A-III-4 Rekonštrukcia dvoch máp

Z domáceho kola si už pamätáme podmienku, ktorú musia čísla d_i spĺňať, aby existovalo aspoň jedno riešenie: keďže platná mapa má presne $n-1$ mostov a každý most má dva konce, musia mať čísla d_i súčet presne $2(n-1)$. Teraz potrebujeme zistiť, kedy existujú mapy aspoň dve a kedy je naopak mapa určená jednoznačne.

Vo zvyšku tohto riešenia budeme používať terminológiu z teórie grafov: ostrovy a mosty sú vrcholy stromu, ktorý sa snažíme zrekonštruovať, čísla d_i sú predpísané stupne vrcholov tohto stromu. Vrcholy budeme deliť na listy ($d_i = 1$) a vnútorné ($d_i > 1$).

Z domáceho kola tiež vieme, že vnútorné vrcholy musia všetky „držať pokope“ (odborne: tvoriť podstrom), keďže cez list nevieme prepojiť dve rôzne časti grafu. A vieme, že nech vnútorné vrcholy pospájame do stromu ľubovoľne, vždy k ním vieme jednoznačne doplniť listy. Pri riešení úlohy sa teda stačí pozerať na vnútorné vrcholy a klásť si pre ne otázku: existuje len jeden spôsob, ako ich pospájať, alebo je takých spôsobov viac?

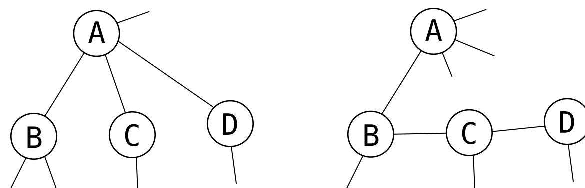
Cesta

Ak máme všetky stupne rovné 2 až na dva ktoré sú rovné 1, existuje presne jedna mapa: hľadaný strom musí byť cesta, ktorá začína a končí vo vrcholoch so stupňom 1 a prechádza všetkými ostatnými.

Nižšie teda môžeme predpokladať, že aspoň jeden vrchol má predpísaný stupeň aspoň 3.

Aspoň štyri vnútorné vrcholy

Ak máme aspoň štyri vnútorné vrcholy (a už vieme, že aspoň jeden má stupeň aspoň 3), vieme ich určite spojiť aspoň dvoma spôsobmi:



(Vrchol A má stupeň aspoň 3, vrcholy B, C, D stupeň aspoň 2. Ak máme ďalšie vrcholy stupňa aspoň 2, tvoria v oboch prípadoch tú istú cestu pripojenú k vrcholu D.)

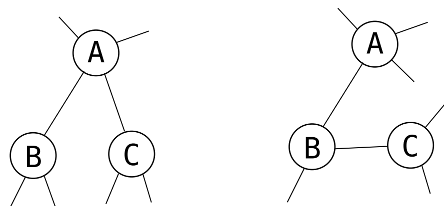
Nanajvýš dva vnútorné vrcholy

Tu je situácia ešte jednoduchšia: riešenie je zjavne vždy jednoznačné.

(Netreba zabudnúť na to, že vnútorných vrcholov môže byť aj nula. Toto nastáva práve vtedy, keď $n = 2$. Teda, aj pre $n = 1$, ale to vylučujú obmedzenia v zadaní.)

Tri vnútorné vrcholy

Tri vrcholy budú vždy ležať na ceste, jediné, čo môžeme meniť, je ich poradie na nej. A tu už ľahko nahliadneme, že ak majú všetky tri predpísaný ten istý stupeň, existuje len jedno riešenie, a vo všetkých ostatných prípadoch existujú riešenia aspoň dve. (Ak nemajú všetky tri vnútorné vrcholy predpísaný ten istý stupeň, existuje vnútorný vrchol, ktorý má iný stupeň ako ostatné dva. Tento vrchol môžeme dať buď na kraj cesty alebo do jej stredu, čím dostaneme dve principiálne rôzne riešenia.)



(Vľavo majú vrcholy na ceste postupne stupne 3, 4, 3, vpravo je to 4, 3, 3.)

Konštrukcia riešenia

Práve sme rozobrali posledný prípad, ktorý môže nastať. Implementáciou testu na vyššie uvedené podmienky vieme získať polovicu bodov za túto úlohu. Všetky vyššie uvedené argumenty sú navyše aj konštruktívne, ak teda podľa nich implementujeme aj konštrukciu rôznych máp, dostaneme aj zvyšné body.

Listing programu (C++)

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

void vypis_strom(const vector< pair<int,int> > hrany) {
    for (auto h : hrany) cout << h.first+1 << " " << h.second+1 << "\n";
}

void vyrob_strom(vector<int> cisla_velkych, vector<int> stupne_velkych, vector<int> cisla_listov, bool hviezda=false) {
    vector< pair<int,int> > hrany;
    int lo = 0;
    if (hviezda) {
        // najdi vnutorny vrchol stupna >= 3 a spoj ho s troma inymi vnutornymi vrcholmi
        int velky = 0;
        while (stupne_velkych[velky] == 2) ++velky;
        swap( cisla_velkych[0], cisla_velkych[velky] );
        swap( stupne_velkych[0], stupne_velkych[velky] );
        for (int i=1; i<=3; ++i) {
            hrany.push_back( { cisla_velkych[0], cisla_velkych[i] } );
            --stupne_velkych[0];
            --stupne_velkych[i];
        }
        lo = 3;
    }
    // pospajaj zvysne vnutorne vrcholy do cesty
    for (unsigned i=lo; i+1<cisla_velkych.size(); ++i) {
        hrany.push_back( { cisla_velkych[i], cisla_velkych[i+1] } );
        --stupne_velkych[i];
        --stupne_velkych[i+1];
    }
    // pridaj listy
    for (unsigned i=0; i<cisla_velkych.size(); ++i) {
        while (stupne_velkych[i]--) {
            int list = cisla_listov.back();
            cisla_listov.pop_back();
            hrany.push_back( { cisla_velkych[i], list } );
        }
    }
    vypis_strom(hrany);
}

int main() {
    int T; cin >> T;
    int N; cin >> N;
    vector<int> cisla_velkych, stupne_velkych, cisla_listov;
    int sumd = 0;
    for (int n=0; n<N; ++n) {
        int d; cin >> d; sumd += d;
        if (d > 1) { cisla_velkych.push_back(n); stupne_velkych.push_back(d); }
        if (d == 1) cisla_listov.push_back(n);
    }
    // osetri pripady kedy existuje nanajvys jedno riesenie
    if (sumd != 2*(N-1)) { cout << "0\n"; return 0; }
    if (N == 2) { cout << "1\n1 2\n"; return 0; }
    int V = cisla_velkych.size();
    int mn = *min_element(stupne_velkych.begin(),stupne_velkych.end());
    int mx = *max_element(stupne_velkych.begin(),stupne_velkych.end());
    if (V <= 2 || mx == 2 || (V == 3 && mx == mn)) {
        cout << "1\n";
        vyrob_strom(cisla_velkych,stupne_velkych,cisla_listov);
        return 0;
    }
    // osetri pripady kedy existuju dve riesenia
    cout << "2\n";
    if (V >= 4) {
        vyrob_strom(cisla_velkych,stupne_velkych,cisla_listov);
        vyrob_strom(cisla_velkych,stupne_velkych,cisla_listov,true);
    } else {
        // dame na kraj cesty vrchol s unikatnym stupnom
        while (stupne_velkych[0] == stupne_velkych[1] || stupne_velkych[0] == stupne_velkych[2]) {
            rotate( cisla_velkych.begin(), cisla_velkych.begin()+1, cisla_velkych.end() );
            rotate( stupne_velkych.begin(), stupne_velkych.begin()+1, stupne_velkych.end() );
        }
        vyrob_strom(cisla_velkych,stupne_velkych,cisla_listov);
        // a nasledne ho dame do stredu cesty
        rotate( cisla_velkych.begin(), cisla_velkych.begin()+2, cisla_velkych.end() );
        rotate( stupne_velkych.begin(), stupne_velkych.begin()+2, stupne_velkych.end() );
```

```
        vyrob_strom(cisla_velkych,stupne_velkych,cisla_listov);
    }
}
```

A-III-5 Veže

V tomto riešení si najskôr ukážeme techniku „kompresie súradníc" a pomocou nej úlohu vyriešime v čase kvadratickom od počtu veží. Potom si ukážeme, ako toto riešenie ďalej zlepšiť.

Nakreslíme do bitmapy

Ak je číslo d, teda rozmer šachovnice, malé, môžeme si v pamäti spraviť pole $d \times d$ a doň zaznačiť polohu jednotlivých veží. Ak potom pre každé políčko pôjdeme dohora, dodola, doľava aj doprava, až kým nenarazíme buď na okraj šachovnice alebo na najbližšiu vežu, dostaneme korektné riešenie s časovou zložitosťou $O(d^3)$.
O čosi lepšie riešenie dostaneme, ak naopak pre každú vežu prejdeme políčka, ktoré ohrozuje, a na tie si zaznačíme jej farbu. Takéto riešenie má časovú zložitosť len $O(d^2)$, keďže každé políčko z každej strany ohrozuje nanajvýš jedna veža, a teda pri značení informácie o vežiach každé políčko navštívime nanajvýš štyrikrát.

Kompresia súradníc

Ak máme veľké d a malé n, bude v bitmape z predchádzajúceho riešenia väčšina riadkov aj stĺpcov prázdna. A čo je ešte dôležitejšie, susedné prázdne stĺpce aj riadky vždy nutne vyzerajú presne rovnako. To znie tak, že by sa tu dalo ušetriť veľa zbytočnej práce.
Predstavme si napríklad, že vieme, že v stĺpcoch 10 až 14 nemáme žiadnu vežu. Týchto päť stĺpcov môžeme nahradiť jedným, ku ktorému si budeme pamätať, že má „váhu" 5.
Ako vieme povedať, ktoré stĺpce obsahujú veže a ktoré nie? Stačí si zobrať všetky čísla stĺpcov, v ktorých veže stoja, a tieto usporiadať. Z takto získaného zoznamu vieme presne povedať aj to, ktoré stĺpce vežu obsahujú, aj to, ktoré bloky po sebe idúcich stĺpcov sú prázdne. No a keďže máme n veží, dostaneme najviac n obsadených stĺpcov a najviac $n+1$ blokov prázdnych stĺpcov.
Keď to isté spravíme aj s riadkami, dostaneme, že namiesto celej šachovnice $d \times d$ si v pamäti stačí vyplniť túto novú, redukovanú tabuľku rozmerov nanajvýš $(2n+1) \times (2n+1)$.
Riešenie, ktoré spraví takúto kompresiu súradníc, a potom do výslednej tabuľky tým šikovnejším spôsobom zaznačí všetky veže, má časovú zložitosť $O(n^2)$ a mohlo získať 5 bodov.

Navzájom rôzne riadky a stĺpce

Takmer zadarmo sa dalo získať dva body, resp. vylepšiť vyššie popísané riešenie na 6-bodové. Na to si stačí uvedomiť, že pre vstupy, kde sú všetky veže v navzájom rôznych riadkoch aj stĺpcoch, existuje vzorec v uzavretom tvare. Odpoveď totiž zjavne nezávisí od toho, v *ktorých konkrétnych* riadkoch a stĺpcoch veže stoja, len od toho, koľko ich je.
Ak máme b bielych a c čiernych veží, máme nutne $v = n - b - c$ voľných riadkov a zároveň aj v voľných stĺpcov. Každá dvojica bielych veží nám určuje dve políčka typu $(2,0)$, každá dvojica bielej a čiernej veže dve políčka typu $(1,1)$, každá dvojica (riadok s bielou vežou, voľný stĺpec) je políčko typu $(1,0)$, a tak ďalej. Riešenie teda vieme z čísel n, b a c vypočítať v konštantnom čase.

Takmer vzorové riešenie

Popíšeme si teraz riešenie, ktoré vedelo získať do 8 bodov. Toto riešenie bude použiteľné pre $d, n \leq 10^6$.
Šachovnicu si rozdelíme na bloky. Každý prázdny riadok bude jeden blok. Ak máme riadok, ktorý obsahuje nejaké veže, bloky budú súvislé úseky prázdnych políčok v ňom. Takýchto blokov teda existuje nanajvýš $d + n$. Naším cieľom bude nájsť spôsob, ako ľubovoľný blok spracovať v čase $O(\log d)$. Ak sa nám toto podarí vymyslieť, zjavne dostaneme dostatočne efektívne riešenie.
Všimnime si teda konkrétny blok. Všetky políčka v tomto bloku majú spoločné to, ktoré veže (ak nejaké) ich ohrozujú zľava a sprava. Ostáva nám teda len zistiť, ako je to s ohrozovaním zhora a zdola.

Povieme, že *polotyp* políčka je to isté ako jeho typ, ale počítajú sa len veže ohrozujúce políčko zdola a zhora.
Keď spracúvame konkrétny blok, potrebujeme o ňom vedieť, koľko políčok ktorého polotypu obsahuje. Z tejto informácie vieme vypočítať, koľko obsahuje políčok ktorého typu, jednoducho tak, že ku každému polotypu pridáme veže, ktoré blok ohrozujú zľava a sprava.
Rôznych polotypov je len šesť. Pre každý polotyp p budeme mať jeden súčtový intervalový strom (alebo jednoduchší Fenwickov strom) s d listami. Každý list stromu zodpovedá jednému políčku práve spracúvaného riadku a je v ňom zapísaná hodnota 1, ak toto políčko má polotyp p, alebo hodnota 0, ak toto políčko tento polotyp nemá. Súčet ľubovoľného úseku potom zjavne zodpovedá počtu políčok v danom úseku, ktoré majú tento konkrétny polotyp.
Celé riešenie teraz bude vyzerať nasledovne:

1. Načítame si súradnice veží. Inicializujeme si šesť stromov – jeden pre každý možný polotyp.
2. Pre každý stĺpec, ktorý neobsahuje vežu, zapíšeme 1 do stromu pre polotyp $(0, 0)$. Pre každý stĺpec, ktorý nejaké veže obsahuje, zapíšeme 1 pre polotyp $(0, f)$, kde f je farba prvej veže v danom stĺpci.
3. Šachovnicu spracúvame riadok po riadku. Spracovanie konkrétneho riadku vyzerá nasledovne:
   (a) Riadok rozdelíme na jednotlivé bloky.
   (b) Pre každý blok: zistíme, koľko obsahuje políčok ktorého polotypu, a z toho vypočítame, koľko obsahuje políčok ktorého typu.
   (c) Pre každú vežu v tomto riadku: upravíme informácie v stromoch. Nech ide o vežu v stlpci s. Doteraz políčka v stĺpci s ohrozovala nejaká veža zhora a práve spracovaná veža zdola. Odteraz to bude táto veža zhora a nasledujúca (ak ešte v stĺpci s nejakú máme) zdola. V strome zodpovedajúcom doterajšiemu polotypu zmeníme 1 na 0 a naopak, v strome zodpovedajúcom novému polotypu zmeníme 0 na 1.

Pre každý z $O(d + n)$ blokov aj pre každú z $O(n)$ veží spravíme konštantne veľa operácií na stromoch, každú z nich v čase $O(\log d)$. Dokopy má teda toto riešenie časovú zložitosť $O((d + n) \log d)$.

Vzorové riešenie

Plné, 10-bodové riešenie, už nevyžaduje žiadne nové myšlienky, je len implementačne náročnejšie. Aby sme dosiahli časovú zložitosť $O(n \log n)$, teda nezávislú od d, potrebujeme spojiť dokopy obe hlavné myšlienky, ktoré sme popísali vyššie. Začneme teda tým, že spravíme kompresiu súradníc pre stĺpce, čím počet stĺpcov zredukujeme z d na $O(n)$. Pre každý stĺpec si budeme pamätať aj váhu, a stromy upravíme tak, aby namiesto súčtov jednotiek vracali súčet im zodpovedajúcich váh. Tým budeme naďalej o každom bloku vedieť povedať, koľko políčok ktorého polotypu obsahuje. Len naše stromy majú teraz $O(n)$ záznamov, preto sa časová zložitosť operácií s nimi zmení z $O(\log d)$ na $O(\log n)$.)
Zvyšok riešenia bude vyzerať rovnako, len opäť spravíme aj kompresiu súradníc pre riadky, aby sme nespracúvali po sebe idúce prázdne riadky každý zvlášť ale všetky naraz.
Takéto riešenie teda postupne spracuje $O(n)$ blokov, o každom v čase $O(\log n)$ zistí, koľko čoho obsahuje, a pomedzi to n-krát zmení v čase $O(\log n)$ informáciu o polotypoch pre nejaký stĺpec. Dokopy teda dostávame sľubovanú časovú zložitosť $O(n \log n)$.

Implementačné detaily

Pre pohodlnejšiu implementáciu si môžeme predstaviť, že biela a čierna sú farby 1 a 2, a okrem nich ešte existuje aj farba 0 (priesvitná?). Veže farby 0 umiestnime pred začiatok a za koniec každého riadku aj stĺpca. Takto je každé políčko ohrozované presne štyrmi vežami a nemusíme ošetrovať žiadne špeciálne prípady.
V autorskom programe potom namiesto šiestich stromov používame deväť: jeden pre každú kombináciu (farba zhora, farba zdola). Opäť, je to o čosi pohodlnejšie a na časovej zložitosti sa to prejaví len malým konštantným faktorom.
V stromoch si nepamätáme váhy. Namiesto toho ošetríme zvlášť polotyp $(0, 0)$. Pre každý iný polotyp vieme, že zaujímavé sú len tie stĺpce, ktoré obsahujú veže, takže môžeme priamo použiť predchádzajúce riešenie a sčitovať jednotky. No a počet políčok polotypu $(0, 0)$ určíme ako počet všetkých políčok v bloku mínus počet políčok iných polotypov.

No a na záver, pri kompresii súradníc v stĺpcoch za zaujímavé považujeme nielen stĺpce obsahujúce veže, ale aj ľavý stĺpec, pravý stĺpec a stĺpce susediace s vežou. Toto je opäť len technický detail dobrý na to, aby sa nám ľahšie indexovalo: každý blok potom začína aj končí v nejakom zaujímavom stĺpci.

Listing programu (C++)

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

struct veza {
    int r, c, farba;
    veza(int r, int c, int t) : r(r), c(c), farba(t) {}
};

bool operator<(const veza &A, const veza &B) { if (A.r != B.r) return A.r < B.r; else return A.c < B.c; }

struct fenwick_tree {
    int size;
    vector<int> T;

    fenwick_tree(int maxval) {
        size=1; while (size < maxval) size <<= 1; T.resize(size+1);
    }

    void update(int x, int delta) { // pre 1 <= x <= init_maxval
        while (x <= size) { T[x] += delta; x += x & -x; }
    }

    int sum(int x1, int x2) { // sucet v uzavretom intervale [x1,x2]
        if (x1 > x2) return 0;
        int res=0;
        --x1;
        while (x2 > 0) { res += T[x2]; x2 -= x2 & -x2; }
        while (x1 > 0) { res -= T[x1]; x1 -= x1 & -x1; }
        return res;
    }
};

void vypis(const vector< vector<long long> > &odpoved) {
    for (int b=0; b<5; ++b) for (int c=0; c<5; ++c) if (odpoved[b][c] > 0) {
        cout << b << " " << c << " " << odpoved[b][c] << endl;
    }
}

void sprav_blok(vector< vector<long long> > &odpoved,
                vector< vector<fenwick_tree> > &FT,
                long long nasob, int povodna_dlzka, int clo, int chi, int fvlavo, int fvpravo) {

    int c00 = povodna_dlzka;

    for (int fhore=0; fhore<3; ++fhore) for (int fdole=0; fdole<3; ++fdole) {
        if (fhore == 0 && fdole == 0) continue;
        vector<int> pocty_farieb(3,0);
        ++pocty_farieb[fvlavo];
        ++pocty_farieb[fvpravo];
        ++pocty_farieb[fhore];
        ++pocty_farieb[fdole];
        int biele = pocty_farieb[1], cierne = pocty_farieb[2];
        int kolko = FT[fhore][fdole].sum(clo,chi);
        odpoved[biele][cierne] += nasob * kolko;
        c00 -= kolko;
    }
    {
        vector<int> pocty_farieb(3,0);
        ++pocty_farieb[fvlavo];
        ++pocty_farieb[fvpravo];
        int biele = pocty_farieb[1], cierne = pocty_farieb[2];
        odpoved[biele][cierne] += nasob * c00;
    }
}

int main() {
    // nacitame vstup
    int D, N;
    cin >> D >> N;
    vector<veza> nacitane_veze;
    for (int n=0; n<N; ++n) {
        int r, c;
        string t;
        cin >> r >> c >> t;
        nacitane_veze.emplace_back( r, c, t == "B" ? 1 : 2);
    }
    sort(nacitane_veze.begin(), nacitane_veze.end());

    map<int, vector<veza> > veze_podla_riadkov;
    for (auto v : nacitane_veze) veze_podla_riadkov[v.r].push_back(v);
```

```
    // spravime kompresiu suradnic pre stlpce
    set<int> rozne_suradnice = {0, D-1};
    for (auto v : nacitane_veze) for (int d=-1; d<=1; ++d) if (0 <= v.c+d && v.c+d < D) rozne_suradnice.insert(v.c+d);
    vector<int> suradnice( rozne_suradnice.begin(), rozne_suradnice.end() );
    unordered_map<int,int> redukuj_stlpec;
    for (int i=0; i<int(suradnice.size()); ++i) redukuj_stlpec[suradnice[i]] = i+1;
    int Y = redukuj_stlpec.size();

    // vyrobime si pole pre odpovede = pocty policok jednotlivych typov
    vector< vector<long long> > odpoved( 5, vector<long long>(5,0) );
    if (N == 0) { odpoved[0][0] = 1LL*D*D; vypis(odpoved); return 0; }

    // vyrobime si stromy pre vsetky kombinacie farieb
    vector< vector<fenwick_tree> > FT(3, vector<fenwick_tree>(3, fenwick_tree(Y+1)));

    // roztriedime si veze podla stlpcov
    // v kazdom stlpci si pridame na spodok priesvitnu zarazku a usporiadame zdola hore
    vector< vector<veza> > veze_podla_stlpcov(Y+1);
    for (auto v : nacitane_veze) veze_podla_stlpcov[ redukuj_stlpec[v.c] ].push_back(v);
    for (int y=1; y<=Y; ++y) {
        veze_podla_stlpcov[y].emplace_back(D,-47,0);
        reverse( veze_podla_stlpcov[y].begin(), veze_podla_stlpcov[y].end() );
    }

    // zatial nastavime ze v kazdom stlpci je farba 0 zhora a farba jeho najvyssej veze zdola
    vector<int> farba_hore(Y+1,0);
    for (int y=1; y<=Y; ++y) FT[0][ veze_podla_stlpcov[y].back().farba ].update(y,+1);

    // postupne prechadzame vsetky riadky
    int pred_riadok = -1;
    for (auto &rec : veze_podla_riadkov) {
        int akt_riadok = rec.first;
        vector<veza> &akt_veze = rec.second;

        // pridame priesvitne zarazky na konce
        akt_veze.insert( akt_veze.begin(), {akt_riadok,-1,0} );
        akt_veze.insert( akt_veze.end(),   {akt_riadok,D,0} );

        // spracujeme prazdne riadky pred aktualnym
        sprav_blok(odpoved, FT, akt_riadok-pred_riadok-1, D, redukuj_stlpec[0], redukuj_stlpec[D-1], 0, 0);
        pred_riadok = akt_riadok;

        // spracujeme bloky prazdnych policok v tomto riadku
        for (unsigned i=0; i+1<akt_veze.size(); ++i) {
            int lo = akt_veze[i].c + 1, hi = akt_veze[i+1].c - 1;
            if (lo > hi) continue;
            sprav_blok(odpoved, FT, 1, hi-lo+1, redukuj_stlpec[lo], redukuj_stlpec[hi], akt_veze[i].farba, akt_veze[i+1].farba);
        }

        // upravime data pre stlpce ktore dostali nove veze
        for (unsigned i=1; i+1<akt_veze.size(); ++i) {
            int c = redukuj_stlpec[ akt_veze[i].c ];
            int f1 = farba_hore[c];
            int f2 = akt_veze[i].farba;
            veze_podla_stlpcov[c].pop_back();
            int f3 = veze_podla_stlpcov[c].back().farba;
            FT[f1][f2].update(c,-1);
            FT[f2][f3].update(c,+1);
            farba_hore[c] = f2;
        }
    }
    if (pred_riadok < D-1) sprav_blok(odpoved, FT, D-pred_riadok-1, D, redukuj_stlpec[0], redukuj_stlpec[D-1], 0, 0);
    vypis(odpoved);
}
```

A-III-6 Cestou na trh

Ak sú vo veršíku spomenuté postupne čísla $a_1, \dots, a_k$, išlo toho dokopy na trh $n = a_1 + a_1a_2 + \cdots + a_1a_2 \dots a_k$. Hlavným pozorovaním, ktoré budeme potrebovať, je všimnúť si, že všetky sčítance na pravej strane sú deliteľné a_1. A teda v každom platnom riešení musí a_1 byť deliteľom n.

Delitele čísla n vieme nájsť v čase priamo úmernom $\sqrt{n}$ vďaka pozorovaniu, že ak d je deliteľom n, tak aj n/d je deliteľom n a spomedzi čísel d a n/d musí aspoň jedno byť $\leq \sqrt{n}$. Stačí teda vyskúšať všetky d po odmocninu z n, a zakaždým, keď nájdeme nejaké, ktoré delí n, našli sme jednu dvojicu deliteľov.

Takto teda vieme nájsť všetky možné hodnoty a_1. Keď si nejakú z nich zvolíme, ako postupovať ďalej? Upravme si vyššie uvedený vzťah nasledovne:

$$n = a_1 + a_1a_2 + \cdots + a_1a_2 \dots a_k$$
$$n/a_1 = 1 + a_2 + a_2a_3 + \cdots + a_2 \dots a_k$$
$$(n/a_1) - 1 = a_2 + a_2a_3 + \cdots + a_2 \dots a_k$$

Na to, aby sme našli optimálne hodnoty $a_2, \dots, a_k$ pre zadané n a nami zvolené a_1, teda stačí optimálne vyriešiť pôvodnú úlohu pre číslo $n/a_1 - 1$.

Na základe vyššie popísaných myšlienok vieme teraz napísať rekurzívnu funkciu, ktorá bude vyššie popísaným postupom skúšať všetky možnosti. Jediným ďalším vylepšením bude pridanie memoizácie: Akonáhle úlohu pre nejaké n vyriešime, toto riešenie si zapamätáme. A ak budeme niekedy v budúcnosti znova poznať riešenie pre toto n, namiesto opakovania výpočtu jednoducho rovno vrátime zapamätané riešenie.

Toto riešenie je už dostatočne rýchle na zisk plného počtu bodov. Presná analýza jeho časovej zložitosti je príliš zložitá, ale aspoň načrtneme, prečo bude pre obmedzenia zo zadania dostatočne rýchle. Čísla nad 10^6 nazvime veľké, ostatné čísla budú malé. Pri každom rekurzívnom volaní hodnotu n niekoľkokrát zmenšíme, preto bude veľkých volaní len rozumne málo. No a aj keby sme malé volania spravili všetky, ešte stále nám nepokazia časovú zložitosť. Navyše, ak nám nejaké n vyrobí veľa rôznych veľkých podproblémov (keďže má veľa malých deliteľov), tieto sa ďalej nebudú vetviť rovnako – totiž ak n bolo deliteľné nejakým p ktoré nedelí a_1, tak aj n/a_1 ním deliteľné je, ale potom $n/a_1 - 1$ ním už deliteľné nebude.

Nižšie uvedená implementácia si ku každému úspešne vyriešenému n ukladá rovno celé riešenie, teda jednu optimálnu postupnosť a_i. Keďže jej dĺžka je nanajvýš logaritmická od n, nespraví ani toto žiadne výrazné spomalenie (hoci by bolo o chlp efektívnejšie ukladať si len jej optimálnu dĺžku a na konci potom už zrekonštruovať len optimálne riešenie pre pôvodné n).

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;
unordered_map<long long, vector<long long> > memo;

vector<long long> delitele(long long n) {
    vector<long long> answer;
    for (long long d=1; d*d<=n; ++d) if (n%d == 0) {
        answer.push_back(d);
        if (d*d < n) answer.push_back(n/d);
    }
    return answer;
}

vector<long long> zostroj_najlepsie(long long n) {
    if (memo.count(n)) return memo[n];
    vector<long long> &answer = memo[n];
    answer.resize(1,n);
    for (long long d : delitele(n)) if (3 <= d && d < n) {
        vector<long long> option = zostroj_najlepsie(d-1);
        if (option.size() >= answer.size()) {
            option.insert( option.begin(), 1 );
            for (auto &x : option) x *= n/d;
            answer = option;
        }
    }
    return answer;
}

int main() {
    long long n; cin >> n;
    vector<long long> answer = zostroj_najlepsie(n);
    cout << answer.size() << endl;
    for (unsigned i=0; i<answer.size(); ++i) cout << answer[i] << (i+1 == answer.size() ? '\n' : ' ');
}
```

TRIDSIATY PIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2020