôr si zistíme, koľko prvkov leží v $(x-\delta, x+\delta)$ a z toho vieme vypočítať, koľko nových dvojíc dostatočne vzdialených prvkov nám práve pribudlo.

Toto riešenie teda každý príkaz (pridanie alebo odobratie čísla) spracuje v čase $O(\log f)$.

Pôvodná úloha

Vráťme sa teraz k našej pôvodnej úlohe.

Technika, ktorú si ukážeme, sa dá použiť vždy, keď sú splnené isté predpoklady. V prvom rade potrebujeme, aby išlo (rovnako ako v tejto úlohe) o *offline* problém – teda všetky otázky dostaneme naraz a môžeme si vybrať, v akom poradí ich zodpovieme. To ale samo o sebe nestačí.

Druhá vlastnosť, ktorú potrebujeme, súvisí práve s vyššie uvedenou jednoduchšou úlohou. Naša technika bude použiteľná pre všetky problémy, kde takýmto spôsobom zjednodušenú úlohu vieme efektívne riešiť.

Odmocninové vedierka

Na vstupe sme dostali q rôznych otázok, ktoré máme všetky zodpovedať. Náš algoritmus bude efektívny vďaka tomu, že si otázky šikovne prerozdelíme do skupín a potom vyriešime vždy celú skupinu naraz.

Predstavme si, že sme si celý rad antén (pole dĺžky n) rozdelili na $\sqrt{n}$ úsekov. Následne môžeme zobrať všetky otázky a roztriediť ich do $\sqrt{n}$ vedierok podľa toho, *v ktorom úseku začínajú.*

Formálne, zoberme si $s = \lceil\sqrt{n}\,\rceil$ a potom každú otázku $[l_i, h_i]$ umiestnime do vedierka $\lfloor l_i/s \rfloor$.

Čo sme takto dostali? V každom vedierku máme otázky, ktoré *začínajú zhruba na tom istom mieste.* (Končiť však môžu veľmi rôzne. Vedierko číslo 0 môže obsahovať ako otázku na jediné políčko, tak aj otázku na úplne celé pole.)

Spracovanie jedného vedierka

Každé vedierko s otázkami teraz spracujeme nasledujúcim spôsobom:

1. Všetky otázky vo vedierku usporiadame vzostupne podľa ich *konca.*
2. Prvú otázku zodpovieme hrubou silou: začneme s prázdnou množinou a postupne po jednom do nej pridáme prvky úseku, na ktorý sa otázka pýta.

[11] https://www.ksp.sk/kucharka/intervalovy_strom

3. Každú ďalšiu otázku zodpovieme tak, že postupne prerobíme predchádzajúci interval na nový: najskôr pridávame prvky na konci, potom pridávame alebo uberáme prvky na začiatku.

Teda ak sme napr. práve zodpovedali otázku $[7, 47]$ a čaká nás otázka $[3, 52]$, tak postupne do našej množiny pridáme prvky na indexoch 48, 49, 50, 51, 52, 6, 5, 4, a 3. Ak by potom nasledovala otázka $[4, 53]$, tak by sme do našej množiny následne pridali prvok na indexe 53 a potom opäť odstránili prvok na indexe 3.

A to je už úplne všetko. Jednoduché, nie?

Odhad časovej zložitosti

Implemenácia je skutočne veľmi jednoduchá, odhad časovej zložitosti nás však trochu potrápi.

V každom vedierku, v ktorom máme aspoň jednu otázku, budeme jednu otázku spracúvať hrubou silou. Keďže táto otázka má dĺžku $O(n)$, jej spracovanie si vyžiada $O(n)$ množinových operácií. Ak by sme teda z každého z $\sqrt{n}$ vedierok spracovali len prvú otázku, trvalo by to $O(\sqrt{n} \cdot n \log f)$.

Koľko práce dokopy pridá spracovanie všetkých ostatných otázok?

V každom vedierku potrebujeme otázky usporiadať podľa konca. Ak každé vedierko samostatne usporiadame napríklad MergeSortom[12], dokopy nám to bude trvať $O(q \log q)$ času (keďže dokopy je vo vedierkach q otázok). Iná možnosť s rovnakou časovou zložitosťou je usporiadať si na začiatku všetky otázky podľa konca a až potom ich prerozdeliť do vedierok – čím automaticky budú usporiadané v každom z vedierok. No keďže koniec je číslo z rozsahu 0 až $n - 1$, namiesto MergeSortu môžeme použiť CountSort, čím dostaneme zložitosť $O(q + n)$. Ak je počet otázok q zhruba rovný počtu antén n, časová zložitosť ľubovoľnej z týchto možností bude zanedbateľná oproti zvyšku riešenia.

Keď prerábame nejakú otázku na nasledujúcu, robíme dva typy zmien:

- pridávame prvky na pravom okraji otázky
- pridávame a uberáme prvky na ľavom okraji otázky

Pridávanie prvkov na pravom okraji je efektívne. Pozrime sa na ľubovoľné vedierko. Keďže sú otázky v ňom usporiadané podľa konca, vždy, keď ideme na nasledujúcu, budeme na pravom kraji len pridávať prvky, nikdy nie odoberať. A teda každý z n prvkov poľa takto pridáme do našej množiny nanajvýš raz. Dokopy teda tieto kroky pre jedno vedierko budú trvať $O(n \log f)$. No a keďže vedierok je $\sqrt{n}$, celkový čas strávený posúvaním pravého konca je $O(n\sqrt{n} \log f)$.

No a aj pridávanie a uberanie prvkov na ľavom okraji je efektívne. Toto je to miesto, kde sa ukáže, prečo sme vlastne delili otázky do vedierok. Totiž keď sú dve otázky v tom istom vedierku, sú ich začiatky vzdialené nanajvýš o $\sqrt{n}$. No a teda keď prerábame jednu z nich na druhú, na ľavom okraji stačí postupne spraviť $O(\sqrt{n})$ zmien. Dokopy pre všetkých q otázok teda dostávame $O(q\sqrt{n})$ zmien, a keďže každú vieme spraviť v čase $O(\log f)$, celkový čas strávený týmito krokmi je $O(q\sqrt{n} \log f)$.

Celkovú časovú zložitosť teraz dostaneme sčítaním vyššie uvedených odhadov.

Historická poznámka

Táto technika je pomerne známa v svete efektívnych algoritmov, ale do súťažného programovania dorazila až v roku 2009, kedy Mo Tao touto technikou vyriešil úlohu počas čínskeho prípravného sústredenia pred IOI. V súťažnom programovaní sa následne zaužívalo volať túto techniku *Mo's algorithm*.

Listing programu (C++)

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

struct Fenwick1D {
    int size;
    vector<int> T;
    Fenwick1D(int maxval) { size = 1; while (size < maxval) size <<= 1; T.clear(); T.resize(size+1,0); }
    void update(int x, int delta) { while (x <= size) { T[x] += delta; x += x & -x; } }
    int sum(int x1, int x2) { // sum in the closed interval [x1,x2]
        int res=0;
        --x1;
        while (x2) { res += T[x2]; x2 -= x2 & -x2; }
        while (x1) { res -= T[x1]; x1 -= x1 & -x1; }
        return res;
    }
};

const int SQRT = 317; // must be >=sqrt(maxN)

struct query { int lo, hi, id; };
```

[12] https://www.ksp.sk/kucharka/mergesort

```
void change(Fenwick1D &FT, int &curr_elements, long long &curr_pairs, int newval, int K, int delta) {
    int minv = max(1,newval-K+1), maxv = min(FT.size,newval+K-1);
    if (delta == -1) { FT.update(newval,delta); --curr_elements; }
    curr_pairs += delta * (curr_elements - FT.sum(minv,maxv));
    if (delta == +1) { FT.update(newval,delta); ++curr_elements; }
}

int main() {
    ios_base::sync_with_stdio(false); cin.tie(NULL); cout.tie(NULL);

    int N, delta;
    cin >> N >> delta;

    vector<int> F(N);
    for (int &f : F) cin >> f;

    int Q;
    vector< vector<query> > queries(SQRT);
    cin >> Q;
    for (int q=0; q<Q; ++q) {
        int l, h;
        cin >> l >> h;
        queries[l/SQRT].push_back( {l,h+1,q} );
    }

    vector<long long> answers(Q);

    for (int a=0; a<SQRT; ++a) {
        if (queries[a].empty()) continue;

        sort( queries[a].begin(), queries[a].end(), [](const query &A, const query &B) -> bool { return A.hi < B.hi; } );

        Fenwick1D FT(1000047);
        int curr_elements = 0;
        long long curr_pairs = 0;
        int curr_lo = queries[a][0].lo, curr_hi = queries[a][0].hi;
        for (int x=curr_lo; x<curr_hi; ++x) change( FT, curr_elements, curr_pairs, F[x], delta, +1 );
        answers[ queries[a][0].id ] = curr_pairs;
        for (unsigned b=1; b<queries[a].size(); ++b) {
            while (curr_hi < queries[a][b].hi) change( FT, curr_elements, curr_pairs, F[curr_hi++], delta, +1 );
            while (curr_lo > queries[a][b].lo) change( FT, curr_elements, curr_pairs, F[--curr_lo], delta, +1 );
            while (curr_lo < queries[a][b].lo) change( FT, curr_elements, curr_pairs, F[curr_lo++], delta, -1 );
            answers[ queries[a][b].id ] = curr_pairs;
        }
    }
    for (auto a : answers) cout << a << endl;
}
```