P-III-4 Rekonstrukce dvou map

Z domácího kola si pamatujeme podmínku, kterou musí čísla d_i splňovat, aby existovalo aspoň jedno řešení: protože platná mapa má přesně $n-1$ mostů a každý most má dva konce, musí mít čísla d_i součet přesně $2(n-1)$. Nyní potřebujeme zjistit, kdy existují mapy alespoň dvě a kdy je naopak mapa určena jednoznačně.

Budeme používat terminologii z teorie grafů: ostrovy a mosty jsou vrcholy a hrany stromu, který se snažíme zrekonstruovat, čísla d_i jsou předepsané stupně vrcholů tohoto stromu. Vrcholy budeme dělit na listy ($d_i = 1$) a vnitřní vrcholy ($d_i > 1$).

Z domácího kola také víme, že vnitřní vrcholy musí všechny „držet pohromadě" (odborněji řečeno: musí tvořit podstrom), jelikož přes list nemůžeme propojit dvě různé části grafu. Víme také, že když vnitřní vrcholy jakkoliv pospojujeme do stromu, vždy k nim můžeme jednoznačně doplnit listy. Při řešení úlohy se tedy stačí dívat na vnitřní vrcholy a klást si otázku: existuje jenom jeden způsob, jak je spojit, nebo je takových způsobů více?

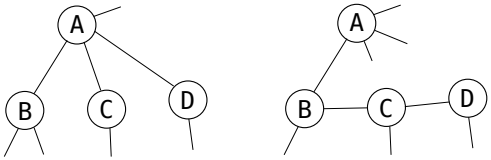
Cesta

Mají-li všechny vrcholy stupeň 2 až na dva, které mají stupeň 1, existuje přesně jedna mapa: hledaným stromem musí být cesta, která začíná a končí ve vrcholech se stupněm 1 a prochází všemi ostatními vrcholy.

Nadále tedy můžeme předpokládat, že alespoň jeden vrchol má stupeň aspoň 3.

Alespoň čtyři vnitřní vrcholy

Jestliže máme alespoň čtyři vnitřní vrcholy (a už víme, že nejméně jeden z nich má stupeň aspoň 3), můžeme je spojit minimálně dvěma způsoby:



(Vrchol A má stupeň aspoň 3, vrcholy B, C, D stupeň aspoň 2. Máme-li další vrcholy stupně aspoň 2, tvoří v obou případech stejnou cestu připojenou k vrcholu D.)

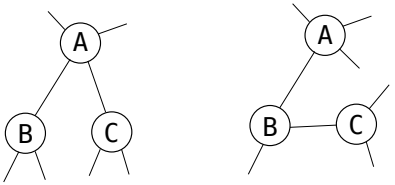
Nejvýše dva vnitřní vrcholy

Zde je situace ještě jednodušší: řešení je zjevně vždy jednoznačné.

Tři vnitřní vrcholy

Tři vrcholy budou vždy ležet na cestě, měnit můžeme pouze jejich pořadí. Snadno nahlédneme, že mají-li všechny tři vrcholy stejný stupeň, existuje jenom

jediné řešení, ve všech ostatních případech existují řešení alespoň dvě. (Když nemají všechny tři vnitřní vrcholy stejný stupeň, existuje vnitřní vrchol, který má jiný stupeň, než zbývající dva. Tento vrchol můžeme umístit buď na kraj cesty, nebo do jejího středu, čímž dostáváme dvě různá řešení.)



(Vlevo mají vrcholy na cestě postupně stupně 3, 4, 3, vpravo je to 4, 3, 3.)

Výsledné řešení

Právě jsme rozebrali poslední případ, který může nastat. Implementací testu na výše uvedené podmínky získáme za tuto úlohu polovinu bodů. Všechny uvedené argumenty jsou navíc i konstruktivní. Pokud tedy podle nich implementujeme také sestrojení příslušných map, dostaneme i zbývající body.

```
#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 vytvor_strom(vector<int> cisla_velkych, vector<int> stupne_velkych,
                  vector<int> cisla_listu, bool hvezda=false) {
    vector< pair<int,int> > hrany;
    int lo = 0;

    if (hvezda) {
        // Najdi vnitřní vrchol stupně >= 3 a spoj ho
        // s třemi jinymi vnitřními vrcholy
        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;
    }

    // Spoj zbývající vnitřní 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];
    }
```

```
    // Přidej listy
    for (unsigned i=0; i < cisla_velkych.size(); ++i)
        while (stupne_velkych[i]--) {
            int list = cisla_listu.back();
            cisla_listu.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_listu;
    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_listu.push_back(n);
    }

    // Ošetři případy, kdy existuje nejvýše jedno řešení
    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";
        vytvor_strom(cisla_velkych,stupne_velkych,cisla_listu);
        return 0;
    }

    // Ošetři případy, kdy existují dvě řešení
    cout << "2\n";
    if (V >= 4) {
        vytvor_strom(cisla_velkych,stupne_velkych,cisla_listu);
        vytvor_strom(cisla_velkych,stupne_velkych,cisla_listu,true);
    } else {
        // Dáme na kraj cesty vrchol s unikátním stupněm
        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());
        }
        vytvor_strom(cisla_velkych,stupne_velkych,cisla_listu);

        // a následně ho dáme do středu cesty
        rotate(cisla_velkych.begin(), cisla_velkych.begin()+2,
               cisla_velkych.end());
        rotate(stupne_velkych.begin(), stupne_velkych.begin()+2,
               stupne_velkych.end());
        vytvor_strom(cisla_velkych,stupne_velkych,cisla_listu);
    }
}
```

P-III-5 Věže

Nejprve si ukážeme techniku „komprese souřadnic“ a s její pomocí úlohu vyřešíme v čase kvadratickém vzhledem k počtu věží. Potom si předvedeme, jak lze toto řešení ještě zlepšit.

Nakreslení do bitmapy

Pokud je číslo d (rozměr šachovnice) malé, můžeme si v paměti vytvořit pole $d \times d$ a do něj zaznačit polohu jednotlivých věží. Když pak pro každé políčko půjdeme nahoru, dolů, doleva a doprava, dokud nenarazíme na okraj šachovnice nebo na nejbližší věž, dostaneme korektní řešení s časovou složitostí $\mathcal{O}(d^3)$.

O něco lepší řešení dostaneme, když naopak pro každou věž projdeme políčka, která tato věž ohrožuje, a tato políčka označíme barvou věže. Takové řešení má časovou složitost jen $\mathcal{O}(d^2)$, neboť každé políčko z každé strany ohrožuje nejvýše jedna věž. Při označování políček tedy navštívíme každé z nich nejvýše čtyřikrát.

Komprese souřadnic

Pokud máme velké d a malé n, bude v bitmapě z předchozího řešení většina řádků a sloupců prázdná. Ještě důležitější je, že sousední prázdné sloupce i řádky vypadají vždy úplně stejně. Zde se dá ušetřit mnoho zbytečné práce.

Představme si například, že ve sloupcích 10 až 14 nemáme žádnou věž. Těchto pět sloupců můžeme nahradit jedním, u něhož si budeme pamatovat, že má „váhu“ 5.

Jak zjistíme, které sloupce obsahují věže a které ne? Stačí si vzít všechna čísla sloupců, v nichž věže stojí, a tato čísla uspořádat. Z takto získaného seznamu umíme přesně určit, které sloupce obsahují věž a které bloky po sobě jdoucích sloupců jsou prázdné. Protože máme celkem n věží, dostaneme nejvýše n obsazených sloupců a nejvýše $n + 1$ bloků prázdných sloupců.

Když totéž provedeme s řádky, zjistíme, že namísto celé šachovnice $d \times d$ stačí vyplnit v paměti tuto novou redukovanou tabulku o rozměrech nejvýše $(2n + 1) \times (2n + 1)$.

Řešení, které provede takovouto kompresi souřadnic a do výsledné tabulky zaznačí všechny věže, má časovou složitost $\mathcal{O}(n^2)$ a mohlo získat 5 bodů.

Navzájem různé řádky a sloupce

Téměř zadarmo bylo možné získat dva body, resp. vylepšit výše uvedené řešení na 6-bodové. Stačí uvědomit si, že pro vstupy, kde jsou všechny věže v navzájem různých řádcích i sloupcích, existuje jednoduchý vzorec. Odpověď totiž zjevně nezávisí na tom, ve *kterých konkrétních* řádcích a sloupcích věže stojí, ale pouze na tom, kolik jich je.

Máme-li b bílých a c černých věží, máme nutně $v = n - b - c$ volných řádků a zároveň také v volných sloupců. Každá dvojice bílých věží určuje dvě políčka typu $(2, 0)$, každá dvojice bílé a černé věže dvě políčka typu $(1, 1)$, každá dvojice (řádek s bílou věží, volný sloupec) je políčko typu $(1, 0)$, a tak dále. Řešení tedy dokážeme spočítat z čísel n, b a c v konstantním čase.

Téměř vzorové řešení

Popíšeme si nyní řešení, za které jste mohli získat až 8 bodů. Toto řešení bude použitelné pro $d, n \leq 10^6$.

Šachovnici si rozdělíme na bloky. Každý prázdný řádek bude jeden blok. Máme-li řádek, který obsahuje nějaké věže, bloky budou souvislé úseky prázdných políček v něm. Takových bloků tedy existuje nejvýše $d+n$. Naším cílem bude nalézt způsob, jak libovolný blok zpracovat v čase $\mathcal{O}(\log d)$. Když se nám toto podaří vymyslet, zjevně dostaneme dostatečně efektivní řešení.

Sledujme tedy nějaký konkrétní blok. Všechna políčka v tomto bloku mají společné to, které věže (pokud nějaké) je ohrožují zleva a zprava. Zbývá zjistit, jak je to s ohrožováním shora a zdola.

Řekneme, že *polotyp* políčka je to totéž jako jeho typ, ale počítají se jenom věže ohrožující políčko zdola a shora.

Když zpracováváme konkrétní blok, potřebujeme o něm vědět, kolik políček kterého polotypu obsahuje. Z této informace umíme určit, kolik obsahuje políček kterého typu – jednoduše tak, že ke každému polotypu přidáme věže, které blok ohrožují zleva a zprava.

Různých polotypů existuje pouze šest. Pro každý polotyp p budeme mít jeden součtový intervalový strom (nebo jednodušší Fenwickův strom) s d listy. Každý list stromu odpovídá jednomu políčku právě zpracovávaného řádku a je v něm zapsána hodnota 1, jestliže toto políčko má polotyp p, nebo hodnota 0, pokud toto políčko tento polotyp nemá. Součet libovolného úseku potom zjevně odpovídá počtu políček v daném úseku, která mají tento konkrétní polotyp.

Celé řešení bude vypadat takto:

1. Přečteme si souřadnice věží. Inicializujeme šest stromů – jeden pro každý možný polotyp.
2. Pro každý sloupec, který neobsahuje věž, zapíšeme 1 do stromu pro polotyp $(0, 0)$. Pro každý sloupec, který nějaké věže obsahuje, zapíšeme 1 pro polotyp $(0, f)$, kde f je barva první věže v daném sloupci.
3. Šachovnici zpracováváme řádek po řádku. Zpracování konkrétního řádku vypadá následovně:
   a. Řádek rozdělíme na jednotlivé bloky.
   b. Pro každý blok: zjistíme, kolik obsahuje políček kterého polotypu, a z toho vypočítáme, kolik obsahuje políček kterého typu.
   c. Pro každou věž v tomto řádku: upravíme informace ve stromech. Nechť se jedná o věž ve sloupci s. Dosud políčka ve sloupci s ohrožovala nějaká věž shora a právě zpracovávaná věž zdola. Nyní to bude tato věž shora a následující (pokud ještě ve sloupci s nějakou máme) zdola. Ve stromu odpovídajícím dosavadnímu polotypu změníme 1 na 0 a naopak, ve stromu odpovídajícím novému polotypu změníme 0 na 1.

Pro každý z $\mathcal{O}(d+n)$ bloků a pro každou z $\mathcal{O}(n)$ věží provedeme konstantní počet operací na stromech, každou z nich v čase $\mathcal{O}(\log d)$. Celkově má toto řešení časovou složitost $\mathcal{O}((d+n)\log d)$.

Vzorové řešení

Plné 10-bodové řešení už nevyžaduje žádné nové myšlenky, je pouze implementačně náročnější. Abychom dosáhli časové složitosti $\mathcal{O}(n \log n)$, tedy nezávislé na d, potřebujeme spojit dohromady obě hlavní myšlenky, které jsme popsali výše. Začneme tedy tím, že provedeme kompresi souřadnic pro sloupce, čímž počet sloupců zredukujeme z d na $\mathcal{O}(n)$. Pro každý sloupec si budeme pamatovat i jeho váhu a stromy upravíme tak, aby místo součtů jednotek vracely součet odpovídajících vah. Tím budeme moci nadále o každém bloku říci, kolik políček kterého polotypu obsahuje. Naše stromy mají nyní $\mathcal{O}(n)$ záznamů, proto se časová složitost operací s nimi změní z $\mathcal{O}(\log d)$ na $\mathcal{O}(\log n)$.

Zbytek řešení bude vypadat stejně, jen opět provedeme také kompresi souřadnic pro řádky, abychom nezpracovávali po sobě jdoucí prázdné řádky každý zvlášť, ale všechny najednou.

Toto řešení tedy postupně zpracuje $\mathcal{O}(n)$ bloků, o každém v čase $\mathcal{O}(\log n)$ zjistí, kolik čeho obsahuje, a přitom n-krát změní v čase $\mathcal{O}(\log n)$ informaci o polotypech pro nějaký sloupec. Celkem tak dostáváme slibovanou časovou složitost $\mathcal{O}(n \log n)$.

Implementační detaily

Pro pohodlnější implementaci si můžeme představit, že bílá a černá jsou barvy 1 a 2, a kromě nich ještě existuje barva 0 (průsvitná). Věže barvy 0 umístíme před začátek a za konec každého řádku i sloupce. Tím je každé políčko ohrožováno přesně čtyřmi věžemi a nemusíme ošetřovat žádné speciální případy.

V ukázkovém programu potom namísto šesti stromů používáme devět: jeden pro každou kombinaci (barva shora, barva zdola). Je to opět o něco pohodlnější a v časové složitosti se to projeví jen malým konstantním faktorem.

Ve stromech si nepamatujeme váhy. Místo toho ošetříme zvlášť polotyp $(0, 0)$. Pro každý jiný polotyp víme, že zajímavé jsou jenom ty sloupce, které obsahují věže, takže můžeme přímo použít předchozí řešení a sčítat jedničky. Počet políček polotypu $(0, 0)$ určíme jako počet všech políček v bloku mínus počet políček jiných polotypů.

Při kompresi souřadnic ve sloupcích za zajímavé považujeme nejen sloupce obsahující věže, ale i levý sloupec, pravý sloupec a sloupce sousedící s věží. Toto je opět pouze technický detail dobrý k tomu, aby se nám snáze indexovalo: každý blok potom začíná i končí v některém zajímavém sloupci.

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

struct vez {
    int r, c, barva;
    vez(int r, int c, int t) : r(r), c(c), barva(t) {}
};
```

```
bool operator<(const vez &A, const vez &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) { // pro 1 <= x <= init_maxval
        while (x <= size) { T[x] += delta; x += x & -x; }
    }

    int sum(int x1, int x2) { // součet v uzavřeném intervalu [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 vytvor_blok(vector< vector<long long> > &odpoved,
                 vector< vector<fenwick_tree> > &FT,
                 long long nasob, int puvodni_delka,
                 int clo, int chi,
                 int fvlevo, int fvpravo)
{
    int c00 = puvodni_delka;

    for (int fnahoru=0; fnahoru<3; ++fnahoru)
        for (int fdolu=0; fdolu<3; ++fdolu) {
            if (fnahoru == 0 && fdolu == 0)
                continue;
            vector<int> pocty_barev(3,0);
            ++pocty_barev[fvlevo];
            ++pocty_barev[fvpravo];
            ++pocty_barev[fnahoru];
            ++pocty_barev[fdolu];
            int bile = pocty_barev[1], cerne = pocty_barev[2];
            int kolik = FT[fnahoru][fdolu].sum(clo,chi);
            odpoved[bile][cerne] += nasob * kolik;
            c00 -= kolik;
        }

    vector<int> pocty_barev(3,0);
```

```
    ++pocty_barev[fvlevo];
    ++pocty_barev[fvpravo];
    int bile = pocty_barev[1], cerne = pocty_barev[2];
    odpoved[bile][cerne] += nasob * c00;
}

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

    map<int, vector<vez> > veze_podle_radku;
    for (auto v : nactene_veze)
        veze_podle_radku[v.r].push_back(v);

    // Vytvoříme kompresi souřadnic pro sloupce
    set<int> ruzne_souradnice = {0, D-1};
    for (auto v : nactene_veze)
        for (int d=-1; d<=1; ++d)
            if (0 <= v.c+d && v.c+d < D)
                ruzne_souradnice.insert(v.c+d);
    vector<int> souradnice( ruzne_souradnice.begin(), ruzne_souradnice.end() );
    unordered_map<int,int> redukuj_sloupec;
    for (int i=0; i<int(souradnice.size()); ++i)
        redukuj_sloupec[souradnice[i]] = i+1;
    int Y = redukuj_sloupec.size();

    // Vyrobíme si pole pro odpovědi = počty políček jednotlivých typů
    vector< vector<long long> > odpoved( 5, vector<long long>(5,0) );
    if (N == 0) {
        odpoved[0][0] = 1LL*D*D;
        vypis(odpoved);
        return 0;
    }

    // Vytvoříme si stromy pro všechny kombinace barev
    vector< vector<fenwick_tree> > FT(3, vector<fenwick_tree>(3, fenwick_tree(Y+1)));

    // Roztřídíme si věže podle sloupců
    // V každém sloupci přidáme dolů průsvitnou zarážku a uspořádáme zdola nahoru
    vector< vector<vez> > veze_podle_sloupcu(Y+1);
    for (auto v : nactene_veze)
        veze_podle_sloupcu[ redukuj_sloupec[v.c] ].push_back(v);
    for (int y=1; y<=Y; ++y) {
        veze_podle_sloupcu[y].emplace_back(D,-47,0);
        reverse( veze_podle_sloupcu[y].begin(), veze_podle_sloupcu[y].end() );
    }

    // Zatím nastavíme, že v každém sloupci je barva 0 shora
    // a barva jeho nejvyšší věže zdola
```

```
    vector<int> barva_nahoru(Y+1,0);
    for (int y=1; y<=Y; ++y)
        FT[0][ veze_podle_sloupcu[y].back().barva ].update(y,+1);

    // Postupně procházíme všechny řádky
    int pred_radek = -1;
    for (auto &rec : veze_podle_radku) {
        int akt_radek = rec.first;
        vector<vez> &akt_veze = rec.second;

        // Přidame průsvitné zarážky na konec
        akt_veze.insert( akt_veze.begin(), {akt_radek,-1,0} );
        akt_veze.insert( akt_veze.end(),   {akt_radek,D,0} );

        // Zpracujeme prázdné řádky před aktuálním
        vytvor_blok(odpoved, FT, akt_radek-pred_radek-1, D,
                    redukuj_sloupec[0], redukuj_sloupec[D-1],
                    0, 0);
        pred_radek = akt_radek;

        // Zpracujeme bloky prázdných políček v tomto řádku
        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;
            vytvor_blok(odpoved, FT, 1, hi-lo+1,
                        redukuj_sloupec[lo], redukuj_sloupec[hi],
                        akt_veze[i].barva, akt_veze[i+1].barva);
        }

        // Upravíme data pro sloupce, které dostaly nové věže
        for (unsigned i=1; i+1<akt_veze.size(); ++i) {
            int c = redukuj_sloupec[ akt_veze[i].c ];
            int f1 = barva_nahoru[c];
            int f2 = akt_veze[i].barva;
            veze_podle_sloupcu[c].pop_back();
            int f3 = veze_podle_sloupcu[c].back().barva;
            FT[f1][f2].update(c,-1);
            FT[f2][f3].update(c,+1);
            barva_nahoru[c] = f2;
        }
    }

    if (pred_radek < D-1)
        vytvor_blok(odpoved, FT, D-pred_radek-1, D,
                    redukuj_sloupec[0], redukuj_sloupec[D-1],
                    0, 0);

    vypis(odpoved);
}
```

P-III-6 Cestou na trh

Hledané počty označíme postupně $a_1, \ldots, a_k$, na trh toho tedy šlo dohromady $n = a_1 + a_1a_2 + \cdots + a_1a_2 \ldots a_k$.

Hlavním pozorováním je uvědomit si, že všechny sčítance na pravé straně rovnosti jsou dělitelné číslem a_1. V každém platném řešení proto musí být a_1 dělitelem čísla n.

Všechny dělitele čísla n dokážeme nalézt v čase přímo úměrném $\sqrt{n}$ díky skutečnosti, že pokud d je dělitelem n, potom také n/d je dělitelem n, a alespoň jedno z čísel d a n/d musí být $\leq \sqrt{n}$. Stačí tedy vyzkoušet všechna d až po odmocninu z n a pokaždé, když najdeme nějaké, které dělí n, našli jsme jednu dvojici dělitelů.

Tímto způsobem tedy najdeme všechny možné hodnoty a_1. Když si některou z nich zvolíme, jak postupovat dále? Upravíme výše uvedený vztah následovně:

$$\begin{aligned} n &= a_1 + a_1a_2 + \cdots + a_1a_2 \ldots a_k \\ n &= a_1 \left(1 + a_2 + \cdots + a_2 \ldots a_k\right) \\ n/a_1 &= 1 + a_2 + a_2a_3 + \cdots + a_2 \ldots a_k \\ (n/a_1) - 1 &= a_2 + a_2a_3 + \cdots + a_2 \ldots a_k \end{aligned}$$

K tomu, abychom našli optimální hodnoty $a_2, \ldots, a_k$ pro zadané n a námi zvolené a_1, stačí tedy optimálně vyřešit původní úlohu pro číslo $n/a_1 - 1$.

Na základě popsaných myšlenek nyní napíšeme rekurzivní funkci, která bude výše popsaným postupem zkoušet všechny možnosti. Jediným dalším vylepšením bude přidání memoizace: Jakmile úlohu pro nějaké n vyřešíme, toto řešení si zapamatujeme. Jestliže budeme někdy v budoucnu znovu potřebovat řešení pro toto n, namísto opakovaného výpočtu jednoduše přímo vrátíme zapamatované řešení.

Toto řešení je dostatečně rychlé na zisk plného počtu bodů. Přesná analýza jeho časové složitosti je poměrně složitá, ale načrtneme alespoň, proč bude pro omezení ze zadání úlohy dostatečně rychlé. Čísla nad 10^6 nazveme velká, ostatní čísla budou malá. Při každém rekurzivním volání hodnotu n několikrát zmenšíme, proto bude velkých volání jen rozumně málo. I kdybychom malá volání provedli všechna, stále nám nepokazí časovou složitost. Navíc když nám některé n vytvoří mnoho různých velkých podproblémů (když má mnoho malých dělitelů), tyto se dále nebudou větvit stejným způsobem – jestliže totiž n bylo dělitelné nějakým p, které nedělí a_1, potom také n/a_1 jím dělitelné je, ale $n/a_1 - 1$ jím už dělitelné nebude.

Níže uvedená implementace ke každému úspěšně vyřešenému n ukládá rovnou celé řešení, tedy jednu optimální posloupnost a_i. Protože její délka je nejvýše logaritmická vzhledem k n, nezpůsobí to žádné výrazné zpomalení (i když by bylo o něco efektivnější ukládat si jen její optimální délku a na konci potom zrekonstruovat pouze optimální řešení pro původní n).

```
#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> sestroj_nejlepsi(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 = sestroj_nejlepsi(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 = sestroj_nejlepsi(n);
    cout << answer.size() << endl;
    for (unsigned i=0; i<answer.size(); ++i)
        cout << answer[i] << (i+1 == answer.size() ? '\n' : ' ');
}
```