P-III-4 Stavba dálnic

Dva body lze získat snadno – pro každý úsek si zapamatujeme, jestli je již rozestavěný. S novým dokumentem přepíšeme hodnoty u příslušných úseků a potom všechny úseky projdeme a spočítáme počet rozestavěných intervalů. Pro každý dokument tak projdeme všechny úseky, takže složitost bude $\mathcal{O}(nq)$.

Řešení za 6 bodů

Na více bodů je zapotřebí jít na úlohu jinak. Budeme si udržovat všechny aktuální intervaly rozestavěných úseků a pokaždé, když nám přijde dokument $[a_i, b_i]$, rozdělíme ho na dokumenty $[a_i, a_i], [a_i + 1, a_i + 1], \ldots [b_i, b_i]$ a ty postupně budeme přidávat a spojovat s aktuálními intervaly.

Procházet všechny intervaly a najít ty správné by trvalo $\mathcal{O}(n^2)$, ale když si je budeme udržovat seřazené v binárním vyhledávacím stromu, tak budeme schopni rychle hledat, kam interval přidat a s čím ho spojit.

Náš algoritmus bude vypadat následovně: Udržujeme si aktuální intervaly ve vyhledávacím stromu seřazené vzestupně. Pokaždé, když nám přijde dokument, rozdělíme jej na jednotkové intervaly. Každý z nich potom vložíme do vyhledávacího stromu, a pokud sousedí s vedlejšími intervaly, tak je spojíme. (Pokud by ležel uvnitř nějakého intervalu, tak ho tam vkládat ani nebudeme.) Poté, co vložíme všechny jednotkové dokumenty, vypíšeme počet intervalů ve vyhledávacím stromu.

Jakou má toto řešení složitost? Označíme-li jako m aktuální počet intervalů ve vyhledávacím stromu, pro každý jednotkový interval provedeme dotaz v čase $\mathcal{O}(\log m)$, podíváme se na sousední prvky v čase $\mathcal{O}(\log m)$ a případně je odstraníme a spojíme do jednoho v čase $\mathcal{O}(\log m)$. Všimněme si, že ve stromu bude nejvýše $\mathcal{O}(q)$ intervalů, protože jednotkové intervaly z jednoho dokumentu se spojí. Celkem tedy složitost bude $\mathcal{O}(q\ell \log q)$, kde ℓ je délka nejdelšího intervalu.

Zrychlujeme

Pomalá část na našem řešení je to, že rozdělujeme každý interval na jednotkové. Co kdybychom vkládali intervaly do stromu celé? Budeme udržovat invariant, že ve stromu jsou po každém kroku uložené intervaly seřazené podle jejich počátků, které se navzájem nepřekrývají (tj. mezi pravým koncem každého intervalu a levým koncem následujícího intervalu je alespoň jeden nerozestavěný úsek dálnice).

Pokud nyní přijde dokument, že se rozestavěl nový interval, potřebujeme strom updatovat: Nový interval vložíme, a dokud bude s intervalem nalevo od něj mít přesah nebo budou končit hned vedle sebe, spojíme je a opakujeme tuto kontrolu. Jakmile už je nebudeme moct spojit, tak budeme stejně opakovat spojování pro intervaly vpravo.

Může se zdát, že při opakovaném spojování uděláme mnoho operací, ale když opakovaně spojujeme, tak spojené intervaly mažeme. A protože každý interval jednou přidáme a nejvýše jednou odebereme, výsledná složitost bude $\mathcal{O}(q \log q)$.

```
#include <iostream>
#include <vector>
#include <set>
#include <utility>

using namespace std;

int main() {
    int n, m;
    cin >> n >> m;
    set<pair<int, int>> intervals;
    intervals.insert({-2, -2});   // hack na krajní prvky
    intervals.insert({n+2, n+2}); // ve vyhledávacím stromu
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        vector<pair<int, int>> to_delete;

        bool skip = false;
        if (intervals.size()) {
            auto p=intervals.lower_bound({a, b});
            auto q=intervals.lower_bound({a, b}); q--;
            while (true) {
                if ((p->first <= a && b <= p->second)
                    || (q->first <= a && b <= q->second)) {
                    skip = true;
                    break;
                } else if (p != intervals.end() && b >= p->first - 1) {
                    b = max(b, p->second);
                    to_delete.push_back({p->first, p->second});
                    p++;
                } else if (a <= q->second + 1) {
                    a = min(a, q->first);
                    to_delete.push_back({q->first, q->second});
                    if (q == intervals.begin()) break;
                    q--;
                } else {
                    break;
                }
            }
            for (auto x: to_delete)
                intervals.erase(x);
        }

        if (!skip)
            intervals.insert({a, b});
        cout << intervals.size() - 2 << endl;
    }
}
```

P-III-5 Malý princ

Máme zadaný cyklus délky n (v zadání úlohy je slíbeno, že $n = 10^5$) a v každém vrcholu je napsané celé číslo t_i takové, že $t_{i+1} = t_i + 1$ nebo $t_{i+1} = t_i - 1$, přičemž indexujeme cyklicky, takže $t_{n+i} = t_i$. Chceme najít i takové, že $t_i = t_{i+n/2}$, anebo rozhodnout, že takové i neexistuje.

Definujme $d_i = t_i - t_{i+n/2}$. Všimněte si, že $d_i = 0$ právě tehdy, když $t_i = t_{i+n/2}$. Takže úloha ekvivalentně chce, abychom nalezli vrchol i takový, že $d_i = 0$. Jednoduchým rozborem případů můžeme dokázat, že pro každé i platí $d_{i+1} \in \{d_i - 2, d_i, d_i + 2\}$. Z toho speciálně plyne, že všechna d_i mají stejnou paritu (zbytek po dělení dvěma). Takže pokud zjistíme, že nějaké d_i je liché, můžeme odpovědět, že žádné dva protější vrcholy nemají stejnou teplotu. Můžete si rozmyslet, že tohle nastane právě tehdy, když $n/2$ je liché číslo, takže v našem případě bude vždy řešení existovat. Dále budeme předpokládat, že všechna d_i jsou sudá.

Zřejmě platí, že $d_i = t_i - t_{i+n/2} = -(t_{i+n/2} - t_i) = -(t_{i+n/2} - t_{i+n/2+n/2}) = -d_{i+n/2}$, tedy protější vrcholy mají opačné hodnoty d_i. Poslední pozorování je, že d_i splňuje variantu *diskrétní spojitosti*: Pro $i < j$ a pro každé x takové, že $\min(d_i, d_j) \leq x \leq \max(d_i, d_j)$ a x má stejnou paritu jako d_i, existuje k takové, že $i \leq k \leq j$ a $d_k = x$. Lidsky řečeno, funkce d na libovolném intervalu nabývá všech hodnot, které leží mezi hodnotami v krajních bodech intervalu a mají stejnou paritu.

Takže jsou dvě možnosti: Buď $d_0 = 0$ a potom jsme našli vyhovující vrchol, anebo $d_0 \neq 0$. Potom $d_{n/2} = -d_0$, a tudíž někde na cestě mezi vrcholem 0 a vrcholem $n/2$ musí ležet vrchol k takový, že $d_k = 0$. Pomocí binárního vyhledávání* umíme potom za $\lceil \log_2(n/2) \rceil = \lceil \log_2 n \rceil - 1$ kroků nalézt k takové, že $d_k = 0$. V každém kroku potřebujeme zjistit d_m pro m ležící zhruba v polovině aktuálního intervalu, a na zjištění d_m se potřebujeme Malého prince zeptat na t_m a $t_{m+n/2}$, což jsou dva dotazy. Další dva dotazy jsme potřebovali na zjištění hodnoty d_0, tedy celkově jsme potřebovali $2\lceil \log_2 n \rceil$ dotazů. To stačilo na zisk devíti bodů, jelikož $2\lceil \log_2 n \rceil = 34$.

Poslední bod se dal získat za pozorování, že binární vyhledávání můžeme ve skutečnosti ukončit o krok dříve: Pokud víme, že pro nějaké i máme $b(i) < 0$ a $b(i + 2) > 0$ (nebo opačně), pak musí platit, že $b(i + 1) = 0$, a tudíž teplotu ve vrcholech $i + 1$ a $i + 1 + n/2$ není nutné měřit.

```
#include <bits/stdc++.h>

using namespace std;

int n = 100000;

int get_b(int i) {
        assert(i < n/2);
        int t1, t2;
        cout << "0 " << i << endl << flush;
        cin >> t1;
        cout << "0 " << i + n/2 << endl << flush;
```

* Binární vyhledávání je vysvětlené například v kuchařce KSP Binární vyhledávání na *https://ksp.mff.cuni.cz/kucharky/binarni-vyhledavani/*.

```
        cin >> t2;
        return t1 - t2;
}

bool same_sgn(int x, int y) {
        return (x < 0 && y < 0) || (x > 0 && y > 0);
}

int main() {
        int l = 0, r = n/2;
        int b0 = get_b(l);
        if (b0 == 0) {
                cout << "1 " << 0 << endl << flush;
                return 0;
        }

        while (r - l > 2) {
                int m = (r+l) / 2;
                int b = get_b(m);

                if (b == 0) {
                        cout << "1 " << m << endl << flush;
                        return 0;
                }

                if (same_sgn(b, b0)) {
                        l = m;
                        b0 = b;
                } else {
                        r = m;
                }
        }

        // Binární vyhledávání končíme o krok dříve.
        // Pokud jsme doteď nenašli vhodné řešení, musí platit b(l+1) = 0.
        cout << "1 " << l+1 << endl;
}
```

P-III-6 Bonbónovník

V první podúloze stačí pro každý vrchol vyzkoušet všechny možnosti, jak vybrat bonbóny na odnesení, a z nich vybrat tu nejlepší. Jen je potřeba ověřit, zda vybrané vrcholy tvoří (souvislý) podstrom. Zkoušení všech možností můžeme elegantně naprogramovat pomocí bitmasek – budeme posloupně procházet čísla od 0 do $2^n - 1$ a i-tý vrchol bude pro nás vybraný, právě když bude 1 na i-té pozici binárního zápisu daného čísla.

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

const int NMAX = 11234567;

struct Nd
{
        vector<int> e;
        int val;
} nd[NMAX];
```

```
int unmark_component(int mask, int i)
{
        if ((mask>>i) & 1) {
                mask &= ~(1<<i);
                for (int it : nd[i].e)
                        mask = unmark_component(mask, it);
        }
        return mask;
}

int main(int argc, char ** argv)
{
        int n;
        scanf("%d", &n);
        for (int i=0; i<n; i++) {
                char c;
                scanf(" %c", &c);
                nd[i].val = c=='L' ? 1 : -1;
        }
        for (int i=0; i<n-1; i++) {
                int x,y;
                scanf("%d%d", &x, &y);
                nd[x].e.push_back(y);
                nd[y].e.push_back(x);
        }
        for (int i=0; i<n; i++) {
                int opt = -n;   // Horší to být nemůže
                for (int mask=0; mask<(1<<n); mask++) if ((mask>>i) & 1) {
                        if (unmark_component(mask, i) == 0) {
                                int val = 0;
                                for (int j=0; j<n; j++) if ((mask>>j) & 1)
                                        val += nd[j].val;
                                opt = max(opt, val);
                        }
                }
                printf("%d\n", opt);
        }
        return 0;
}
```

Předchozí řešení je opravdu pomalé. Pojďme ho zrychlit. Představme si, že strom je zakořeněný ve vrcholu, kde je cenovka. Pro každý podstrom si můžeme spočítat, jaký nejlepší rozdíl z něj zvládneme dostat za předpokladu, že vybereme jeho kořen. Podstromy můžeme reprezentovat vrcholem, který je jeho kořenem.

U listů je to jednoduché. List je vybraný a žádný další vrchol v podstromu není, takže optimální rozdíl bude prostě 1 v případě, že je v daném vrcholu lékorka, a -1, když je tam bonpari. U ostatních vrcholů stejným způsobem spočteme hodnotu kořene. Pak ještě ovšem musíme uvážit ostatní vrcholy. Zde využijeme toho, že si nejprve spočítáme optimální rozdíly pro všechny vrcholy podstromu. Když vybereme nějakého potomka, tak se nám očividně vyplatí z jeho podstromu získat nejlepší možný rozdíl. Tuto hodnotu již ale máme spočítanou. Zbývá tedy rozhodnout, které potomky vybrat a které ne. Jelikož podstromy potomků nejsou nijak propojené, výběr jednoho potomka nijak neovlivní výběr ostatních. U každého potomka tedy

stačí uvážit, jestli se nám vyplatí nebo ne. Ovšem vyplatí se právě tehdy, když jeho optimální rozdíl je kladný.

Počítání optimálních rozdílů je vlastně prohledávání do hloubky na stromu s tím, že optimální rozdíl vrcholu spočteme až poté, co se vrátíme z podstromů. Toto můžeme snadno implementovat pomocí rekurzivní funkce.

Tento algoritmus můžeme provést pro každý vrchol grafu. Výsledná časová složitost tedy bude $\mathcal{O}(n^2)$.

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

const int NMAX = 11234567;

struct Nd
{
        vector<Nd *> e;
        int val;
        int calc(Nd *from)
        {
                int opt = val;
                for (Nd * it : e) if (it!=from) {
                        opt += max(0, it->calc(this));
                }
                return opt;
        }
} nd[NMAX];

int main(int argc, char ** argv)
{
        int n;
        scanf("%d", &n);
        for (int i=0; i<n; i++) {
                char c;
                scanf(" %c", &c);
                nd[i].val = c=='L' ? 1 : -1;
        }
        for (int i=0; i<n-1; i++) {
                int x, y;
                scanf("%d%d", &x, &y);
                nd[x].e.push_back(nd+y);
                nd[y].e.push_back(nd+x);
        }
        for (int i=0; i<n; i++)
                printf("%d\n", nd[i].calc(NULL));
        return 0;
}
```

Předchozí algoritmus nám kromě optimálního rozdílu pro vrchol s cenovkou téměř spočte i optimální rozdíly ostatních vrcholů v jejich podstromech až na to, že nemůžeme vybrat žádnou část stromu směrem „nahoru“. Pojďme tedy pro každý vrchol dopočítat, kolik můžeme získat tím, že dovolíme vybrat i část stromu nad ním.

Aktuální vrchol označme jako X a vrchol nad ním (pokud existuje) jako Y. Pro kořen se nic nemění, nad ním už nic není. Pro ostatní vrcholy se můžeme rozhodnout,

jestli vybereme vrchol nad ním. Pokud ano, pak také můžeme jednak vybírat i část od Y dále směrem ke kořeni (pokud Y již není kořen) a také můžeme pokračovat do ostatních potomků vrcholu Y (různých od X). Optimální rozdíly podstromů ostatních potomků již ale máme spočítané. Pokud si tedy nejprve spočteme optimální část směrem ke kořeni, tak již můžeme vše snadno spočítat. Podle toho, jaký bonbón je ve vrcholu nad Y, začneme s 1 nebo -1. K tomu přičteme optimální rozdíl od něj směrem nahoru a všechny kladné optimální rozdíly podstromů ostatních potomků. Pokud vyjde záporné číslo, tak řekneme, že optimum směrem nahoru je 0, tedy vyplatí se nevzít v daném směru nic.

Výše popsaný algoritmus je zase prohledávání do hloubky, jen teď již musíme počítat optimální rozdíl směrem ke kořeni před tím, než půjdeme zpracovávat potomky.

Popsaný algoritmus pro každý vrchol prochází všechny potomky vrcholu nad ním. Jeho složitost tedy můžeme odhadnout jako $\mathcal{O}(nd)$, kde d je maximální stupeň vrcholu. V případě, že graf je jen kořen, pod kterým jsou přímo pověšené ostatní vrcholy, tak skutečně bude složitost algoritmu $\Theta(nd) = \Theta(n^2)$.

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

const int NMAX = 11234567;

struct Nd
{
        vector<Nd *> e;
        int val;
        int opt, opt_up;

        int calc_up(Nd *from)
        {
                opt = val;
                for (Nd * it : e) if (it != from) {
                        opt += max(0, it->calc_up(this));
                }
                opt_up = opt;
                return opt;
        }

        void calc_down(Nd * from, int from_up)
        {
                for (Nd * it : e) if (it != from) {
                        int from_me = val + max(0, from_up);
                        for (Nd * jt : e) if (jt != from) if (jt != it) {
                                from_me += max(0, jt->opt_up);
                        }
                        it->calc_down(this, from_me);
                }
                opt += max(0, from_up);
        }
} nd[NMAX];

int main()
{
        int n;
```

```
        scanf("%d", &n);
        for (int i=0; i<n; i++) {
                char c;
                scanf(" %c", &c);
                nd[i].val = c=='L' ? 1 : -1;
        }
        for (int i=0; i<n-1; i++) {
                int x, y;
                scanf("%d%d", &x, &y);
                nd[x].e.push_back(nd+y);
                nd[y].e.push_back(nd+x);
        }
        nd->calc_up(NULL);
        nd->calc_down(NULL, 0);
        for (int i=0; i<n; i++)
                printf("%d\n", nd[i].opt);
        return 0;
}
```

Od předešlého řešení je už jen krůček k optimálnímu řešení. Všimneme si, že když k optimálnímu rozdílu směrem nahoru z X do Y (před omezení na kladné hodnoty) přičteme optimum od X dolů (pokud je kladně), získáme optimální rozdíl pro celý strom s cenovkou v Y. Toto můžeme využít v opačném směru – vhodným odečtením od optima pro Y snadno spočteme optimum od X nahoru a tím dopočteme optimum pro X.

Takovýto algoritmus již bude mít časovou složitost $\mathcal{O}(n)$, protože pro každý vrchol již bude stačit udělat konstantně mnoho operací.

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

const int NMAX = 11234567;

struct Nd
{
        vector<Nd *> e;
        int val;
        int opt, opt_up;

        int calc_up(Nd *from)
        {
                opt = val;
                for (Nd *it : e) if (it != from) {
                        opt += max(0, it->calc_up(this));
                }
                opt_up = opt;
                return opt;
        }

        void calc_down(Nd *from, int from_up)
        {
                opt += max(0, from_up);
                for (Nd *it : e) if (it != from) {
                        it->calc_down(this, opt - max(0, it->opt_up));
                }
```

```
        }
} nd[NMAX];

int main()
{
        int n;
        scanf("%d", &n);
        for (int i=0; i<n; i++) {
                char c;
                scanf(" %c", &c);
                nd[i].val = c=='L' ? 1 : -1;
        }
        for (int i=0; i<n-1; i++) {
                int x, y;
                scanf("%d%d", &x, &y);
                nd[x].e.push_back(nd+y);
                nd[y].e.push_back(nd+x);
        }
        nd->calc_up(NULL);
        nd->calc_down(NULL, 0);
        for (int i=0; i<n; i++)
                printf("%d\n", nd[i].opt);
        return 0;
}
```