A-II-1 Bezpečné medziplanetárne cestovanie

Stručná myšlienka vzorového riešenia

Zo zadaného grafu odstránime nepoužiteľné vrcholy a hrany. V tom, čo nám ostalo, nájdeme mosty. Navyše si vhodne predpočítame, ako sú rozložené vrcholy typu 2, aby sme pre každý most vedeli povedať, či sú všetky na tej istej jeho strane (vtedy je kritický) alebo nie.

Spoločný úvod pre všetky riešenia

Keďže nás zaujímajú bezpečné cesty a žiadna bezpečná cesta cez planétu typu 0 nevedie, tak ich budeme ignorovať. A tak isto budeme ignorovať teleporty, ktoré majú niektorý koniec na takejto planéte. Najľahšie je zahodiť ich priamo pri načítaní vstupu.

Navyše ľahko spravíme jedno ďalšie predspracovanie vstupu: rozdelíme ho na komponenty súvislosti. Zjavne stačí každý komponent riešiť samostatne. V ďalšom texte riešenia budeme preto predpokladať, že spracúvaná sieť teleportov je súvislá.

Ešte si môžeme všimnúť, že ak niektorý komponent súvislosti obsahuje samé planéty typu 1, tak žiaden teleport v ňom nie je kritický – totiž ani teraz sa zo žiadnej z týchto planét nedá zachrániť. Taktiež žiadne kritické teleporty nemôže obsahovať komponent obsahujúci samé planéty typu 2 – tam sa nie je odkiaľ zachraňovať. Takéto komponenty teda môžeme rovno ignorovať, zjednoduší to neskôr rozbor prípadov.

Riešenie takmer hrubou silou

Skúsme teraz nájsť algoritmus, ktorý pre daný teleport t povie, či je kritický. (Pritom predpokladáme, že komponent súvislosti, obsahujúci tento teleport, má v sebe aspoň jednu planétu typu 2.)

Ľahko môžeme vidieť, že pokiaľ obidva konce teleportu sú na planéte typu 2, tak teleport určite kritický nie je. Totiž keby nejaká zachraňujúca cesta viedla cez neho, tak určite existuje aj taká cesta, ktoré skončí už pred týmto teleportom.

Ak je na niektorom konci teleportu planéta p typu 1, tak potrebujeme overiť, či sa z nej aj po odstránení teleportu t dá zachrániť. To môžeme spraviť tak, že z planéty p spustíme prehľadávanie do hĺbky a zakážeme mu prejsť cez teleport t. Ak sa toto prehľadávanie dostane na planétu typu 2, tak sa vieme zachrániť. (Ak teleport t viedol medzi dvomi planétami typu 1, postupne spustíme dve nezávislé prehľadávania. Ak sa pri čo len jednom z nich nevieme zachrániť, t je nutne kritický.)

Teraz si už len stačí uvedomiť, že pre žiadne iné planéty už nepotrebujeme overovať možnosť záchrany. Totiž keby odstránenie teleportu t pokazilo možnosť záchrany pre nejakú inú planétu q typu 1, tak určite pokazí aj možnosť záchrany pre planétu p typu 1, ktorá je „na tom istom konci“ teleportu t ako q. (Detaily tohto ľahkého dôkazu sporom prenechávame na čitateľa.)

Overenie kritickosti teleportu nás teda prinajhoršom stojí dve prehľadávania do hĺbky, čiže jeho časová zložitosť v sieti s n planétami a m teleportami bude $O(m + n)$. Keby sme takto overovali každý teleport, tak máme výsledný algoritmus s časovou zložitosťou $O((m + n)^2)$.

Rýchlejšie riešenie

Majme ľubovoľnú súvislú sieť teleportov. Pozrime sa na konkrétny teleport. Ak je aj po jeho odstránení sieť teleportov súvislá (teda vieme prejsť z ľubovoľnej planéty na ľubovoľnú inú), tak dotyčný teleport evidentne nemôže byť kritický. Kritické teleporty stačí teda hľadať medzi tými, ktorých odstránenie rozpojí sieť teleportov na dve časti. V grafovej terminológii sa takéto hrany grafu volajú *mosty*.

Mostov v grafe nemôže nikdy byť príliš veľa: ak má graf n vrcholov, môže mať nanajvýš $n - 1$ mostov. Súčasťou vzorového riešenia bude nájdenie všetkých mostov v zadanom grafe. Aj bez toho však vieme zlepšiť predchádzajúce riešenie. Na to nám bude stačiť, ak o väčšine hrán budeme vedieť povedať, že mostom *nie sú*.

Ako na to? Na našom súvislom grafe spustíme ľubovoľné prehľadávanie. Pre každý vrchol si zapamätáme hranu, ktorou sme ho počas prehľadávania prvýkrát objavili. Takto dostaneme jednu možnú *kostru* nášho grafu. (Kostra n-vrcholového grafu je strom tvorený $n - 1$ jeho hranami.)

No a teraz si už len stačí uvedomiť, že každý most musí byť súčasťou práve nájdenej kostry. Totiž keď z nášho grafu vyhádžeme trebárs aj všetky ostatné hrany, stále bude držať pokope. Lenže kostrových hrán je len $n-1$. A tu sme práve ušetrili! Namiesto toho, aby sme ako kritický teleport testovali každú z m hrán, stačí nám ich otestovať $n-1$. Tento vylepšený algoritmus má teda časovú zložitosť $O(n(m+n))$.

Vzorové riešenie

Ako sme si už ukázali, nutnou (ale nie postačujúcou!) podmienkou na to, aby teleport bol kritický, je, že musí byť mostom. No a most je kritickým teleportom vtedy a len vtedy, ak po jeho odobraní vzniknú dva komponenty: jeden ktorý obsahuje iba planéty typu 1, a druhý ktorý obsahuje aspoň jednu planétu typu 2.

Náš algoritmus najprv nájde všetky mosty a potom zistí, ktoré z nich sú kritické.

Na hľadanie mostov použijeme štandardný algoritmus, ktorý sa dá popísať nasledovne:

Z nejakého vrcholu pustíme prehľadávanie do hĺbky. Toto prehľadávanie nám hrany rozdelí na dva druhy: *stromové a spätné*. (Stromové hrany sú tie, ktorými sme objavili nejaký nový vrchol, spätné sú tie ostatné.)

Stromové hrany nám, ako u každého prehľadávania, vytvoria kostru nášho grafu. Pri prehľadávaní do hĺbky budeme tejto kostre hovoriť *DFS strom*. Rozmyslite si, že v DFS strome každá spätná hrana vedie medzi nejakým vrcholom a jeho predkom v DFS strome. (Túto vlastnosť napr. kostra určená prehľadávaním do šírky nemá!)

Už vieme, že mosty sú niektoré zo stromových hrán. Potrebujeme vedieť nejako identifikovať, ktoré z nich to sú a ktoré nie. Uvažujme teda nejakú stromovú hranu uv, ktorou sme objavili vrchol v. Kedy je táto hrana mostom? Vtedy a len vtedy, keď z podstromu s koreňom vo vrchole v žiadna spätná hrana nevedie von (teda do vrcholu u alebo jeho niektorého predka). Na základe tohto pozorovania teraz sformulujeme náš algoritmus.

Každému vrcholu x vieme priradiť hĺbku v DFS strome, označíme ju $h(x)$. Navyše pre každý vrchol spočítame nasledovnú veličinu $d(x)$: najmenšiu hĺbku, kam sa vieme dostať pomocou niekoľkých stromových hrán smerujúcich nadol a následne najviac jednej spätnej hrany.

Ak máme spočítané toto, tak mosty vieme nájsť nasledovne: Spätná hrana určite most nebude (keď ju odstránime, tak stále existuje cesta pomocou stromových hrán). Ak pre vrchol x, ktorý objavila stromová hrana, platí $h(x) > d(x)$, tak táto stromová hrana tiež nie je most (vieme ju nahradiť niekoľkými stromovými a spätnou). Ináč daná stromová hrana most je.

Zostáva popísať ako spočítame hodnoty $h(x)$ a $d(x)$. Hodnoty $h(x)$ vieme triviálne počítať pri objavovaní vrcholov. Hodnotu $d(x)$ spočítame ako minimum z hodnôt $d(x)$ synov vrcholu x v DFS strome a hĺbok vrcholov, kam vedú spätné hrany z vrcholu x.

Tento postup sa dá použiť priamo po načítaní grafu zo vstupu: postupne spúšťame prehľadávania do hĺbky, čím rozdelíme zadaný graf na komponenty a zároveň v každom komponente nájdeme DFS strom a na ňom všetky mosty. Keďže každé prehľadávanie má časovú zložitosť priamo úmernú veľkosti príslušného komponentu, má táto časť riešenia celkovú časovú zložitosť $O(m+n)$.

Teraz ešte potrebujeme spočítať podmienky pre kritickosť teleportu. Počas prehľadávania do hĺbky pre každý vrchol x spočítame dve hodnoty $c_1(x)$ a $c_2(x)$: počty vrcholov typu 1 a typu 2 v podstrome, ktorý visí pod ním v DFS strome (vrátane samotného vrcholu x). Keď takto spracujeme celý komponent, v koreni jeho DFS stromu dostaneme celkové počty C_1 a C_2 vrcholov typu 1 a 2 v ňom. Ak $C_1 = 0$ alebo $C_2 = 0$, komponent môžeme odignorovať. Inak platí, že most je kritický, ak sú všetky vrcholy typu 2 na tej istej jeho strane – teda ak platí ($c_2(x) = 0$ alebo $c_2(x) = C_2$).

Celé riešenie má teda zjavne optimálnu časovú aj pamäťovú zložitosť $O(m+n)$.

Listing programu:

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;
#define INF 1234567890

struct Vrchol {
  Vrchol() : depth(-1), best(INF), c1(0), c2(0) {}
  vector<int> next;
  int type, depth, best, c1, c2;
};
```

```
vector<Vrchol> v; // Vrcholy grafu
vector<pair<int, int> > mosty; // Kandidati na mosty: (index hrany, index vrcholu kam vedie)

// Prehlada vrchol x, vrati trojicu best, c1, c2 daneho vrchola,
// pripadne depth, 0, 0 ak sme tam uz boli
pair<int, pair<int, int> > dfs(int x, int depth, int odkial) {
  if (v[x].depth != -1) return make_pair(v[x].depth, make_pair(0, 0));
  v[x].depth = depth;
  for (int i = 0; i < v[x].next.size(); i++) {
    if (v[x].next[i] == odkial) continue;
    pair<int, pair<int, int> > ret = dfs(v[x].next[i], depth+1, x);
    v[x].best = min(v[x].best, ret.first);
    v[x].c1 += ret.second.first;
    v[x].c2 += ret.second.second;
  }
  if (v[x].type == 1) v[x].c1++;
  if (v[x].type == 2) v[x].c2++;
  if (v[x].best >= v[x].depth && depth != 0) mosty.push_back(make_pair(odkial, x));
  return make_pair(v[x].best, make_pair(v[x].c1, v[x].c2));
}

int main() {
  int n, m;
  scanf("%d %d", &n, &m);
  v.resize(n);
  for (int i = 0; i < n; i++) scanf("%d", &v[i].type);
  for (int i = 0; i < m; i++) {
    int a, b; scanf("%d %d", &a, &b); a--; b--;
    if (v[a].type == 0) continue;
    if (v[b].type == 0) continue;
    v[a].next.push_back(b);
    v[b].next.push_back(a);
  }
  for (int i = 0; i < n; i++) {
    mosty.clear();
    dfs(i, 0, -1);
    int C1 = v[i].c1,  C2 = v[i].c2;
    for (int j = 0; j < mosty.size(); j++) {
      if (v[mosty[j].second].c1 > 0 && v[mosty[j].second].c2 == 0 && C2 > 0) {
        printf("%d %d\n", mosty[j].first+1, mosty[j].second+1);
      }
      if (C1 - v[mosty[j].second].c1 > 0 && C2 - v[mosty[j].second].c2 == 0
          && v[mosty[j].second].c2 > 0) {
        printf("%d %d\n", mosty[j].first+1, mosty[j].second+1);
      }
    }
  }
}
```

A-II-2 Vlámačka à la Banach-Tarski

Na začiatok na chvíľu zabudnime na možnosť používať duplikátor. Dostaneme tak úlohu, ktorá je vo svete známa pod názvom *0-1 knapsack problem* – chceme vybrať niektoré z modelov tak, aby súčet ich hmotností nepresiahol m a zároveň sa snažíme maximalizovať súčet ich cien.

Táto úloha je NP-úplná, teda nepoznáme algoritmus, ktorý by ju riešil v čase polynomiálnom od počtu modelov na vstupe. V našom prípade však máme v zadaní zaručené, že hmotnosti jednotlivých modelov sú celé kladné čísla a že celková Robertova nosnosť je rozumne malá. To nám umožní použiť pseudopolynomiálny algoritmus založený na myšlienke dynamického programovania so zložitosťou $O(mn)$.

Definujme si $P[i, j]$ ako najvyššiu cenu, ktorú vieme získať ukradnutím niektorých modelov, ak máme na výber len z prvých i modelov na vstupe a súčet hmotností tých, ktoré vyberieme, môže byť najviac j. Riešením, ktoré chceme nájsť, je potom hodnota $P[n, m]$.

Hodnoty $P[0, j]$ sú pre všetky j rovné 0, keďže nemáme k dispozícii nijaký model.

Pre $i \geq 1$ vypočítame $P[i, j]$ takto: Označme si hmotnosť i-teho modelu w_i a jeho cenu c_i. Ak tento model nevezmeme, najlepšia cena, ktorú vieme dosiahnuť, je $P[i-1, j]$. A najlepšia cena, ktorú vieme dosiahnuť, ak ho vezmeme, je $c_i + P[i-1, j-w_i]$. To preto, lebo dostaneme cenu c_i za práve vybranú vec, a následne môžeme

z prvých $i-1$ modelov vyberať len tak, aby sme nepresiahli hmotnosť $j-w_i$). Spomedzi týchto dvoch možností si samozrejme vyberieme tú lepšiu.[1] Hodnota $P[i,j]$ sa teda rovná $\max(\ c_i + P[i-1, j-w_i],\ P[i-1,j]\ )$.

Všimnite si, že na výpočet $P[i,j]$ nám stačí poznať len $P[i-1,j]$ a $P[i-1,j-w_i]$. Preto ak budeme počítať hodnoty P postupne od menších i k väčším, budeme už mať v čase výpočtu $P[i,j]$ potrebné hodnoty pripravené. Takto dostávame riešenie s časovou zložitosťou $O(mn)$ – každú z $O(mn)$ hodnôt $P[i,j]$ vypočítame v konštantnom čase.

Ak by sme si pamätali všetky hodnoty P, potrebovali by sme $O(mn)$ pamäte. To však nie je nutné – počas výpočtu hodnôt P pre prvých i modelov si stačí pamätať len hodnoty P pre prvých $i-1$ modelov. Naša vzorová implementácia si to dokonca pamätá v jedinom poli a postupne od väčších j k menším nahrádza hodnoty $P[i-1,j]$ hodnotami $P[i,j]$. (Rozmyslite si, že si nikdy neprepíšeme informáciu, ktorú by sme ešte neskôr potrebovali.) Pamäťová zložitosť je teda $O(m)$.

Pomalšie riešenie pôvodnej úlohy

Teraz už priamo vieme navrhnúť funkčné a vcelku efektívne riešenie pôvodnej úlohy. Stačí vyskúšať všetkých n možností, na ktorú vec použiť duplikátor. Akonáhle duplikátor použijeme, stojíme pred obyčajnou úlohou 0-1 knapsack s $n+1$ modelmi. Stačí nám teda n-krát vyriešiť 0-1 knapsack a spomedzi všetkých nájdených riešení si vybrať to najlepšie. Takéto riešenie má časovú zložitosť $O(n^2m)$ a pamäťovú zložitosť $O(m+n)$.

Vzorové riešenie pôvodnej úlohy

Rýchlejšie riešenie úlohy s duplikátorom dostaneme vhodným rozšírením dynamického programovania, ktoré používame v zjednodušenej úlohe. Definujme si $Q[i,j]$ ako najvyššiu cenu, ktorú vieme získať ukradnutím niektorých modelov, ak máme na výber len z prvých i modelov na vstupe, súčet ich hmotností môže byť najviac j a navyše môžeme raz použiť duplikátor.

Hodnoty $Q[0,j]$ sú, rovnako ako $P[0,j]$, rovné 0.

Pozrime sa teraz, ako vyjadriť $Q[i,j]$ pre $i \geq 1$. Znova si označme hmotnosť i-teho modelu w_i a jeho cenu c_i. Oproti výpočtu $P[i,j]$ nám pribudla tretia možnosť: tento model môžeme zduplikovať a zobrať obe kópie. Takto získame riešenie, ktorého optimálna cena bude $2c_i + P[i-1, j-2w_i]$. (Spomedzi zvyšných $i-1$ modelov chceme vybrať najdrahšiu podmnožinu, ktorej hmotnosť je nanajvýš $j-2w_i$. Duplikátor sme už použili, preto pri výbere týchto $i-1$ modelov ho už použiť nesmieme.) Hodnotu $Q[i,j]$ teda vypočítame nasledovne:

$$Q[i,j] = \max\Big(\quad Q[i-1,j],\quad c_i + Q[i-1, j-w_i],\quad 2c_i + P[i-1, j-2w_i]\quad\Big)$$

Súčasťou úlohy bolo aj vypísať číslo modelu, ktorý sme v optimálnom riešení zduplikovali. Preto si pri výpočte $Q[i,j]$ pre každé i a j zapamätáme aj toto číslo.

Toto riešenie má časovú zložitosť $O(mn)$, teda rovnakú ako riešenie úlohy bez duplikátora. A opäť vieme dosiahnuť pamäťovú zložitosť $O(m)$: stačí postupne pre všetky i vhodne striedavo počítať hodnoty P a Q.

Listing programu:

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

int main() {
    int n, m; cin >> m >> n;
    vector<int> P(m + 1, 0);
    // v Q mame dvojice (najvyssia cena, cislo zduplikovaneho modelu)
    vector<pair<int, int> > Q(m + 1, make_pair(0, 0));
    for (int i = 0; i < n; ++i) {
        int w, c;
        cin >> w >> c;
        for (int j = m; j >= w; --j) {
            if (P[j - w] + c > P[j])
```

[1] Drobný technický detail: Na výber máme len vtedy, ak daný model ešte unesieme, teda ak $w_i \leq j$. V opačnom prípade nám ostáva jediná možnosť – i-ty model nevziať. V programe toto ošetríme vhodnou podmienkou.

```
                P[j] = P[j - w] + c;
            if (Q[j - w].first + c > Q[j].first)
                Q[j] = make_pair(Q[j - w].first + c, Q[j - w].second);
            if (j >= 2 * w && P[j - 2 * w] + 2 * c > Q[j].first)
                Q[j] = make_pair(P[j - 2 * w] + 2 * c, i + 1);
        }
    }
    if (Q[m].first > P[m]) cout << "zduplikuj model " << Q[m].second << endl
                               << "najlepsia cena " << Q[m].first << endl;
    else                   cout << "nezduplikuj nic" << endl
                               << "najlepsia cena " << P[m] << endl;
}
```

Iné vzorové riešenie

Predchádzajúce riešenie by sa dalo efektívne rozšíriť, ak by sme mali povolené použiť duplikátor nie raz, ale s-krát. Namiesto polí P a Q by sme mali polia P_0, P_1, ..., P_s, pričom $P_k[i, j]$ by sme si definovali ako najvyššiu cenu, ktorú vieme získať ukradnutím niektorých modelov, ak máme na výber len z prvých i modelov na vstupe, súčet ich hmotností môže byť najviac j a duplikátor môžeme použiť najviac k-krát.

V našom prípade však existuje aj jednoduchší prístup. Vypočítame len hodnoty P z predchádzajúceho riešenia. Bez duplikovania by najlepším riešením bola hodnota $P[n, m]$. Ak by sme zduplikovali i-ty model a duplikát ukradli, budeme mať pred sebou pôvodných n vecí, pričom si z nich môžeme nabrať do hmotnosti $m-w_i$. Najvyššia dosiahnuteľná cena by teda bola $c_i + P[n, m - w_i]$. Celkovo najlepšie riešenie nájdeme vyskúšaním všetkých možností čo zduplikovať. Časová zložitosť bude znova $O(mn)$, pamäťová zložitosť $O(m + n)$.

A-II-3 Parazity

V celom riešení predpokladáme, že platí $d \geq n$. V opačnom prípade (keď je typov báz viac ako dĺžka DNA, ktorú spracúvame), je totiž zjavne odpoveď 0.

Prvé riešenie, ktoré nám napadne skoro hneď, je použiť hrubú silu: Pre každý možný súvislý úsek si zistíme, či sa parazitom páči alebo nie. Toto vieme ľahko urobiť s časovou zložitosťou $O(d^3)$ tak, že si zvolíme začiatok a koniec (ktorých je rádovo d^2) a zakaždým pomocou jedného prechodu práve skúšaného úseku zistíme, či je tam každá báza aspoň raz.

Toto riešenie sa však dá veľmi jednoducho vylepšiť. Označme si $p(i)$ prvú takú pozíciu, že úsek i až $p(i)$ je parazitmi obľúbený. Potom pre každé $j > p(i)$ platí, že úsek i až j je tiež parazitmi obľúbený. Toto je celkom ľahké ukázať. Keďže na pozíciách i až $p(i)$ sa objavili všetky možné typy báz, tak sa tým skôr všetky typy báz vyskytujú v úseku od i po j. Stačí teda, keď pre každý možný začiatok i nájdeme jemu zodpovedajúci najľavejší možný koniec $p(i)$.

Ako to spraviť efektívne? Zoberieme si pole veľkosti n, do ktorého si budeme pre každú bázu značiť, či sme ju už videli. A navyše budeme mať premennú `pocet`, v ktorej si budeme pamätať, koľko *typov* báz sme už videli. Pridanie novej bázy x do práve spracúvaného úseku teraz vyzerá nasledovne: Pozrieme do poľa, či sme už bázu x videli. Ak áno, tak sa nič nezmenilo oproti predchádzajúceme úseku. Ak sme ju ešte nevideli, tak si ju v poli zaznačíme ako už videnú a zvýšime premennú `pocet`. Akonáhle `pocet` dosiahne n, našli sme pre práve skúšaný začiatok najľavejší možný koniec. Práve popísané riešenie má časovú zložitosť $O(d^2)$.

Vzorové riešenie

Vyššie popísané riešenie však stále nie je optimálne. K optimálnemu riešeniu nám chýba ešte jedno pozorovanie: Ak $i < j$ tak $p(i) \leq p(j)$. Toto sa dá ukázať sporom. Ak by platilo, že $p(j) < p(i)$, tak to znamená, že na úseku j až $p(j)$ sa objavili všetky typy báz. Lenže toto je len nejaká časť úseku i až $p(i)$, a teda aj na úseku i až $p(j)$ sú všetky typy báz, a to je spor s tým, že $p(i)$ je najmenší taký index, pre ktorý platí, že i až $p(i)$ je parazitmi obľúbený.

Ako toto pozorovanie využiť? Hovorí nám vlastne, že keď už poznáme hodnotu $p(i)$, tak pri spracúvaní úseku začínajúceho na pozícii $i+1$ stačí jeho koniec hľadať od pozície $p(i)$ ďalej. Lenže na to by sme potrebovali vedieť, ktoré bázy sa nachádzajú v úseku od pozície $i + 1$ po pozíciu $p(i)$. A na to nám pole boolovských premenných, ktoré sme použili v predchádzajúcom riešení, nestačí – nevieme z neho povedať, či sa báza z pozície i nachádza ešte niekde medzi pozíciami $i + 1$ až $p(i)$.

Tento nedostatok ale ľahko odstránime: namiesto boolovských premenných použijeme celočíselné. V nich si budeme pre každú bázu pamätať, *koľkokrát* sa nachádza v práve spracúvanom úseku. V nižšie uvedenom programe ide o pole `vyskyty`. Naďalej budeme používať premennú `pocet`, v ktorej budeme mať uložený počet rôznych typov báz v práve spracúvanom úseku.

Začneme tým, že si vyrátame hodnotu $p(1)$, teda najľavejší možný koniec, ak je začiatkom úseku prvý prvok na vstupe. V okamihu, kedy nájdeme hodnotu $p(1)$, máme poli `vyskyty` pre každý typ bázy jeho počet výskytov na pozíciách 1 až $p(1)$. Teraz posunieme začiatok na pozíciu 2. Tým nám z aktuálneho úseku vypadla báza z pozície 1. Zmenšíme teda hodnotu na príslušnom políčku poľa `vyskyty`. A ak sme ju zmenšili na 0, tak o jedno zmenšíme aj hodnotu v premennej `pocet`. A pokračujeme rovnako ako predtým: kým je `pocet` menej ako n, posúvame koniec aktuálneho úseku. Takto pokračujeme postupne pre všetky možné začiatky, až do chvíle, kým pre niektorý začiatok nezistíme, že už žiaden možný koniec nevyhovuje.

Nakoľko sme vlastne zlepšili časovú zložitosť? Stačí si uvedomiť, že v priebehu riešenia len meníme dva indexy: jeden, čo ukazuje na aktuálne skúšaný začiatok úseku, a jeden ukazujúci na jeho koniec. Posunutie každého indexu vieme spraviť v konštantnom čase. No a v priebehu celého riešenia každý z našich indexov prejde (nanajvýš) raz celú postupnosť. Preto je časová zložitosť tohto riešenia $O(d)$.

Listing programu:

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

int main() {
    int n, d; cin >> n >> d;
    vector<int> vstup(d);
    for (int i=0; i<d; ++i) { cin >> vstup[i]; --vstup[i]; }

    vector<int> vyskyty(n,0); // pre kazdu bazu pocet vyskytov
    int pocet = 0;
    long long vysledok = 0;

    for (int zaciatok=0, p=0; zaciatok<d ; ++zaciatok) {
        while (p<d && pocet<n) {
            if (vyskyty[ vstup[p] ]==0) ++pocet;
            ++vyskyty[ vstup[p] ];
            ++p;
        }
        if (pocet==n) vysledok += d-p+1;
        --vyskyty[ vstup[zaciatok] ];
        if (vyskyty[ vstup[zaciatok] ]==0) --pocet;
    }
    cout << vysledok << endl;
}
```

Iné dobré riešenia

Namiesto poľa `vyskyty`, ktoré budeme priebežne udržiavať, si môžeme nejaké údaje predpočítať. Napríklad si môžeme pre každú pozíciu x (od 1 po d) a každý typ bázy y (od 1 po n) predpočítať, kedy najbližšie za pozíciou x sa vyskytne báza y. Tieto informácie vieme priamočiaro predpočítať v čase $O(nd)$ a pomocou nich už ľahko implementujeme riešenie podobné vzorovému – vždy, keď posunieme začiatok úseku, nájdeme si, kde sa najbližšie vyskytuje báza, ktorá z neho práve vypadla.

To isté vieme spraviť ešte trochu šikovnejšie v čase $O(d)$, čím dostaneme iné optimálne riešenie. Stačí si napríklad pre každý typ bázy predpočítať zoznam pozícií, na ktorých sa nachádza. (A sú dva rôzne spôsoby, ako toto implementovať: buď môžeme mať n disjunktných zoznamov – jeden pre každú bázu – alebo jednoducho pomocné pole dĺžky d, kde je pre každú pozíciu i zaznačené, na ktorej nasledujúcej pozícii leží báza rovnaká ako na pozícii i.)

Iná technika návrhu efektívneho riešenia je použitie binárneho vyhľadávania. Pri takýchto riešeniach si najskôr predpočítame nejaké informácie, z ktorých vieme rýchlo povedať, či je konkrétny úsek vyhovujúci. (Napr. vyššie uvedené predpočítanie v $O(nd)$ má takúto vlastnosť, sú však aj lepšie spôsoby.) Následne pre každý z d možných začiatkov binárnym vyhľadávaním nájdeme jemu zodpovedajúci koniec. Najbežnejšie časové zložitosti takýchto riešení sú $O(d \log d)$, $O(d \log^2 d)$, prípadne $O(dn + d \log d)$.

A-II-4 Zlomkové programy

V oboch podúlohách budú riešenia založené na podobnom princípe ako v domácom kole: Na každé prvočíslo v rozklade aktuálnej hodnoty sa môžeme dívať ako na premennú. Teda ak je napríklad aktuálna hodnota rovná $2^4 3^5 5^2$, môžeme to čítať nasledovne: „v premennej #2 je uložená hodnota 4, v premennej #3 je uložená hodnota 5 a v premennej #5 je uložená hodnota 2“.

A ako prebieha krok výpočtu? Hľadáme vhodný zlomok – teda pozeráme sa na menovatele a pre každý z nich *otestujeme*, či sú v nejakých premenných dostatočne vysoké hodnoty. Napríklad zlomok s menovateľom $2^3 7$ môžeme použiť len vtedy, keď aktuálne máme v premennej #2 hodnotu aspoň 3 a v premennej #7 hodnotu aspoň 1. No a keď nájdeme vhodný zlomok, najskôr *znížime* hodnoty premenných zodpovedajúcich jeho menovateľu a potom *zvýšime* hodnoty iných premenných, zodpovedajúcich jeho čitateľu.

Podúloha a)

Zadanie tejto podúlohy si teraz môžeme preformulovať nasledovne: na začiatku sú v premenných #2 a #3 uložené vstupné hodnoty x a y. Naším cieľom je v premennej #5 vyrobiť výstupnú hodnotu $\max(x, y)$.

Slovne si riešenie tejto úlohy môžeme popísať nasledovne:

1. kým sú premenné #2 a #3 obe kladné, obe zníž a zvýš premennú #5
2. kým je premenná #2 kladná, zníž ju a zvýš premennú #5
3. kým je premenná #3 kladná, zníž ju a zvýš premennú #5

(Spomedzi krokov 2 a 3 sa vždy najviac jeden naozaj vykoná, keďže po kroku 1 ostala v aspoň jednej z premenných #2 a #3 nula.)

A tomuto slovnému popisu zodpovedá nasledujúci jednoduchý program:

$$\left(\frac{5}{6} \;,\; \frac{5}{2} \;,\; \frac{5}{3} \right)$$

Príklad výpočtu pre $n = 144 = 2^4 3^2$:

$$2^4 3^2 \quad \xrightarrow{1} \quad 2^3 3^1 5 \quad \xrightarrow{1} \quad 2^2 5^2 \quad \xrightarrow{2} \quad 2^1 5^3 \quad \xrightarrow{2} \quad 5^4$$

Príklad výpočtu pre $n = 729 = 2^0 3^6$:

$$3^6 \quad \xrightarrow{3} \quad 3^5 5 \quad \xrightarrow{3} \quad 3^4 5^2 \quad \xrightarrow{3} \quad 3^3 5^3 \quad \xrightarrow{3} \quad 3^2 5^4 \quad \xrightarrow{3} \quad 3^1 5^5 \quad \xrightarrow{3} \quad 5^6$$

Podúloha b)

Túto úlohu si môžeme preformulovať nasledovne: treba sčítať obsah premenných #2 a #3, ale zároveň zachovať obsah týchto premenných.

Ak by nebolo treba zachovať vstupné premenné, bolo by riešenie tejto úlohy jednoduché:

1. kým je premenná #2 kladná, zníž ju a zvýš premennú #5
2. kým je premenná #3 kladná, zníž ju a zvýš premennú #5

Aby sme nestratili pôvodný obsah premenných, budeme vždy naraz zvyšovať dve premenné: aj premennú #5, aj novú pomocnú premennú. Začiatok nášho algoritmu bude teda vyzerať takto:

1. kým je premenná #2 kladná, zníž ju, zvýš premennú #5, zvýš premennú #43
2. kým je premenná #3 kladná, zníž ju, zvýš premennú #5, zvýš premennú #47

A následne už len „upraceme“ – obsah premenných #43 a #47 vrátime späť:

3. kým je premenná #43 kladná, zníž ju a zvýš premennú #2

4. kým je premenná #47 kladná, zníž ju a zvýš premennú #3

Ak by sme ale priamo podľa tohoto algoritmu napísali zlomkový program, nastali by podobné problémy ako v riešení druhej podúlohy domáceho kola: program by nikdy neskončil. Akonáhle by sa vykonal krok 3, bola by premenná #2 opäť kladná, opäť by sa vykonal krok 1, a tak dokola.

My potrebujeme zabezpečiť, aby sa pri vykonávaní nášho zlomkového programu kroky 1 až 4 vykonali len raz, a to presne v tomto poradí.

A na to využijeme hodnotu 7, ktorou je zaručene deliteľné vstupné číslo. Zlomkový program navrhneme tak, aby počas výpočtu bola v každom okamihu práve jedna z premenných #7, #11, #13 a #17 nenulová. A podľa toho, ktorá z nich to je, budeme vykonávať ďalší krok výpočtu.

V spresnenom algoritme už teda nemusíme číslovať jednotlivé kroky, ich poradie bude jednoznačne určené. Nový, presnejší popis nášho algoritmu vyzerá nasledovne:

- kým je premenná #7 kladná:
  - ak je premenná #2 kladná, zníž ju, zvýš premennú #5, zvýš premennú #43
  - ak je premenná #3 kladná, zníž ju, zvýš premennú #5, zvýš premennú #47
  - ak sú premenné #2 aj #3 nulové, vynuluj premennú #7, nastav premennú #13 na 1
- kým je premenná #13 kladná:
  - ak je premenná #43 kladná, zníž ju a zvýš premennú #2
  - ak je premenná #47 kladná, zníž ju a zvýš premennú #3
  - ak sú premenné #43 aj #47 nulové, vynuluj premennú #13 a tým výpočet končí

Premenné #11 a #17 budú nášmu programu slúžiť ako pomocné pri kontrole, či je premenná #7, resp. #13, kladná. Napr. vždy, keď aktuálnu hodnotu (v krokoch 1 a 2) vydelíme 7, zároveň ju vynásobíme 11. A následne použijeme zlomok 7/11, aby sme opäť dostali hodnotu deliteľnú 7.

Výsledný program:

$$\left(\frac{5\cdot 43\cdot 11}{2\cdot 7}\ ,\ \frac{5\cdot 47\cdot 11}{3\cdot 7}\ ,\ \frac{7}{11}\ ,\ \frac{13}{7}\ ,\ \frac{2\cdot 17}{43\cdot 13}\ ,\ \frac{3\cdot 17}{47\cdot 13}\ ,\ \frac{13}{17}\ ,\ \frac{1}{13} \right)$$

Príklad výpočtu pre $n = 2^1 \cdot 3^2 \cdot \mathbf{7}$:

$$\begin{aligned} 2^1\cdot 3^2\cdot \mathbf{7} &\xrightarrow{1} 3^2\cdot 5^1\cdot \mathbf{11}\cdot 43^1 \xrightarrow{3} 3^2\cdot 5^1\cdot \mathbf{7}\cdot 43^1 \xrightarrow{2} 3^1\cdot 5^2\cdot \mathbf{11}\cdot 43^1\cdot 47^1 \xrightarrow{3} 3^1\cdot 5^2\cdot \mathbf{7}\cdot 43^1\cdot 47^1 \\ &\xrightarrow{2} 5^3\cdot \mathbf{11}\cdot 43^1\cdot 47^2 \xrightarrow{3} 5^3\cdot \mathbf{7}\cdot 43^1\cdot 47^2 \xrightarrow{4} 5^3\cdot \mathbf{13}\cdot 43^1\cdot 47^2 \xrightarrow{5} 2^1\cdot 5^3\cdot \mathbf{17}\cdot 47^2 \\ &\xrightarrow{7} 2^1\cdot 5^3\cdot \mathbf{13}\cdot 47^2 \xrightarrow{6} 2^1\cdot 3^1\cdot 5^3\cdot \mathbf{17}\cdot 47^1 \xrightarrow{7} 2^1\cdot 3^1\cdot 5^3\cdot \mathbf{13}\cdot 47^1 \xrightarrow{6} 2^1\cdot 3^2\cdot 5^3\cdot \mathbf{17} \\ &\xrightarrow{7} 2^1\cdot 3^2\cdot 5^3\cdot \mathbf{13} \xrightarrow{8} 2^1\cdot 3^2\cdot 5^3 \end{aligned}$$

(Tučným písmom je všade vysádzané prvočíslo reprezentujúce aktuálny „stav výpočtu“. Ostatné prvočísla majú vždy uvedený exponent, aj ak je rovný jednej – tieto exponenty sú hodnoty dotyčných premenných.)

Trochu iné riešenie: V poslednej fáze programu už nepotrebujeme mať špeciálne prvočíslo, ktoré nám hovorí, aké zlomky môžeme používať – poslednú fázu totiž môžeme spoznať jednoducho tak, že sa nedá použiť žiadny zlomok z predchádzajúcich fáz. Toto pozorovanie vedie k o niečo jednoduchšiemu a stručnejšiemu programu:

$$\big(\ (5.43.11)/(2.7),\ (5.47.11)/(3.7),\ 7/11,\ 1/7,\ 2/43,\ 3/47\ \big)$$

DVADSIATY SIEDMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Autori úloh: Michal Anderle, Vladimír Boža, Michal Forišek, Peter Fulla
Recenzent: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2012