P-II-1 Bezpečné cestování

Společný úvod pro všechna řešení

Žádná bezpečná cesta nevede přes planetu typu 0. Tyto planety proto můžeme vynechat, stejně jako všechny teleporty, které mají některý svůj konec na planetě typu 0. Nejlepší je vypustit je hned při čtení vstupních dat. Poté můžeme snadno provést ještě jedno předzpracování vstupu: zadaný graf rozdělíme na komponenty souvislosti. Je zjevné, že stačí každou komponentu souvislosti řešit samostatně.

Všimněte si, že když některá komponenta souvislosti obsahuje pouze planety typu 1, pak žádný její teleport není kritický – z těchto planet se totiž člověk stejně nemůže zachránit. Podobně ani komponenta souvislosti tvořená pouze planetami typu 2 neobsahuje žádné kritické teleporty – tam zase není jediná planeta, odkud bychom mohli člověka zachraňovat. Takové komponenty tedy můžeme rovnou ignorovat, což nám později zjednoduší rozbor případů.

Řešení téměř hrubou silou

Popíšeme algoritmus, který pro zvolený teleport t určí, zda je kritický. Přitom předpokládáme, že komponenta souvislosti obsahující tento teleport má aspoň jednu planetu typu 2.

Pokud se oba konce teleportu t nacházejí na planetách typu 2, pak tento teleport určitě není kritický. Kdyby totiž přes něj vedla nějaká zachraňující cesta, jistě by existovala i zachraňující cesta, která skončí již před ním.

Je-li na některém konci teleportu planeta p typu 1, potřebujeme ověřit, zda se z ní i po odstranění teleportu t člověk zachrání. To můžeme zjistit tak, že z planety p spustíme prohledávání do hloubky a zakážeme mu projít teleportem t. Jestliže toto prohledávání dojde až na nějakou planetu typu 2, člověk se z planety p zachránit může. Pokud teleport t vede mezi dvěma planetami typu 1, postupně spustíme dvě nezávislá prohledávání. Když se aspoň při jednom z nich člověk nedokáže zachránit, pak je teleport t kritický.

Nyní už jen stačí uvědomit si, že pro žádné jiné planety nepotřebujeme ověřovat možnost záchrany. Kdyby totiž odstranění teleportu t pokazilo možnost záchrany pro nějakou jinou planetu q typu 1, tak jistě pokazí i možnost záchrany pro planetu p typu 1, která je „na stejném konci“ teleportu t jako q.

Ověření každého teleportu nás stojí nejvýše dvě prohledávání do hloubky, takže jeho časová složitost v síti s n planetami a m teleporty je $\mathcal{O}(m+n)$. Kdybychom takto ověřovali každý teleport, máme výsledný algoritmus s časovou složitostí $\mathcal{O}((m+n)^2)$.

Rychlejší řešení

Mějme libovolnou souvislou síť teleportů. Uvažujme v ní jeden konkrétní teleport. Pokud i po jeho odstranění zůstane síť teleportů souvislá (tzn. pomocí teleportů se dostaneme z libovolné planety na libovolnou jinou), pak dotyčný teleport

evidentně nemůže být kritický. Kritické teleporty stačí tedy hledat mezi těmi, jejichž odstranění rozpojí síť teleportů na dvě části. V grafové terminologii takové hrany grafu nazýváme *mosty.*

V grafu nikdy nemůže být příliš mnoho mostů: má-li graf n vrcholů, může obsahovat nejvýše $n - 1$ mostů. Součástí našeho vzorového řešení bude nalezení všech mostů v zadaném grafu. I bez toho ale dokážeme zlepšit předcházející řešení. K tomu nám bude stačit, když o většině hran dokážeme s jistotou říci, že mostem *nejsou.*

Spustíme libovolné prohledávání našeho souvislého grafu. Pro každý vrchol si zapamatujeme hranu, kudy jsme do něj během prohledávání poprvé přišli. Takto dostaneme jednu možnou *kostru* grafu. (Kostra n-vrcholového grafu je strom tvořený $n - 1$ jeho hranami.)

Nyní stačí uvědomit si, že každý most musí být součástí právě nalezené kostry. Když totiž z grafu vypustíme třeba i všechny ostatní hrany, stále zůstane souvislý. Kostra ale obsahuje jenom $n-1$ hran. Namísto toho, abychom jako kritický teleport testovali každou z m hran grafu, stačí nám jich otestovat pouze $n-1$. Takto vylepšený algoritmus má časovou složitost $\mathcal{O}(n(m + n))$.

Vzorové řešení

Jak jsme už ukázali, aby byl teleport kritický, musí být mostem – to je nutnou, ale ne postačující podmínkou. Most je kritickým teleportem právě tehdy, když jeho odebráním vzniknou dvě komponenty tak, že jedna z nich obsahuje pouze planety typu 1 a druhá obsahuje aspoň jednu planetu typu 2.

Náš algoritmus nejprve určí všechny mosty v grafu a potom zjistí, které z nich jsou kritické. Pro hledání mostů použijeme standardní algoritmus, který lze popsat následovně:

Z některého vrcholu spustíme prohledávání do hloubky. Toto prohledávání nám hrany rozdělí na dva druhy: *stromové* a *zpětné.* Stromové hrany jsou ty, jimiž jsme poprvé přišli do nějakého vrcholu, zpětné jsou ostatní hrany.

Jako u každého prohledávání, stromové hrany nám vytvoří kostru grafu. Při prohledávání do hloubky budeme této kostře říkat *DFS strom.** Rozmyslete si, že v DFS stromě každá zpětná hrana vede mezi nějakým vrcholem a jeho předkem v DFS stromě. (Například kostra určená prohledáváním do šířky tuto vlastnost nemá!)

Už víme, že mosty jsou některé ze stromových hran. Potřebujeme rozhodnout, které z nich to jsou a které ne. Uvažujme tedy nějakou stromovou hranu uv, po níž jsme přišli z vrcholu u do vrcholu v. Kdy je tato hrana mostem? Právě tehdy, když z podstromu s kořenem v nevede žádná zpětná hrana ven (tzn. do vrcholu u nebo do některého jeho předka). Na základě tohoto pozorování nyní zformulujeme náš algoritmus.

Každému vrcholu x přiřadíme hloubku v DFS stromě, kterou označíme $h(x)$. Navíc pro každý vrchol spočítáme následující veličinu $d(x)$: nejmenší hloubku, kam

* Z anglického názvu Depth-First Search.

se dokážeme dostat pomocí několika stromových hran směřujících dolů a následně nejvýše jedné zpětné hrany.

Máme-li spočítány uvedené hodnoty, můžeme mosty určit následovně: Zpětná hrana určitě mostem nebude – když ji odstraníme, tak stále existuje cesta pomocí stromových hran. Když pro vrchol x, do něhož jsme přišli stromovou hranou, platí $h(x) > d(x)$, pak tato stromová hrana také není mostem (dokážeme ji nahradit několika stromovými a zpětnou hranou). V opačném případě daná stromová hrana mostem je.

Zbývá ukázat, jak spočítáme hodnoty $h(x)$ a $d(x)$. Hodnoty $h(x)$ můžeme triviálně počítat v průběhu prohledávání grafu do hloubky. Hodnotu $d(x)$ spočítáme jako minimum z hodnot $d(x)$ synů vrcholu x v DFS stromě a hloubek vrcholů, kam vedou zpětné hrany z vrcholu x.

Popsaný výpočet provedeme hned po načtení grafu ze vstupu. Postupně budeme spouštět prohledávání do hloubky, čímž rozdělíme zadaný graf na komponenty souvislosti a zároveň v každé komponentě najdeme DFS strom a v něm všechny mosty. Jelikož každé prohledávání má časovou složitost přímo úměrnou velikosti příslušné komponenty, má tato část řešení celkovou časovou složitost $\mathcal{O}(m + n)$.

Nyní ještě potřebujeme ověřit, které mosty v grafu představují kritické teleporty. Při prohledávání do hloubky pro každý vrchol x spočítáme dvě hodnoty $c_1(x)$ a $c_2(x)$: počty vrcholů typu 1 a typu 2 v podstromě, který visí pod ním v DFS stromě (včetně samotného vrcholu x). Když takto zpracujeme celou komponentu, v kořeni DFS stromu dostaneme celkové počty C_1 a C_2 vrcholů typu 1 a 2. Je-li $C_1 = 0$ nebo $C_2 = 0$, komponentu můžeme ignorovat. Jinak platí, že most je kritický, pokud všechny vrcholy typu 2 leží na téže jeho straně – tedy když je $c_2(x) = 0$ nebo $c_2(x) = C_2$.

Celé řešení má optimální časovou a paměťovou složitost $\mathcal{O}(m + n)$.

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

struct Vrchol {
    Vrchol() : depth(-1), best(INF), c1(0), c2(0) {}
    int type;
    vector<int> next;
    int depth;
    int best;
    int c1, c2;
};
vector<Vrchol> v;              // Vrcholy grafu
vector<pair<int, int>> mosty; // Kandidáti na mosty: (index hrany, index jejího cíle)

// Prohledá vrchol x, vrátí trojici (best, c1, c2) daného vrcholu,
// případně (depth, 0, 0), pokud jsme tam už byli.
pair<int, pair<int, int>> dfs(int x, int depth, int odkud) {
    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] == odkud) 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(odkud, 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 || 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);
        }
    }

    return 0;
}
```

P-II-2 Krádež

Problém batohu

Nejprve na chvíli zapomeneme na možnost používat duplikátor. Dostaneme tím úlohu, která je známá pod názvem *problém batohu* – chceme vybrat některé z modelů tak, aby součet jejich hmotností nepřesáhl m, a zároveň se snažíme maximalizovat součet jejich cen.

Tato úloha je NP-úplná, neznáme tedy algoritmus, který by ji řešil v čase polynomiálním vzhledem k počtu modelů. V našem případě však máme v zadaní zaručeno, že hmotnosti jednotlivých modelů jsou celá kladná čísla a že celková zlodějova nosnost je rozumně nízká. To nám umožní použít algoritmus založený na myšlence dynamického programování s časovou složitostí $\mathcal{O}(mn)$.

Definujme si $P[i, j]$ jako nejvyšší cenu, kterou dokážeme získat ukradením některých modelů, máme-li na výběr jenom prvních i modelů na vstupu a součet hmotností těch z nich, které vybereme, může být roven nejvýše j. Řešením úlohy pak bude hodnota $P[n, m]$.

Hodnoty $P[0, j]$ jsou pro všechna j rovny 0, jelikož nemáme k dispozici žádný model.

Pro $i \geq 1$ vypočítáme $P[i, j]$ takto: Označme si hmotnost i-tého modelu w_i a jeho cenu c_i. Pokud tento model nevezmeme, pak nejlepší cena, jaké můžeme dosáhnout, je rovna $P[i-1, j]$. Pokud naopak tento model vezmeme, potom nejlepší dosažitelná cena je rovna $c_i + P[i-1, j-w_i]$. Je tomu tak proto, že dostaneme cenu c_i za právě vybranou věc a k tomu si ještě můžeme z prvních $i-1$ modelů vybrat co nejvýhodněji tak, abychom tím nepřesáhli hmotnost $j - w_i$.

Z uvedených dvou možností si vybereme tu výhodnější. Hodnota $P[i, j]$ se teda rovná $\max(c_i + P[i-1, j-w_i], P[i-1, j])$. Na výběr samozřejmě máme jen tehdy, když daný model uneseme, tzn. když $w_i \leq j$. V opačném případě nám zůstává jen jediná možnost – i-tý model nevzít. V programu to ošetříme vhodnou podmínkou.

Všimněte si, že k výpočtu $P[i, j]$ stačí znát hodnoty $P[i-1, j]$ a $P[i-1, j-w_i]$. Proto budeme-li počítat hodnoty P postupně od menších i k větším, budeme už mít v době výpočtu $P[i, j]$ potřebné hodnoty připravené. Takto dostáváme řešení s časovou složitostí $\mathcal{O}(mn)$ – každou z $\mathcal{O}(mn)$ hodnot $P[i, j]$ vypočítáme v konstantním čase.

Kdybychom si chtěli ukládat všechny hodnoty P, potřebovali bychom $\mathcal{O}(mn)$ paměti. To ale není nutné – při výpočtu hodnot P pro prvních i modelů stačí znát pouze hodnoty P pro prvních $i-1$ modelů, starší hodnoty P už nepotřebujeme. Naše implementace dokonce vystačí s jedním jednorozměrným polem, ve kterém postupně od větších j k menším nahrazuje hodnoty $P[i-1, j]$ hodnotami $P[i, j]$. (Rozmyslete si, že si tím nikdy nepřepíšeme informaci, kterou ještě budeme potřebovat.) Paměťová složitost řešení je tedy $\mathcal{O}(m)$.

Pomalejší řešení původní úlohy

Nyní už dokážeme snadno navrhnout funkční a celkem efektivní řešení původní úlohy. Stačí vyzkoušet všech n možností, na který model použít duplikátor. Jakmile duplikátor použijeme, stojíme před obyčejným problémem batohu s $n+1$ modely. Stačí nám tedy n-krát vyřešit problém batohu a ze všech nalezených řešení vybrat to nejlepší. Takové řešení má časovou složitost $\mathcal{O}(n^2 m)$ a paměťovou složitost $\mathcal{O}(m+n)$.

Vzorové řešení původní úlohy

Rychlejší řešení úlohy s duplikátorem získáme vhodným rozšířením dynamického programování, které používáme ve zjednodušené úloze. Definujme si $Q[i, j]$ jako

nejvyšší cenu, kterou dokážeme získat ukradením některých modelů, máme-li na výběr jenom prvních i modelů, součet jejich hmotností může být nejvýše j a navíc můžeme jednou použít duplikátor.

Hodnoty $Q[0, j]$ jsou stejně jako $P[0, j]$ rovny 0.

Podívejme se nyní, jak lze vyjádřit $Q[i, j]$ pro $i \geq 1$. Opět si označíme hmotnost i-tého modelu w_i a jeho cenu c_i. Oproti výpočtu $P[i, j]$ nám přibyla třetí možnost: tento model můžeme zduplikovat a vzít obě kopie. Takto získáme řešení, jehož optimální cena bude $2c_i + P[i-1, j-2w_i]$. Ze zbývajících $i-1$ modelů nyní chceme vybrat nejdražší podmnožinu, jejíž hmotnost je nejvýše rovna $j - 2w_i$. Duplikátor jsme už použili, takže při výběru těchto $i-1$ modelů ho už použít nesmíme. Hodnotu $Q[i, j]$ tedy určíme následovně:

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

Součástí úlohy bylo i vypsání čísla modelu, který jsme v optimálním řešení zduplikovali. Proto si při výpočtu $Q[i, j]$ pro každé i a j zapamatujeme rovněž toto číslo.

Popsané řešení má časovou složitost $\mathcal{O}(mn)$, tedy stejnou, jako řešení úlohy bez duplikátoru. Opět dokážeme dosáhnout paměťové složitosti $\mathcal{O}(m)$ – stačí postupně pro všechna i vhodně střídavě počítat hodnoty P a Q.

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

int main() {
    int n, m;
    cin >> m >> n;
    vector<int> P(m + 1, 0);
    // V Q máme dvojice (nejvyšší cena, číslo zduplikovaného 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])
                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
             << "nejlepsi cena " << Q[m].first << endl;
    else
        cout << "nezduplikuj nic" << endl
             << "nejlepsi cena " << P[m] << endl;
    return 0;
}
```

Jiné řešení

Pro zadanou úlohu s nejvýše jedním použitím duplikátoru však existuje i jednodušší řešení. Stejně jako v řešení úlohy bez duplikátoru si vypočítáme pouze hodnoty P. (Hodnoty Q zde nebudeme potřebovat.) Kdybychom zduplikovali a ukradli i-tý model, nejvyšší dosažitelná cena by byla $c_i + P[n, m - w_i]$, neboť si stále můžeme vybírat ze všech n modelů, ale už jen do hmotnosti nejvýše $m - w_i$. Vyzkoušíme tedy všech n možností pro zduplikovaný model a vybereme si z nich tu nejlepší.

Časová složitost bude opět $\mathcal{O}(mn)$, paměťová složitost $\mathcal{O}(m + n)$.

P-II-3 DNA

V celém řešení předpokládáme, že platí $d \geq n$. V opačném případě (když existuje více typů bází, než kolik je délka DNA, kterou zpracováváme) je totiž zjevně odpověď 0.

První řešení, které nás hned napadne, je použít hrubou sílu: Pro každý možný souvislý úsek si zjistíme, zda obsahuje každou bázi, nebo ne. To provedeme s časovou složitostí $\mathcal{O}(d^3)$ tak, že si zvolíme začátek a konec úseku (těch je řádově d^2) a jedním průchodem zkoumaného úseku zjistíme, zda je v něm každý typ báze obsažen aspoň jednou.

Toto řešení lze velmi snadno vylepšit. Označme si $p(i)$ první takovou pozici, že úsek od i do $p(i)$ obsahuje všechny typy bází. Potom pro každé $j > p(i)$ zjevně platí, že úsek od i do j také obsahuje všechny typy bází. Stačí tedy, když pro každý možný začátek úseku i určíme jemu odpovídající nejlevější možný konec úseku $p(i)$.

Jak to provést efektivně? Použijeme pole velikosti n, do kterého si budeme pro každou bázi značit, zda jsme ji už viděli. Navíc budeme mít proměnnou `pocet`, v níž si budeme pamatovat, kolik *typů* bází jsme už viděli. Přidání nové báze x do právě zpracovávaného úseku proběhne následovně: Podíváme se do pole, zda jsme už bázi x viděli. Pokud ano, proti předcházejícímu úseku se nic nezměnilo. Jestliže jsme ji ještě neviděli, v poli si ji označíme a zvýšíme proměnnou `pocet`. Jakmile `pocet` dosáhne hodnoty n, našli jsme pro právě zkoumaný začátek nejlevější možný konec úseku. Popsané řešení má časovou složitost $\mathcal{O}(d^2)$.

Vzorové řešení

Výše uvedené řešení stále ještě není optimální. K němu nám pomůže jedno další pozorování: Když $i < j$, potom $p(i) \leq p(j)$. Platnost tohoto tvrzení dokážeme sporem. Kdyby platilo $p(j) < p(i)$, pak v úseku od j do $p(j)$ se vyskytují všechny typy bází. To je ale jen nějaká část úseku od i do $p(i)$, a proto také v úseku od i do $p(j)$ jsou obsaženy všechny typy bází. To je ale spor s předpokladem, že $p(i)$ je nejmenší takový index, pro který úsek začínající na indexu i obsahuje všechny typy bází.

Jak toto pozorování využít? Říká nám vlastně, že když už známe hodnotu $p(i)$, tak při zkoumání úseku začínajícího na pozici $i + 1$ stačí jeho konec hledat od pozice $p(i)$ dále. K tomu ale potřebujeme vědět, které báze se vyskytují v úseku od pozice $i + 1$ do pozice $p(i)$. Pole booleovských proměnných, které jsme použili v předchozím

řešení, nám zde už nestačí – nemůžeme z něho určit, zda se báze z pozice i neopakuje ještě někde mezi pozicemi $i+1$ až $p(i)$.

Tento nedostatek ovšem snadno odstraníme, když místo booleovských proměnných použijeme proměnné celočíselné. V nich si budeme pro každý typ báze pamatovat, *kolikrát* se nachází v právě zkoumaném úseku. V níže uvedeném programu se jedná o pole `vyskyty`. Nadále budeme používat proměnnou `pocet`, v níž budeme mít uložen počet různých typů bází v právě zkoumaném úseku.

Na začátku výpočtu určíme hodnotu $p(1)$, tedy nejlevější možný konec úseku začínajícího prvním prvkem. V okamžiku, kdy určíme hodnotu $p(1)$, máme v poli `vyskyty` pro každý typ báze uložen počet jeho výskytů na pozicích 1 až $p(1)$. Nyní posuneme začátek na pozici 2. Tím nám z aktuálního úseku vypadla báze z pozice 1. Zmenšíme proto o 1 hodnotu na příslušném místě v poli `vyskyty`. Pokud jsme ji snížili na 0, snížíme o 1 také proměnnou `pocet`. Dále pokračujeme stejně jako dříve: dokud je `pocet` menší než n, posouváme konec aktuálního zkoumaného úseku. Takto pokračujeme postupně pro všechny možné začátky tak dlouho, až pro některý začátek zjistíme, že už pro něj žádný konec úseku nevyhovuje.

Podívejme se nyní, jak jsme touto úpravou zlepšili časovou složitost algoritmu. V průběhu výpočtu pouze měníme dva indexy – jeden ukazující na začátek právě zkoumaného úseku a druhý ukazující na jeho konec. Posunutí každého z těchto indexů provedeme v konstantním čase. Každý z indexů projde nejvýše jednou celou posloupnost, a proto je časová složitost tohoto řešení $\mathcal{O}(d)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
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); // pro každou bázi počet výskytů
    int pocet = 0;
    long long vysledek = 0;

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

Jiná dobrá řešení

Místo pole `vyskyty`, které musíme průběžně udržovat, si můžeme některé potřebné údaje vypočítat předem. Například si můžeme pro každou pozici x (od 1 do d) a každý typ báze y (od 1 do n) spočítat, kde nejblíže za pozicí x se vyskytne báze y. Tyto informace umíme přímočaře určit v čase $\mathcal{O}(nd)$ a pomocí nich již snadno implementujeme řešení podobné vzorovému – vždy, když posuneme začátek úseku, najdeme si, kde nejblíže se vyskytuje báze, která z aktuálně zpracovávaného úseku právě vypadla.

Totéž dokážeme provést ještě o něco šikovněji v čase $\mathcal{O}(d)$, čímž získáme jiné optimální řešení. Stačí si například pro každý typ báze předem spočítat seznam pozic, kde se nachází. Toto lze implementovat různými způsoby: buď můžeme mít n disjunktních seznamů (jeden pro každý typ báze), nebo jednoduše pomocné pole délky d, v němž bude pro každou pozici i zaznamenáno, na které následující pozici leží stejná báze, jako je na pozici i.

Jiná možná řešení jsou založena na využití binárního vyhledávání. V takovýchto řešeních si nejprve spočítáme nějaké informace, z nichž potom dokážeme rychle určit, zda je konkrétní úsek vyhovující. Pro každý z d možných začátků potom binárním vyhledáváním určíme jemu odpovídající konec. Nejběžnější časové složitosti takových řešení jsou $\mathcal{O}(d \log d)$, $\mathcal{O}(d \log^2 d)$, případně $\mathcal{O}(dn + d \log d)$.

P-II-4 Zlomkové programy

V obou podúlohách budou řešení založena na podobném principu jako v domácím kole. Na každé prvočíslo v rozkladu aktuální hodnoty se můžeme dívat jako na proměnnou. Je-li například aktuální hodnota rovna $2^4 3^5 5^2$, můžeme to číst následovně: „v proměnné `#2` je uložena hodnota 4, v proměnné `#3` je uložena hodnota 5 a v proměnné `#5` je uložena hodnota 2.“

A jak probíhá krok výpočtu? Hledáme vhodný zlomek – tedy díváme se na jmenovatele a pro každý z nich *otestujeme,* zda jsou v příslušných proměnných dostatečně vysoké hodnoty. Například zlomek se jmenovatelem $2^3 7$ můžeme použít pouze tehdy, když aktuálně máme v proměnné `#2` hodnotu aspoň 3 a v proměnné `#7` hodnotu aspoň 1. Když najdeme první vhodný zlomek, nejprve *snížíme* hodnoty proměnných odpovídajících jeho jmenovateli a potom *zvýšíme* hodnoty jiných proměnných, odpovídajících jeho čitateli.

a) Maximum

Zadání podúlohy si nyní můžeme přeformulovat následovně: Na začátku výpočtu jsou v proměnných `#2` a `#3` uloženy vstupní hodnoty x a y. Naším cílem je vytvořit v proměnné `#5` výstupní hodnotu $\max(x, y)$.

Řešení úlohy nejprve popíšeme slovně:

1. Dokud jsou obě proměnné `#2` a `#3` kladné, obě sniž a zvyš proměnnou `#5`.
2. Dokud je proměnná `#2` kladná, sniž ji a zvyš proměnnou `#5`.
3. Dokud je proměnná `#3` kladná, sniž ji a zvyš proměnnou `#5`.

Z uvedených kroků 2 a 3 se ve skutečnosti vykoná nejvýše jeden, neboť po skončení kroku 1 zůstane alespoň v jedné z proměnných `#2` a `#3` nula.

Tomuto slovnímu popisu odpovídá následující jednoduchý program:

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

Příklad výpočtu pro $n = 144 = 2^4 3^2$:

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

Příklad výpočtu pro $n = 729 = 2^0 3^6$:

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

b) Sčítání

V této úloze potřebujeme sečíst obsah proměnných `#2` a `#3`, ale zároveň také zachovat jejich původní obsah.

Kdyby nebylo třeba zachovat vstupní proměnné, řešení úlohy by bylo snadné:

1. Dokud je proměnná `#2` kladná, sniž ji a zvyš proměnnou `#5`.
2. Dokud je proměnná `#3` kladná, sniž ji a zvyš proměnnou `#5`.

Abychom neztratili původní obsah proměnných, budeme vždy zvyšovat zároveň dvě proměnné: jednak proměnnou `#5`, jednak nějakou novou pomocnou proměnnou. Začátek našeho algoritmu bude tedy vypadat takto:

1. Dokud je proměnná `#2` kladná, sniž ji a zvyš proměnné `#5` a `#43`.
2. Dokud je proměnná `#3` kladná, sniž ji a zvyš proměnné `#5` a `#47`.

Poté už jenom „uklidíme“ – obsah proměnných `#43` a `#47` vrátíme zpět do proměnných `#2` a `#3`:

3. Dokud je proměnná `#43` kladná, sniž ji a zvyš proměnnou `#2`.
4. Dokud je proměnná `#47` kladná, sniž ji a zvyš proměnnou `#3`.

Kdybychom napsali zlomkový program přímo podle uvedeného algoritmu, narazili bychom na podobné problémy jako při řešení druhé podúlohy domácího kola: program by nikdy neskončil. Jakmile by se vykonal krok 3, byla by proměnná `#2` opět kladná, opět by se provedl krok 1 a tak pořád dokola.

Potřebujeme zajistit, aby se při výpočtu našeho zlomkového programu kroky 1 až 4 provedly jen jednou, a to přesně v tomto pořadí. K tomu využijeme hodnotu 7, kterou je zaručeně dělitelné vstupní číslo. Zlomkový program navrhneme tak, aby kroky 1 a 2 prováděl pouze za předpokladu, že je aktuální hodnota dělitelná sedmi (tzn. proměnná `#7` je nenulová). Jakmile oba tyto kroky skončí (tzn. obě proměnné `#2` a `#3` se vynulují), vydělíme aktuální hodnotu sedmi (tzn. vynulujeme proměnnou `#7`). Poté se už mohou provádět pouze kroky 3 a 4.

Nový, přesnější popis našeho algoritmu vypadá následovně:

1. Dokud je proměnná `#7` kladná:
   a) Je-li proměnná `#2` kladná, sniž ji a zvyš proměnné `#5` a `#43`.
   b) Je-li proměnná `#3` kladná, sniž ji a zvyš proměnné `#5` a `#47`.
   c) Jsou-li proměnné `#2` aj `#3` nulové, vynuluj proměnnou `#7`.
2. Dokud je proměnná `#43` kladná, sniž ji a zvyš proměnnou `#2`.
3. Dokud je proměnná `#47` kladná, sniž ji a zvyš proměnnou `#3`.

Proměnná `#11` bude našemu programu sloužit jako pomocná při kontrole, zda je proměnná `#7` kladná. Pokaždé, když aktuální hodnotu (v krocích 1 a 2) vydělíme 7, zároveň ji vynásobíme 11. Následně použijeme zlomek $7/11$, abychom opět dostali hodnotu dělitelnou 7.

Výsledný program:

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

Příklad výpočtu pro $n = 2^1\cdot 3^2\cdot 7$:

$$2^1\cdot 3^2\cdot 7 \xrightarrow{1} 3^2\cdot 5^1\cdot 11\cdot 43^1 \xrightarrow{3} 3^2\cdot 5^1\cdot 7\cdot 43^1 \xrightarrow{2} 3^1\cdot 5^2\cdot 11\cdot 43^1\cdot 47^1 \xrightarrow{3}$$
$$\xrightarrow{3} 3^1\cdot 5^2\cdot 7\cdot 43^1\cdot 47^1 \xrightarrow{2} 5^3\cdot 11\cdot 43^1\cdot 47^2 \xrightarrow{3} 5^3\cdot 7\cdot 43^1\cdot 47^2 \xrightarrow{4} 5^3\cdot 43^1\cdot 47^2 \xrightarrow{5}$$
$$\xrightarrow{5} 2^1\cdot 5^3\cdot 47^2 \xrightarrow{6} 2^1\cdot 3^1\cdot 5^3\cdot 47^1 \xrightarrow{6} 2^1\cdot 3^2\cdot 5^3.$$

Prvočísla $2, 3, 5, 43, 47$ zde mají pro větší názornost vždy uveden exponent, i když je roven jedné – tyto exponenty odpovídají hodnotám příslušných proměnných.