61. ročník Matematické olympiády – 2011/2012

Řešení úloh domácího kola kategorie P

P-I-1 Bezpečná planeta

Všimněte si, že definici bezpečné planety můžeme přeformulovat následovně: bezpečné jsou ty planety, z nichž *není možné* docestovat na planetu typu 0 nebo 1. Jinými slovy, planeta není bezpečná právě tehdy, když z ní lze docestovat na planetu typu 0 nebo 1.

Množinu všech planet, které *nejsou* bezpečné, můžeme tedy sestrojit jednoduchým prohledáváním zadané sítě teleportů. Začneme na všech planetách typu 0 a 1 a postupujeme z nich proti směru teleportů. Jestliže planeta x není bezpečná a existuje teleport z y do x, potom ani planeta y není bezpečná. Množinu bezpečných planet následně získáme jako doplněk množiny planet, které nejsou bezpečné.

Nyní určíme všechny snesitelné planety. K tomu stačí použít druhé prohledávání, opět proti směru teleportů. Tentokrát začneme na všech bezpečných planetách a budeme postupovat pouze přes planety typu 1 a 2. Takto jistě najdeme všechny planety, z nichž se lze dostat na některou bezpečnou planetu, aniž bychom cestou navštívili nějakou neobyvatelnou planetu.

V našem programu používáme v obou případech prohledávání do šířky. Časová i paměťová složitost našeho řešení je zjevně lineární vzhledem k velikosti vstupu, tedy $\Theta(n+m)$.

Existuje i jiné, komplikovanější řešení se stejnou časovou složitostí. Množinu bezpečných planet můžeme sestrojit tak, že v zadaném grafu určíme silně souvislé komponenty. Planeta je bezpečná právě tehdy, když její komponenta obsahuje samé planety typu 2 a navíc platí, že každý teleport vedoucí z této komponenty vede na bezpečnou planetu.

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

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

vector<bool> prohledej(const vector<int> start, bool muze0) {
    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 (!muze0 && 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);
    }

    // První prohledávání: z planet typu 0 a 1 přes cokoliv
    vector<int> start1;
    for (int n=1; n<=N; ++n) if (typy[n]<2) start1.push_back(n);
    vector<bool> nebezpecne = prohledej(start1,true);

    // Druhé prohledávání: z bezpečných planet přes typ 1 a 2
    vector<int> start2;
    for (int n=1; n<=N; ++n) if (!nebezpecne[n]) start2.push_back(n);
    vector<bool> snesitelne = prohledej(start2,false);

    // Výpis výsledku
    cout << "bezpecne:";
    for (int n=1; n<=N; ++n) if (!nebezpecne[n]) cout << " " << n;
    cout << endl << "snesitelne:";
    for (int n=1; n<=N; ++n) if (nebezpecne[n] && snesitelne[n]) cout << " " << n;
    cout << endl;
    return 0;
}
```

P-I-2 Naložená loď

První řešení, které většinu lidí napadne, je přímočarý „hladový" postup. Dokud jsme ještě neposlali všechny balíčky, opakujeme: najdeme největší kapsli, jakou ještě můžeme použít, naplníme ji balíčky a pošleme. Toto řešení ale není správné, jelikož nemusí použít nejmenší možný počet kapslí. Mohli jste si všimnout, že třeba pro druhý z příkladů uvedených v zadání existuje lepší řešení než to, které získáme tímto hladovým postupem. Budeme na to tedy muset jít jinak.

Rozumně málo balíčků

Ukážeme si nejprve řešení, které bude fungovat pro rozumně malý počet balíčků k. Potom předvedeme, jak lze toto řešení vylepšit, aby fungovalo i pro obrovské počty balíčků. Řešení bude založeno na myšlence dynamického programování. Označme $M[x]$ nejmenší počet kapslí potřebný k přepravě přesně x balíčků. Pro výpočet hodnot $M[x]$ snadno najdeme rekurzivní vztah. Máme-li přepravit x balíčků, musíme

si nejprve vybrat nějakou kapsli i a tou poslat a_i balíčků (přičemž musí být $a_i \leq x$). Poté nám zůstane ještě $x - a_i$ neposlaných balíčků a k jejich přepravě potřebujeme dalších $M[x - a_i]$ kapslí. Formálně můžeme tento vztah zapsat následovně:

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

Nesmíme zapomenout na počáteční podmínku $M[0] = 0$ (neboť na 0 balíčků zjevně stačí 0 kapslí). Minimum je v uvedeném vztahu proto, že si můžeme vybrat, jakou kapsli použijeme, a tak zvolíme tu, která je v dané situaci pro nás nejvýhodnější.

Pomocí tohoto vztahu můžeme spočítat konkrétní hodnotu $M[x]$ v čase $\mathcal{O}(n)$, jestliže už známe všechny hodnoty $M[0 \ldots x-1]$. Budeme-li tedy postupně počítat hodnoty $M[1], M[2], \ldots, M[k]$, dostaneme program, který bude mít časovou složitost $\mathcal{O}(nk)$.

Popsaným postupem určíme hledanou hodnotu $M[k]$, tedy optimální počet kapslí potřebný na přepravu k balíčků, ale to nám nestačí. My navíc potřebujeme sestrojit jeden konkrétní rozvrh přepravy, který používá právě tolik kapslí. K tomu stačí pamatovat si pro každé x to číslo i, pro něž je $M[x - a_i]$ minimální – tedy pořadové číslo kapsle, kterou je nejvýhodnější použít, když máme přesně x balíčků. Existuje-li více možností, zapamatujeme si jednu libovolnou z nich.

Pomocí takto zapamatovaných informací již snadno sestrojíme jedno optimální řešení. Začneme s k balíčky. Podíváme se na optimální velikost kapsle pro k balíčků a použijeme ji. Tím se nám počet zbývajících balíčků zmenší na nějaké k'. Pro něj se opět podíváme na optimální velikost kapsle, použijeme ji, a tak dále, dokud počet zbývajících balíčků neklesne na nulu. Časová složitost této části řešení je $\mathcal{O}(k)$, neboť v každém kroku počet zbývajících balíčků klesá a každý krok vykonáme v konstantním čase.

Tato fáze výpočtu je tedy zanedbatelná oproti výpočtu hodnot $M[0 \ldots k]$. Celková časová složitost našeho algoritmu proto zůstává $\mathcal{O}(nk)$.

Velký počet balíčků

Předcházející řešení při omezení $n \leq 30$ přestává být prakticky použitelné již pro $k = 10^8$. Naše vylepšení tohoto řešení bude založeno na myšlence, že pro obrovské k se nám jistě vyplatí použít mnohokrát největší kapsli.

Jak ale takové tvrzení dokázat? Představte si, že máme například dvě kapsle – jedna je na 5 balíčků, druhá na 7 balíčků. Určitě se nám nevyplatí použít 7-krát menší z nich; místo toho použijeme raději 5-krát tu větší a tím ušetříme dvě kapsle. Zobecněním tohoto příkladu dostáváme následující tvrzení: Pro každé $i < n$ platí, že kapsli číslo i použijeme v každém optimálním řešení méně než a_n-krát.

Toto tvrzení lze ještě zesílit. Využijeme pomocnou větu z teorie čísel: Nechť $s_1, \ldots, s_t$ jsou přirozená čísla. Potom nějaká jejich neprázdná podmnožina má součet dělitelný číslem t.

Důkaz: Uvažujme $t+1$ součtů 0, s_1, $s_1 + s_2$, až $s_1 + \ldots + s_t$. Některé dva z nich, nechť to jsou $s_1 + \ldots + s_i$ a $s_1 + \ldots + s_j$ (pro nějaká $0 \leq i < j \leq t$), dávají nutně po

dělení číslem t stejný zbytek. Potom ale $s_{i+1} + \ldots + s_j$ je dělitelné číslem t, takže jsme našli vyhovující podmnožinu.

Co pro nás z této věty vyplývá? To, že v každém optimálním řešení nejvýše $(a_n - 1)$-krát použijeme jinou než největší kapsli. Pokud bychom totiž použili a_n menších kapslí, pak podle právě dokázané věty existuje jejich podmnožina, jejíž součet je dělitelný číslem a_n. Místo dotyčných menších kapslí by proto bylo možné a výhodnější několikrát použít největší kapsli.

Hledání optimálního řešení můžeme tedy rozdělit do dvou kroků. Nejprve si pro každý počet balíčků od 0 do a_n^2 spočítáme, jak ho lze nejlépe přepravit, aniž bychom použili poslední kapsli. Mezi takto spočítanými řešeními se jistě nachází i část toho celkově optimálního. Potom už jenom vyzkoušíme všechny možné počty použití největší kapsle, které připadají do úvahy, a vybereme z nich nejlepší řešení. Tento postup má časovou složitost $\mathcal{O}(na_n^2)$.

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

int main() {
    // Načtení vstupu a výpočet hranice, po kterou poběží 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;

    // Výpočet optimálních řešení pro malé počty balíčků
    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;

    // Výpočet optimálního řešení pro K balíčků
    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;

    // Výpis řešení
    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" : " ");
    return 0;
}
```

P-I-3 Skladník

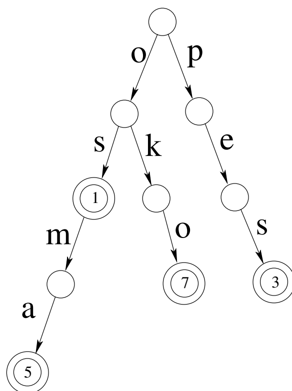
Řešení části a)

Nejvíce starostí nám způsobuje skutečnost, že jednotlivé druhy zboží na vstupu mají jména – znakové řetězce, které musíme zpracovávat. Kdyby se místo nich používala k označení druhů zboží malá přirozená čísla, stačilo by použít v řešení první podúlohy obyčejné pole. Takto ale potřebujeme datovou strukturu, která nám umožní efektivně udržovat množinu řetězců a pro každý z nich si navíc pamatovat nějaké údaje (v našem případě půjde o jedno celé číslo – aktuální počet kusů dotyčného zboží ve skladu).

Jedním možným efektivním řešením je použít datovou strukturu zvanou *písmenkový strom* (anglicky *trie*, čti *trí* – název je odvozen od slova *retrieval*). Písmenkový strom je zakořeněný strom, v němž každý vrchol má nejvýše 26 synů a hrany do synů jsou označeny různými písmeny (od `a` do `z`). Každému vrcholu v odpovídá jedna cesta z kořene dolů a tou je jednoznačně určeno slovo, které odpovídá danému vrcholu. (Toto slovo si „přečteme“ na hranách, po nichž jdeme z kořene do v.)

Do písmenkového stromu dokážeme snadno uložit množinu slov. Jednoduše vytvoříme všechny vrcholy, které jsou třeba k tomu, abychom si mohli na cestě z kořene dolů „přečíst“ každé z našich slov. Označíme ty vrcholy, v nichž některé z uložených slov končí. V našem případě dokonce ani nepotřebujeme nic explicitně označovat. Stačí si v každém vrcholu trie pamatovat jedno celé číslo: počet kusů zboží s příslušným názvem, které máme ve skladu. Ve vrcholech, jejichž řetězce nepopisují zboží ve skladu, budou jednoduše nuly.

Strom odpovídající skladu, v němž je $1 \times$ `os`, $5 \times$ `osma`, $7 \times$ `oko` a $3 \times$ `pes`, by vypadal takto:



Z každého vrcholu vede ve skutečnosti dolů až 26 hran – ty, které nikam nevedou (`NULL` pointery), jsme pro přehlednost nekreslili. Dvojitým kroužkem jsou vyznačeny vrcholy, kde některé ze slov končí. V nich jsou uvedena čísla, která odpovídají počtům kusů příslušného zboží ve skladu. V ostatních vrcholech jsou uloženy nuly.

Kromě samotného písmenkového stromu na vyřešení podúlohy a) už nepotřebujeme téměř nic dalšího. Postačí nám dvě celočíselné proměnné, v nichž si pamatujeme

aktuální počet různých druhů zboží ve skladu a aktuální počet všech kusů ve skladu. Tyto hodnoty dokážeme vždy po zpracování každého řádku ze vstupu v konstantním čase snadno přepočítat. Při práci s písmenkovým stromem zpracujeme libovolný znakový řetězec v čase přímo úměrném jeho délce, tedy v čase $\mathcal{O}(\ell)$. Taková je proto i celková časová složitost zpracování každého řádku vstupu.

```
#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 celkem_druhu;
    long long celkem_kusu;

    trie() { root = new vrchol(); celkem_druhu = celkem_kusu = 0; }

    void update(const string &S, int add) {
        // Najdeme a je-li třeba, vytvoříme odpovídající 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];
        }
        // Upravíme uloženou hodnotu
        celkem_kusu += add;
        if (kde->pocet == 0) ++celkem_druhu;
        kde->pocet += add;
        if (kde->pocet == 0) --celkem_druhu;
    }
};

int main() {
    trie T;
    int zmena;
    string nazev;
    while (cin >> zmena >> nazev) {
        T.update(nazev,zmena);
        cout << "kusu: " << T.celkem_kusu << ", druhu: " << T.celkem_druhu << endl;
    }
    return 0;
}
```

Právě uvedené řešení nemá optimální paměťovou složitost. Můžeme ho vylepšit následovně: vždy, když ze skladu odstraníme poslední kus nějakého druhu zboží, smažeme ty vrcholy písmenkového stromu, které jsou v dané chvíli zbytečné. Tím získáme řešení, jehož paměťová složitost je $\mathcal{O}(\ell t)$, tedy lineární vzhledem k součtu délek názvů těch druhů zboží, které jsou aktuálně ve skladu.

Řešení části b)

Jednou možností, jak lze tuto podúlohu řešit, je použít řešení části a) a jenom si navíc pamatovat, od kterého druhu zboží je momentálně ve skladu nejvíce kusů. Takové řešení ale příliš efektivní nebude. Jak vypadá jeho nejhorší případ? Ten nastane vždy, když ze skladu odebereme několik kusů toho druhu zboží, kterého je v dané chvíli nejvíce. Jelikož o počtech kusů ostatních druhů zboží nic nevíme, musíme je všechny prohlédnout. V nejhorším možném případě tedy budeme muset nalézt největší hodnotu mezi t záznamy, což lze provést s časovou složitostí $\mathcal{O}(\ell t)$.

Jak se můžeme tomuto nepříznivému případu vyhnout? Záznamy o zboží ve skladu se vyplatí *udržovat uspořádané* podle aktuálního počtu kusů. Při každé operaci obětujeme trochu času na „úklid“ – přerovnání záznamů tak, aby byly nadále uspořádány. Odměnou nám bude jistota, že každý požadavek zpracujeme rozumně rychle, bez nutnosti prohlížet zbytečně mnoho záznamů.

V nové datové struktuře potřebujeme umět efektivně provádět dvě operace. První z nich je nalézt záznam odpovídající konkrétnímu druhu zboží a změnit v něm počet kusů. Druhou potřebnou operací je určení záznamu, který má momentálně největší počet kusů. Nejjednodušším řešením bude k písmenkovému stromu přidat ještě druhou datovou strukturu: *haldu.* Popis haldy zde neuvádíme. Zájemcům doporučujeme například text na adrese *http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty*.

V haldě budeme mít uložen pro každý druh zboží jeden záznam, v něm si budeme pamatovat název daného druhu a počet jeho kusů. Záznamy v haldě budou uspořádány podle zadání – tedy primárně podle počtu kusů a sekundárně podle abecedy. Obě naše datové struktury budou mezi sebou navíc provázány: v každém vrcholu písmenkového stromu si budeme pamatovat, kde je uložen jemu odpovídající záznam v haldě (pokud existuje) a v každém záznamu haldy si budeme pamatovat, kterému vrcholu písmenkového stromu odpovídá.

Zpracování jedné instrukce ze vstupu bude vypadat následovně:

- Přečteme ze vstupu jeden řádek s instrukcí.
- Podle druhu zboží najdeme odpovídající vrchol v písmenkovém stromě.
- Pomocí údaje, který si v daném vrcholu písmenkového stromu pamatujeme, najdeme odpovídající záznam v haldě.
- V nalezeném záznamu patřičně upravíme počet kusů daného druhu zboží.
- Změnou údaje v předcházejícím kroku jsme mohli porušit podmínku haldy. Změněný záznam proto v případě potřeby přesuneme haldou nahoru nebo dolů tak, abychom opět dostali platnou haldu.

Nejpomalejší částí řešení je poslední krok, v němž upravujeme haldu. Počet záznamů v haldě je t, její hloubka je tedy $\mathcal{O}(\log t)$. Tolikrát může být potřeba změněný záznam přesunout o úroveň výše nebo níže. Pokaždé musíme přesouvaný záznam porovnat se záznamem bezprostředně nad ním a se dvěma záznamy bezprostředně pod ním, na každé toto porovnání je třeba $\mathcal{O}(\ell)$ času – v případě rovnosti počtů kusů je totiž nutné porovnat i názvy zboží. Celková časová složitost úpravy haldy je proto $\mathcal{O}(\ell \log t)$.

```
#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 nazev;
    int pocet;
    zaznam(vrchol *tv, string n, int p) : trie_vrchol(tv), nazev(n), pocet(p) { }
};

struct trie {
    vrchol *root;

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

    vrchol *find(const string &S) {
        // Najde a je-li třeba, vytvoří vrchol odpovídající řetězci 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.nazev > B.nazev;
}

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

    void vymen(int x, int y) {
        // Vymění v haldě záznamy na pozicích x a y, zároveň patřičně upraví 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 uprav_haldu(int x) {
        // Přesouvá prvek x nahoru nebo dolů podle potřeby, dokud není halda OK
        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 &nazev, int add) {
        vrchol *kde = T.find(nazev);
        // Upravíme záznam v haldě, není-li tam, nejprve ho vytvoříme
        if (kde->index_halda==-1) {
            kde->index_halda=H.size();
            H.push_back(zaznam(kde,nazev,0));
        }
        H[ kde->index_halda ].pocet += add;
        int x = uprav_haldu(kde->index_halda);
        // Pokud jsme zmenšili počet na nulu, vymažeme záznam z haldy
        if (H[x].pocet==0) {
            vymen(x,H.size()-1);
            H[H.size()-1].trie_vrchol->index_halda=-1;
            H.pop_back();
            uprav_haldu(x);
        }
    }
};

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

Místo haldy by také bylo možné použít vyvažovaný binární vyhledávací strom. V programovacích jazycích, které mají tuto datovou strukturu obsaženu ve standardní knihovně, se toto řešení implementuje snáze. Jeho časová složitost je stejná.

Rychlejší řešení části b)

Existuje ovšem efektivnější způsob, jak úlohu vyřešit. Opět použijeme písmenkový strom, ale navíc si do každého jeho vrcholu poznamenáme maximum z počtů kusů všech předmětů, které ve stromu leží pod tímto vrcholem. (Pokud je přímo v tomto vrcholu nějaký předmět, také ho započítáme.)

Tyto hodnoty je snadné udržovat: Kdykoliv změníme počet kusů uložený v nějakém vrcholu stromu, potřebujeme přepočítat ta maxima, která leží na cestě od tohoto vrcholu do kořene stromu. Postupujeme tedy stromem od změněného místa směrem nahoru a každému navštívenému vrcholu upravíme maximum na maximum z jeho počítadla a maxim uložených v jeho synech.

Takto definovaná maxima přitom můžeme využít k nalezení nejčetnějšího druhu zboží. Jeho četnost C je totiž rovna maximu uloženému v kořeni stromu. I jeho jméno je snadné zjistit. Maximum v některém ze synů kořene je totiž rovno číslu C právě

tehdy, nachází-li se v podstromu pod tímto synem některý z nejčetnějších druhů. A jelikož nás zajímá abecedně nejmenší druh zboží, vybereme si nejlevější takový podstrom. V něm můžeme pokračovat stejným způsobem, …, až narazíme na vrchol, jehož počítadlo je rovno C. To je hledaný druh zboží a cesta, po níž jsme tam došli, určuje jeho jméno.

Zbývá odhadnout časovou složitost. Řešení části a), z nějž vycházíme, zpracuje jeden řádek vstupu v čase $\mathcal{O}(\ell)$. My jsme do něj přidali úpravu maxim a hledání nejčetnějšího předmětu. Obojí prochází po jedné cestě v písmenkovém stromu a ta může být dlouhá nejvýše ℓ. Časovou složitost jsme tedy nezhoršili.

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

struct trie {
    struct vrchol {
        int pocet;
        int max;
        vrchol *otec;
        vrchol *syn[26];
        vrchol() { pocet = max = 0; otec = NULL; memset(syn, 0, sizeof(syn)); }
    };

    vrchol *root;
    trie() { root = new vrchol(); }

    void update(const string &S, int add) {
        // Najdeme a je-li třeba, vytvoříme odpovídající 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->syn[idx]->otec = kde;
            }
            kde = kde->syn[idx];
        }

        // Upravíme uloženou hodnotu a přepočítáme maxima
        kde->pocet += add;
        while (kde) {
            kde->max = kde->pocet;
            for (int i=0; i<26; i++)
                if (kde->syn[i] && kde->syn[i]->max > kde->max)
                    kde->max = kde->syn[i]->max;
            kde = kde->otec;
        }
    }

    void vypis_max() {
        vrchol *kde = root;
        int max = root->max;

        // Následujeme maximum a vypisujeme přitom řetězec
        while (kde->pocet != max) {
            int i = 0;
```

```
            while (!kde->syn[i] || kde->syn[i]->max != max)
                i++;
            cout << (char)('a' + i);
            kde = kde->syn[i];
        }

        cout << ' ' << max << endl;
    }
};

int main() {
    trie T;
    int zmena;
    string nazev;
    while (cin >> zmena >> nazev) {
        T.update(nazev,zmena);
        T.vypis_max();
    }
    return 0;
}
```

P-I-4 Zlomkové programy

a) Testování rovnosti

Na vstupu je číslo n tvaru $2^x 3^y 5$, přičemž $x, y \geq 0$. Naším úkolem je napsat zlomkový program, který ho převede na číslo 5, jestliže $x = y$, resp. na číslo 7, pokud $x \neq y$.

Základem programu bude zlomek 1/6. Pokaždé, když ho použijeme ve výpočtu, zmenšíme tím x i y o jedna. Je-li tedy na začátku $x = y$, nic dalšího ani nepotřebujeme: po x násobeních aktuální hodnoty zlomkem 1/6 dostaneme číslo 5 a můžeme skončit.

Co se stane, když $x \neq y$? Program tvořený jediným zlomkem 1/6 by se po konečném počtu kroků zastavil, až by mu došla prvočísla jednoho typu. Pokud by například bylo $x > y$, výpočet by skončil s výslednou hodnotou $2^{x-y}5$.

Jakmile nastane tato (nebo opačná) situace, potřebujeme v prvočíselném rozkladu našeho čísla vytvořit 7 a následně se zbavit všeho kromě této 7. První krok nám zajistí zlomky $7/(2 \cdot 5)$ a $7/(3 \cdot 5)$ a druhý krok zlomky 1/2 a 1/3. Ty musí být uvedeny až na konci programu, aby nebyly použity dříve.

Celý zlomkový program tedy vypadá takto:

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

Příklad výpočtu pro $n = 2^4 3^4 5$:

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

Příklad výpočtu pro $n = 2^2 3^5 5$:

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

b) Dělení dvěma

Na vstupu je číslo n tvaru $2^x 3$, přičemž $x \geq 0$. Naším úkolem je napsat zlomkový program, který ho převede na číslo tvaru 2^y, kde $y = \lfloor x/2 \rfloor$.

Jako „pomocnou proměnnou“ můžeme využít exponent nějakého dalšího prvočísla, například čísla 7. Nabízí se možnost zařadit do programu zlomek 7/4. Každé jeho použití sníží v aktuální hodnotě mocninu 2 o dva a zároveň zvýší mocninu 7 o jedna. Časem bychom se takto měli dopracovat buď k aktuální hodnotě $2^1 7^y 3$, nebo $7^y 3$, podle parity čísla x.

Zlomek 7/4 v programu ale přímo použít nemůžeme. Potřebujeme totiž, aby náš výpočet ve vhodnou chvíli skončil. Když ale máme na konci výpočtu vytvořit výslednou aktuální hodnotu, která je mocninou 2, výpočet by neskončil, neboť by se na ni mohl znovu použít zlomek 7/4.

Využijeme tedy číslo 3, kterým je vstup dělitelný. Přítomnost prvočísla 3 v aktuální hodnotě budeme chápat jako signál, že jsme ještě v první fázi výpočtu, kdy zaměňujeme dvojky za sedmičky. Místo jmenovatele 4 budeme proto používat jmenovatel $4 \cdot 3 = 12$. Zlomek 7/12 však stále ještě není tím, co potřebujeme – mohl by se použít jenom jednou, neboť jeho použitím odstraníme prvočíslo 3 z rozkladu aktuální hodnoty. Potřebujeme ještě umět toto odstraněné prvočíslo vrátit zpět.

Fungovat bude například následující konstrukce: místo jednoho zlomku 7/4 použijeme dvojici zlomků $(7 \cdot 5)/(4 \cdot 3)$ a 3/5. Postupné použití těchto dvou zlomků bude mít stejný efekt, jako použití zlomku 7/4, ale může se provést pouze tehdy, když je aktuální hodnota dělitelná třemi.

Nyní už jenom stačí dořešit úklid. Za tyto dva zlomky přidáme do programu zlomky 1/6 a 1/3, které nás zbaví dělitele 3 a případného posledního dělitele 2, jestliže x bylo liché. Takto dostaneme program, který číslo $2^x 3$ převede na číslo 7^y, kde $y = \lfloor x/2 \rfloor$. Na závěr zbývá ještě změnit všechny sedmičky na dvojky jednoduchým zlomkem 2/7.

Celý zlomkový program vypadá následovně:

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

Příklad výpočtu pro $n = 2^7 3$:

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

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