Riešenia kategórie A

A-I-1 Bezpečná planéta

Všimnime si, že definíciu bezpečnej planéty môžeme preformulovať nasledovne: bezpečné sú tie planéty, z ktorých sa *nedá* docestovať na planétu typu 0 alebo 1. Inými slovami, planéta *nie je* bezpečná práve vtedy, ak sa z nej *dá* docestovať na planétu typu 0 alebo 1.

Množinu všetkých planét, ktoré *nie sú* bezpečné, teda vieme zostrojiť jednoduchým prehľadávaním zadanej siete teleportov. Začneme vo všetkých planétach typu 0 a 1 a odtiaľ pokračujeme proti smeru teleportov – ak planéta x nie je bezpečná a existuje teleport z y na x, tak ani y nie je bezpečná.

Množinu bezpečných planét následne získame ako doplnok množiny planét, ktoré nie sú bezpečné.

Následne zostrojíme všetky znesiteľné planéty. Na to stačí použiť druhé prehľadávanie, opäť proti smeru teleportov. Tentokrát začneme na všetkých bezpečných planétach a pokračovať smieme len cez planéty typu 1 a 2. Takto zjavne nájdeme všetky planéty, odkiaľ vie človek dosiahnuť bezpečnú planétu.

V našom programe používame v oboch prípadoch prehľadávanie do šírky. Časová aj pamäťová zložitosť nášho riešenia je zjavne lineárna od veľkosti vstupu, teda $\Theta(n+m)$.

Existuje aj alternatívne, komplikovanejšie riešenie s rovnakou časovou zložitosťou. Množinu bezpečných planét vieme zostrojiť aj tak, že v zadanom grafe nájdeme silno súvislé komponenty. Planéta je bezpečná práve vtedy, ak jej komponent obsahuje samé planéty typu 2 a navyše platí, že každý teleport vedúci z tohto komponentu vedie na bezpečnú planétu.

Listing programu:

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

int N, M;
vector<int> typy;
vector< vector<int> > teleporty;

vector<bool> prehladaj(const vector<int> start, bool moze0) {
    vector<bool> navstivil(N+1,false);
    queue<int> Q;
    for (unsigned i=0; i<start.size(); ++i) {
        navstivil[ start[i] ] = true;
        Q.push( start[i] );
    }
    while (!Q.empty()) {
        int kde = Q.front(); Q.pop();
        for (unsigned i=0; i<teleporty[kde].size(); ++i) {
            int kam = teleporty[kde][i];
            if (!moze0 && typy[kam]==0) continue;
            if (navstivil[kam]) continue;
            navstivil[kam] = true;
            Q.push(kam);
        }
    }
    return navstivil;
}

int main() {
    cin >> N >> M;
    typy.resize(N+1);
    for (int n=1; n<=N; ++n) cin >> typy[n];
    teleporty.resize(N+1);
    while (M--) {
        int x, y;
        cin >> x >> y;
        teleporty[y].push_back(x);
    }

    // prve prehladavanie: z planet typu 0 a 1 cez cokolvek
```

```
    vector<int> start1;
    for (int n=1; n<=N; ++n) if (typy[n]<2) start1.push_back(n);
    vector<bool> nebezpecne = prehladaj(start1,true);

    // druhe prehladavanie: z bezpecnych planet cez typ 1 a 2
    vector<int> start2;
    for (int n=1; n<=N; ++n) if (!nebezpecne[n]) start2.push_back(n);
    vector<bool> znesitelne = prehladaj(start2,false);

    // vypis vysledku
    cout << "bezpecne:";
    for (int n=1; n<=N; ++n) if (!nebezpecne[n]) cout << " " << n;
    cout << endl << "znesitelne:";
    for (int n=1; n<=N; ++n) if (nebezpecne[n] && znesitelne[n]) cout << " " << n;
    cout << endl;
}
```

A-I-2 Naložená loď

Prvé riešenie, ktoré väčšine ľudí napadne, je priamočiary „pažravý“ postup. Kým sme ešte neposlali všetky balíčky, opakujeme: nájdeme najväčšiu kapsulu, ktorú ešte vieme naplniť, naplníme ju a pošleme.

Problém je, že toto riešenie nemusí použiť najmenší možný počet kapsúl. Pozorní riešitelia si isto všimli, že aj pre jeden z príkladov uvedených v zadaní existuje lepšie riešenie ako to, ktoré vyrobí náš pažravý postup. Budeme na to teda musieť ísť ináč.

Rozumne málo balíčkov

Ukážeme si najskôr riešenie, ktoré bude fungovať pre rozumne malý počet balíčkov k, a potom ukážeme, ako toto riešenie vylepšiť, aby fungovalo aj pre obrovské počty balíčkov. Toto prvé riešenie bude založené na myšlienke dynamického programovania: Označme $M[x]$ najmenší počet kapsúl potrebný na prepravenie presne x balíčkov. Pre tieto hodnoty ľahko nájdeme rekurzívny vzťah: Ak mám prepraviť x balíčkov, musím si vybrať nejakú kapsulu i a tou poslať a_i balíčkov (pričom musí byť $a_i \leq x$). Následne mi zostane ešte $x - a_i$ neposlaných balíčkov, a na tie treba ešte $M[x - a_i]$ kapsúl.

Formálne môžeme tento vzťah zapísať nasledovne:

$$M[x] = 1 + \min_{a_i \leq x} M[x - a_i],$$

pričom nesmieme zabudnúť na začiatočnú podmienku $M[0] = 0$ (na 0 balíčkov zjavne stačí 0 kapsúl). Minimum je v tom vzťahu preto, že my máme na výber, akú kapsulu použijeme, a teda si vyberieme tú, ktorá je v danej situácii pre nás najvýhodnejšia.

Pomocou tohto vzťahu vieme konkrétnu hodnotu $M[x]$ spočítať v čase $O(n)$, ak už poznáme všetky hodnoty $M[0..x-1]$. Ak teda budeme postupne počítať hodnoty $M[1]$, $M[2]$, ..., $M[k]$, dostaneme program, ktorý bude mať časovú zložitosť $O(nk)$.

Práve popísaný program nám síce spočíta hodnotu $M[k]$, teda optimálny počet kapsúl, ktoré treba na poslanie k balíčkov, ale to nám nestačí. My navyše potrebujeme aj zostrojiť jeden konkrétny rozvrh, pri ktorom použijeme práve toľko kapsúl. Ako na to? Zjavne si stačí pre každé x zapamätať to i, pre ktoré je $M[x - a_i]$ minimálne – teda poradové číslo kapsuly, ktorú je najlepšie použiť, ak máme presne x balíčkov. Ak je viac možností pre i, zapamätáme si ľubovoľnú jednu z nich.

Pomocou takto zapamätaných informácií už ľahko zostrojíme jedno optimálne riešenie. Začneme s k balíčkami. Pozrieme sa na optimálnu veľkosť kapsuly pre k balíčkov a použijeme ju. Tým sa nám počet balíčkov zmenší na nejaké k'. Preň sa opäť pozrieme na optimálnu veľkosť kapsuly, použijeme ju, a tak ďalej, až kým počet balíčkov neklesne na nulu. Časová zložitosť tohto zostrojenia riešenia je zjavne $O(k)$, lebo v každom kroku zostávajúci počet balíčkov klesá a každý krok vykonáme v konštantnom čase.

Táto fáza je teda zanedbateľná oproti výpočtu hodnôt $M[0..k]$. Celková časová zložitosť nášho algoritmu zostáva rovná $O(nk)$.

Veľmi veľa balíčkov

Predchádzajúce riešenie pri obmedzení $n \leq 30$ prestáva byť prakticky použiteľné už pri $k = 10^8$. Naše vylepšenie tohto riešenia bude založené na myšlienke, že pre obrovské k sa nám určite oplatí použiť veľakrát najväčšiu kapsulu.

Ako takéto tvrdenie ale dokázať? Predstavme si, že máme napríklad dve kapsuly: jedna na 5 balíčkov, druhá na 7. Čo o nich vieme povedať? Určite sa nám neoplatí použiť 7-krát tú menšiu – namiesto toho by sme použili 5-krát tú väčšiu a tým ušetrili dve kapsuly. Zovšeobecnením tohto príkladu sa dá dokázať nasledujúce tvrdenie: Pre každé $i < n$ platí, že kapsulu číslo i použijeme v každom optimálnom riešení menej ako a_n-krát.

Toto tvrdenie sa však dá ešte zosilniť. Oprieme sa o pomocné tvrdenie z teórie čísel: Nech $s_1, \ldots, s_t$ sú prirodzené čísla. Potom nejaká ich neprázdna podmnožina má súčet deliteľný číslom t. Dôkaz: Uvažujme $t + 1$ súčtov: 0, s_1, $s_1 + s_2$, až $s_1 + \cdots + s_t$. Niektoré dva z nich, nech sú to $s_1 + \cdots + s_i$ a $s_1 + \cdots + s_j$ (pre nejaké $0 \leq i < j \leq t$), dávajú nutne po delení t rovnaký zvyšok. Potom ale $s_{i+1} + \cdots + s_j$ je deliteľné t, a teda sme našli vyhovujúcu podmnožinu.

Čo z tohto tvrdenia pre nás vyplýva? Že v každom optimálnom riešení nanajvýš $(a_n - 1)$-krát použijeme inú ako najväčšiu kapsulu. Totiž ak by sme použili a_n menších kapsúl, podľa práve dokázaného tvrdenia existuje ich podmnožina, ktorej súčet je deliteľný a_n. Namiesto dotyčných menších kapsúl by potom bolo výhodnejšie niekoľkokrát použiť tú najväčšiu.

Hľadanie optimálneho riešenia teda môžeme rozdeliť na dva kroky: Najskôr si pre každý počet balíčkov od 0 po trebárs a_n^2 spočítame, ako ho najlepšie poslať pomocou kapsúl iných ako posledná. Medzi takto spočítanými riešeniami sa určite nachádza aj časť toho celkovo optimálneho. A potom už len vyskúšame všetky možné počty použitia najväčšej kapsuly, ktoré pripadajú do úvahy, a vyberieme z nich ten najlepší. Takéto riešenie má časovú zložitosť $O(na_n^2)$.

Listing programu:

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


int main() {
    // nacitanie vstupu a vypocet hranice, po ktoru pobezi dyn. prog.
    int N; cin >> N;
    vector<long long> A(N);
    for (int n=0; n<N; ++n) cin >> A[n];
    long long K; cin >> K;

    long long K1 = A[N-1]*A[N-1];
    if (K<K1) K1=K;

    // vypocet optimalnych rieseni pre male pocty balickov
    vector<long long> best(K1+1,1LL<<62), how(K1+1,-1);
    best[0]=0;
    for (int k=1; k<=K1; ++k)
        for (int n=0; n<N-1; ++n)
            if (A[n] <= k)
                if (best[k-A[n]]+1 < best[k])
                    best[k]=best[k-A[n]]+1, how[k]=n;

    // vypocet optimalneho riesenia pre K balickov
    long long bestsum = K+1, K2=-1;
    for (int k=0; k<=K1; ++k)
        if ((K-k)%A[N-1]==0)
            if (best[k]+(K-k)/A[N-1] < bestsum)
                bestsum = best[k]+(K-k)/A[N-1], K2=k;

    // vypis riesenia
    cout << bestsum << endl;
    vector<long long> B(N,0);
    B[N-1] = (K-K2) / A[N-1];
    while (K2) { ++B[how[K2]]; K2 -= A[how[K2]]; }
    for (int n=0; n<N; ++n) cout << B[n] << (n==N-1 ? "\n" : " ");
}
```

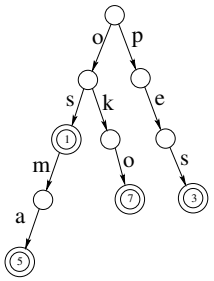
A-I-3 Skladník

Riešenie prvej podúlohy

Hlavnú oštaru nám robí skutočnosť, že veci na vstupe majú mená – reťazce, ktoré musíme spracovať. Keby to boli namiesto reťazcov malé prirodzené čísla, stačilo by v prvej podúlohe použiť obyčajné pole. Takto ale potrebujeme dátovú štruktúru, ktorá nám umožní efektívne meniť množinu reťazcov a pre každý reťazec v nej si pamätať nejaké údaje navyše. (V našom prípade pôjde o jedno celé číslo: aktuálny počet výskytov dotyčného predmetu v sklade.)

Jedno možné efektívne riešenie je použiť dátovú štruktúru nazývanú písmenkový strom, po anglicky *trie*. (Slovo *trie* by sa správne malo čítať *trí*, lebo názov je odvodený od slova *retrieval*. Mnohí ľudia však preferujú výslovnosť *tráj*, aby ho odlíšili od všeobecného stromu (*tree* sa tiež číta *trí*).)

Písmenkový strom je zakorenený strom, v ktorom má každý vrchol najviac 26 synov. Hrany do synov sú označené rôznymi písmenkami (od `a` po `z`). Každému vrcholu v zodpovedá jedna cesta z koreňa nadol, a tou je jednoznačne určené slovo, zodpovedajúce danému vrcholu. (Toto slovo si „prečítame“ na hranách, po ktorých ideme z koreňa do v.)



Obr. 1: Trie zodpovedajúci skladu, v ktorom je: 1× os, 5× osma, 7× oko a 3× pes.

Do písmenkového stromu vieme ľahko uložiť množinu slov. Jednoducho vytvoríme všetky vrcholy, ktoré treba na to, aby sme si mohli na ceste z koreňa dole „prečítať“ každé z našich slov, a následne označíme tie vrcholy, kde niektoré z uložených slov končí. V našom prípade dokonca ani nepotrebujeme nič explicitne označovať. Stačí si v každom vrchole trie pamätať jedno celé číslo: počet vecí so zodpovedajúcim menom, ktoré máme v sklade. Vo vrcholoch, ktorých reťazce nepopisujú veci v sklade, budú jednoducho nuly.

Príklad takéhoto stromu si môžete pozrieť na obrázku 1. Z každého vrcholu vedie v skutočnosti dodola až 26 hrán – tie, ktoré nevedú nikam (NULL pointre), sme pre prehľadnosť nekreslili. Dvojitým krúžkom sú znázornené vrcholy, kde niektoré zo slov končí. V nich sú uvedené čísla, ktoré zodpovedajú počtom daných vecí v sklade. V ostatných vrcholoch sú uložené nuly.

Okrem samotného písmenkového stromu na vyriešenie prvej podúlohy už nepotrebujeme skoro nič. Stačia nám dve celočíselné premenné, v ktorých si pamätáme aktuálny počet rôznych typov vecí a aktuálny počet kusov vecí v sklade. Tieto vieme vždy po spracovaní riadku vstupu v konštantnom čase prepočítať. Pri práci s písmenkovým stromom vieme ľubovoľný reťazec spracovať v čase priamo úmernom jeho dĺžke, teda $O(\ell)$. Taká je teda aj celková časová zložitosť spracovania každého riadku vstupu.

Listing programu:

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

struct trie {
    struct __vrchol {
        int pocet;
        __vrchol *syn[26];
        __vrchol() { pocet=0; memset(syn,0,sizeof(syn)); }
    };

    __vrchol *root;
    int celkovo_typov;
    long long celkovo_veci;

    trie() { root = new __vrchol(); celkovo_typov = celkovo_veci = 0; }

    void update(const string &S, int add) {
        // najdeme a ak treba vyrobime zodpovedajuci vrchol
        __vrchol *kde = root;
        for (unsigned i=0; i<S.size(); ++i) {
            int idx = S[i]-'a';
            if (! kde->syn[idx]) kde->syn[idx] = new __vrchol();
            kde = kde->syn[idx];
```

```
        }
        // upravime pamatanu hodnotu
        celkovo_veci += add;
        if (kde->pocet == 0) ++celkovo_typov;
        kde->pocet += add;
        if (kde->pocet == 0) --celkovo_typov;
    }
};

int main() {
    trie T;
    int zmena;
    string nazov;
    while (cin >> zmena >> nazov) {
        T.update(nazov,zmena);
        cout << "kusov: " << T.celkovo_veci << ", typov: " << T.celkovo_typov << endl;
    }
}
```

(Práve uvedené riešenie ešte nemá optimálnu pamäťovú zložitosť. Vylepšiť ho vieme nasledovne: vždy, keď zo skladu nejakú vec úplne odstránime, zmažeme tie vrcholy písmenkového stromu, ktoré sú v danej chvíli zbytočné. Takto dosiahneme riešenie, ktorého pamäťová zložitosť je $O(\ell t)$, teda lineárna od súčtu dĺžok názvov vecí aktuálne prítomných v sklade. Program bude kvôli úspore miestom uvedený len v elektronickej verzii týchto riešení.)

Riešenie druhej podúlohy

Jednou možnosťou, ako túto podúlohu riešiť, je použiť riešenie z prvej podúlohy a len si navyše pamätať, ktorej veci je momentálne v sklade najviac kusov. Takéto riešenie však príliš efektívne nebude. Ako vyzerá jeho najhorší prípad? Ten nastane zakaždým, keď zo skladu odnesú niekoľko kusov tej veci, ktorej je v danej chvíli najviac. Keďže o počtoch ostatných vecí nič netušíme, musíme ich prezrieť všetky. V najhoršom možnom prípade teda budeme musieť nájsť najväčší spomedzi t záznamov, čo sa dá spraviť s časovou zložitosťou $O(\ell t)$.

Ako sa dá takémuto zlému prípadu vyhnúť? Záznamy o veciach sa oplatí *udržiavať usporiadané* podľa aktuálneho počtu kusov. Pri každej operácii potom obetujeme trochu času na „upratanie“ – preusporiadanie záznamov tak, aby boli naďalej usporiadané. Odmenou nám bude záruka, že každú požiadavku spracujeme rozumne rýchlo, bez nutnosti prezerať zbytočne veľa záznamov.

Aké teda požiadavky máme na novú dátovú štruktúru? Potrebujeme vedieť efektívne robiť dve veci: Prvou je nájsť záznam zodpovedajúci konkrétnemu typu a zmeniť v ňom počet kusov. No a druhou je nájdenie záznamu, ktorý má momentálne najväčší počet kusov. Najjednoduchším riešením bude k písmenkovému stromu pridať ešte jednu dátovú štruktúru: *haldu*.[1]

V halde budeme mať pre každý typ vecí jeden záznam, a v ňom si budeme pamätať názov daného typu a počet jeho kusov. Záznamy v halde budú usporiadané podľa zadania – teda primárne podľa počtu kusov a sekundárne podľa abecedy. Navyše budú naše dve dátové štruktúry medzi sebou prepojené: v každom vrchole písmenkového stromu si budeme pamätať, kde je zodpovedajúci záznam v halde (ak existuje) a v každom zázname haldy si budeme pamätať, ktorému vrcholu písmenkového stromu zodpovedá.

Spracovanie každej inštrukcie potom bude vyzerať nasledovne:

1. Načítame inštrukciu.
2. Podľa názvu typu vecí nájdeme zodpovedajúci vrchol v písmenkovom strome.
3. Pomocou hodnoty, ktorú si v danom vrchole pamätáme, nájdeme zodpovedajúci záznam v halde.
4. Príslušne upravíme počet výskytov daného typu vecí.
5. Zmenou v predchádzajúcom kroku sme mohli porušiť podmienku haldy. Zmenený záznam preto v prípade potreby presunieme haldou dohora alebo dodola tak, aby sme opäť dostali platnú haldu.

Najpomalšou časťou riešenia je posledný krok, v ktorom upravujeme haldu. Počet záznamov v halde je t, jej hĺbka je teda $O(\log t)$. Toľkokrát môže byť potrebné zmenený záznam presunúť o úroveň vyššie či nižšie.

[1] Popis haldy neuvádzame. Záujemcom odporúčame napríklad text o haldách na adrese `http://foja.dcs.fmph.uniba.sk/ads/materialy.php`.

Zakaždým musíme daný záznam porovnať so záznamom bezprostredne nad ním a dvomi bezprostredne pod ním, a na každé toto porovnanie treba $O(\ell)$ času – v prípade rovnosti počtov totiž môže byť potrebné porovnať aj názvy. Celková časová zložitosť úpravy haldy je teda $O(\ell \log t)$.

Listing programu:

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

struct __vrchol {
    __vrchol *syn[26], *otec;
    int index_halda;
    __vrchol(__vrchol* otec=NULL) : otec(otec) { index_halda=-1; memset(syn,0,sizeof(syn)); }
};

struct __zaznam {
    __vrchol *trie_vrchol;
    string nazov;
    int pocet;
    __zaznam(__vrchol *tv, string n, int p) : trie_vrchol(tv), nazov(n), pocet(p) { }
};

struct trie {
    __vrchol *root;

    trie() { root = new __vrchol(); }

    __vrchol *find(const string &S) {
        // najde (a ak treba, vyrobi) vrchol zodpovedajuci retazcu S
        __vrchol *kde = root;
        for (unsigned i=0; i<S.size(); i++) {
            int idx = S[i]-'a';
            if (! kde->syn[idx]) kde->syn[idx] = new __vrchol(kde);
            kde = kde->syn[idx];
        }
        return kde;
    }
};

bool operator< (const __zaznam &A, const __zaznam &B) {
    if (A.pocet != B.pocet) return A.pocet < B.pocet; return A.nazov > B.nazov;
}

struct halda_a_trie {
    trie T;
    vector<__zaznam> H; // halda

    void vymen(int x, int y) {
        // vymeni v halde zaznamy na poziciach x a y, zaroven patricne upravi trie
        __vrchol *tx = H[x].trie_vrchol, *ty = H[y].trie_vrchol;
        swap(H[x],H[y]);
        tx->index_halda = y; ty->index_halda = x;
    }

    int uprac_haldu(int x) {
        // presuva prvok x dohora alebo dodola podla potreby, az kym nie je halda napravena
        while (true) {
            int y=x;
            if (x>0 && H[(x-1)/2]<H[y]) y=(x-1)/2;
            if (2*x+1 < H.size() && H[y]<H[2*x+1]) y=2*x+1;
            if (2*x+2 < H.size() && H[y]<H[2*x+2]) y=2*x+2;
            if (y != x) { vymen(x,y); x=y; } else break;
        }
        return x;
    }

    void update(const string &nazov, int add) {
        __vrchol *kde = T.find(nazov);
        // ak treba, pridame novy zaznam do haldy, nasledne upravime zaznam v halde
        if (kde->index_halda==-1) { kde->index_halda=H.size(); H.push_back(__zaznam(kde,nazov,0)); }
        H[ kde->index_halda ].pocet += add;
        int x = uprac_haldu(kde->index_halda);
        // ak sme zmensili pocet na nulu, vymazeme zaznam z haldy
        if (H[x].pocet==0) {
```

```
            vymen(x,H.size()-1);
            H[H.size()-1].trie_vrchol->index_halda=-1;
            H.pop_back();
            uprac_haldu(x);
        }
    }
};

int main() {
    halda_a_trie HT;
    int zmena;
    string nazov;
    while (cin >> zmena >> nazov) {
        HT.update(nazov,zmena);
        if (HT.H.empty() || HT.H[0].pocet==0) cout << endl;
        else cout << HT.H[0].nazov << " " << HT.H[0].pocet << endl;
    }
}
```

(Alternatívnym riešením by bolo namiesto haldy použiť vyvažovaný binárny strom. V jazykoch, ktoré majú takúto dátovú štruktúru v štandardnej knižnici, sa toto riešenie implementuje ľahšie. Jeho časová zložitosť je rovnaká.)

A-I-4 Zlomkové programy

Testovanie rovnosti

Na vstupe je číslo n tvaru 2^x3^y5, pričom $x, y \geq 0$. Našou úlohou je napísať program, ktorý ho prerobí na číslo 5, ak $x = y$, resp. na číslo 7, ak $x \neq y$.

Základom programu bude zlomok $1/6$. Zakaždým, keď ten použijeme vo výpočte, zmenšíme tým x aj y o jedna. Ak teda na začiatku $x = y$, nič iné ani nepotrebujeme: po x použitiach zlomku $1/6$ dostaneme číslo 5 a môžeme skončiť.

Čo sa stane, ak $x \neq y$? Program, obsahujúci jediný zlomok $1/6$, by sa po konečnom počte krokov zasekol, lebo by sa mu už minuli prvočísla jedného typu. Ak by napríklad bolo $x > y$, program by skončil s hodnotou $2^{x-y}5$.

Ak nastane táto (alebo opačná) situácia, potrebujeme v prvočíselnom rozklade nášho čísla vyrobiť 7 a následne sa zbaviť všetkého okrem tejto 7. Prvý krok zabezpečia zlomky $7/(2 \cdot 5)$ a $7/(3 \cdot 5)$ a následný druhý krok zlomky $1/2$ a $1/3$. (Tieto musia byť uvedené až na konci programu, aby sa nepoužili skôr.)

Celý program teda vyzerá nasledovne:

$$\left(\frac{1}{6}, \frac{7}{10}, \frac{7}{15}, \frac{1}{2}, \frac{1}{3} \right)$$

Príklad výpočtu pre $n = 2^4 3^4 5$:

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

Príklad výpočtu pre $n = 2^2 3^5 5$:

$$2^2 3^5 5 \xrightarrow{1} 2^1 3^4 5 \xrightarrow{1} 3^3 5 \xrightarrow{3} 3^2 7 \xrightarrow{5} 3^1 7 \xrightarrow{5} 7$$

Delenie dvoma

Na vstupe je číslo n tvaru 2^x3, pričom $x \geq 0$. Našou úlohou je napísať program, ktorý ho prerobí na číslo tvaru 2^y, kde $y = \lfloor x/2 \rfloor$.

Toto samozrejme nepôjde robiť tak, že priamo zmeníme mocninu 2, ktorá naše číslo delí. Jednoduchšie by bolo použiť ako „pomocnú premennú“ exponent nejakého iného prvočísla, napríklad čísla 7.

Mohli by sme teda napríklad v programe použiť zlomok 7/4. Každé jeho použitie zníži mocninu 2 o dve a zvýši mocninu 7 o jedno. Časom by sme sa takto dopracovali buď k aktuálnej hodnote $2^1 7^y 3$ alebo $7^y 3$, podľa parity x.

Zlomok 7/4 však priamo použiť nemôžeme. Prečo? Lebo potrebujeme, aby náš výpočet skončil. Ale ak na konci opäť vyrobíme číslo, ktoré je mocninou 2, výpočet by neskončil, lebo by sa znovu dal použiť tento zlomok.

Tu prichádza k slovu číslo 3, ktorým je vstup deliteľný. Prítomnosť prvočísla 3 budeme chápať ako signál, že ešte sme v prvej fáze výpočtu, kedy premieňame 2ky na 7ky. Namiesto menovateľa 4 budeme teda používať menovateľ $4 \cdot 3 = 12$. Zlomok 7/12 však tiež ešte nie je presne to, čo potrebujeme – dal by sa použiť len raz, lebo jeho použitím odstránime prvočíslo 3 z rozkladu aktuálnej hodnoty. Potrebujeme ešte vedieť toto odstránené prvočíslo vrátiť späť.

Fungovať bude napríklad nasledujúca konštrukcia: namiesto jedného zlomku 7/4 použijeme dvojicu zlomkov $(7 \cdot 5)/(4 \cdot 3)$ a 3/5. Postupné použitie týchto dvoch zlomkov bude mať rovnaký efekt ako použitie zlomku 7/4, ale uskutočniť sa môže len vtedy, keď je na začiatku aktuálna hodnota deliteľná 3-kou.

A už stačí len doriešiť upratovanie. Za tieto dva zlomky pridáme zlomky 1/6 a 1/3, ktoré sa zbavia deliteľa 3 a prípadného posledného deliteľa 2, ak x bolo nepárne. Takto dostávame program, ktorý z čísla $2^x 3$ vyrobí číslo 7^y, kde $y = \lfloor x/2 \rfloor$. A už stačí len na konci zmeniť 7ky na 2ky zlomkom 2/7.

Celý program teda vyzerá nasledovne:

$$\left(\frac{35}{12}, \frac{3}{5}, \frac{1}{6}, \frac{1}{3}, \frac{2}{7} \right)$$

Príklad výpočtu pre $n = 2^7 3$:

$$\begin{aligned} 2^7 3 &\xrightarrow{1} 2^5 7^1 5 \xrightarrow{2} 2^5 7^1 3 \xrightarrow{1} 2^3 7^2 5 \xrightarrow{2} 2^3 7^2 3 \xrightarrow{1} 2^1 7^3 5 \\ &\xrightarrow{2} 2^1 7^3 3 \xrightarrow{3} 7^3 \xrightarrow{4} 2^1 7^2 \xrightarrow{4} 2^2 7^1 \xrightarrow{4} 2^3 \end{aligned}$$

DVADSIATY SIEDMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Autori úloh: Vladimír Boža, Michal Forišek, Peter Fulla, Ján Katrenič
Recenzent: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2011