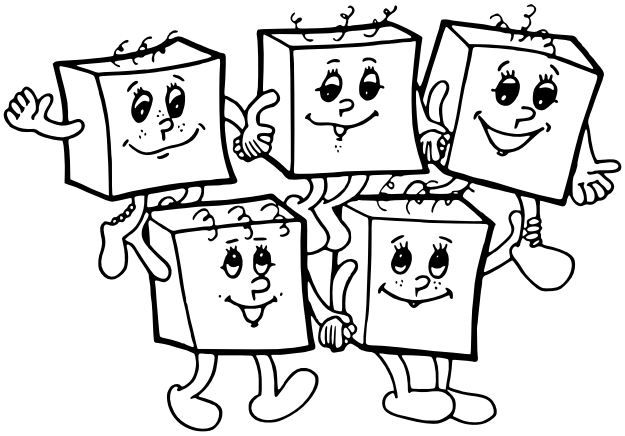
OLYMPIÁDA V INFORMATIKE

NA STREDNÝCH ŠKOLÁCH

http://oi.sk/



dvadsiaty šiesty ročník
školský rok 2010/11

riešenia krajského kola
kategória A

A-II-1 Vlak

Priamočiarym riešením je napríklad vyskúšať všetky možnosti, ako sa dajú odobrať vagóny, a pre každú z nich vyskúšať, či vznikne palindróm. Takýchto možností je toľko, koľko je podmnožín n-prvkovej množiny. A tých je až 2^n, čo je už pre $n = 50$ neúnosne veľa.

Skúsme teda úlohu vyriešiť po krokoch. Reprezentujme si vlak ako postupnosť písmen $a_1a_2a_3 \ldots a_{n-1}a_n$. Pre podpostupnosť $a_ia_{i+1} \ldots a_{j-1}a_j$ označme $V[i, j]$ najmenší počet vagónov, ktoré z nej treba vyradiť, aby bola symetrická. Špeciálne teda riešenie pre celý pôvodný vlak (to, ktoré nás zaujíma) bude označené $V[1, n]$.

Čo vieme o týchto hodnotách povedať? Ak $i = j$, tak zjavne $V[i, j] = 0$, lebo vlak s jedným vagónom je vždy symetrický. Nech teda $i < j$. Ako vyzerá optimálne riešenie pre vlak $a_ia_{i+1} \ldots a_{j-1}a_j$?

V prípade, že $a_i = a_j$, prvý ani posledný vagón evidentne v optimálnom riešení vyradiť netreba.[1] V takomto prípade nám ostáva vyriešiť našu úlohu pre vagóny $a_{i+1} \ldots a_{j-1}$. A teda platí $V[i, j] = V[i + 1, j - 1]$.

V prípade, že $a_i \neq a_j$, musíme aspoň jeden z týchto dvoch vagónov vyhodiť (inak by hneď za lokomotívou bol po otočení iný typ vagónu). Ak by sme vyhodili vagón a_i, ostane nám vlak tvorený vagónmi $a_{i+1} \ldots a_{j-1}a_j$. Pre tento vlak potrebujeme vyhodiť ďalších $V[i + 1, j]$ vagónov. A naopak, ak by sme vyhodili vagón a_j, ostane nám $a_ia_{i+1} \ldots a_{j-1}$ a pre ten treba ešte $V[i, j - 1]$ zmien. Z týchto dvoch možností, ktoré máme na výber, si samozrejme vyberieme tú lepšiu pre nás. Preto v tomto prípade platí $V[i, j] = 1 + \min(V[i + 1, j], V[i, j - 1])$.

V oboch prípadoch sme teda našli vzťah, ktorý optimálne riešenie $V[i, j]$ spočíta v konštantnom čase z optimálnych riešení pre iné, kratšie vlaky. A toto nám už stačí na napísanie algoritmu s časovou zložitosťou $\Theta(n^2)$: Budeme postupne spracúvať všetky podreťazce zadaného vlaku, začínajúc od najkratších a končiac celým vlakom. A pre každý úsek si spočítame a zapamätáme optimálny počet vyhodení vagónu.

Pamäťová zložitosť takéhoto riešenia je tiež $\Theta(n^2)$. Musíme si totiž zapamätať všetky hodnoty $V[i, j]$, aby sme na konci vedeli jedno optimálne riešenie zostrojiť. (Existuje aj riešenie s časovou zložitosťou $\Theta(n^2)$ a pamäťovou $\Theta(n)$, ale je zbytočne komplikované, preto ho nebudeme uvádzať.)

[1] Dôkaz: Od najlepšieho riešenia, v ktorom oba vyradíme, je lepšie to isté riešenie, v ktorom ale oba ponecháme. Ak by sme v optimálnom riešení vyradili len jeden z nich, BUNV nech to je a_j, dostaneme rovnako dobré riešenie, ak namiesto a_j vyradíme ten vagón, ktorý zostal posledný nevyradený.

Listing programu:

```
#include <cstdio>
#include <algorithm>

#define MAX 10010

int N;
char vstup[MAX];
unsigned short V[MAX][MAX]; // hodnoty V[i,j] z popisu riesenia, zmestia sa do unsigned short

int main() {
    scanf("%d %s", &N, vstup);

    // podvlaky dlzky 0 a 1 su optimalne
    for (int i = 0; i < N; i++) V[i][i-1] = V[i][i] = 0;

    // pre dlzky 2..N si spocitame optimalnu dlzku riesenia
    for (int dlzka = 2; dlzka <= N; dlzka++) {
        for (int zaciatok = 0; zaciatok+dlzka <= N; zaciatok++) {
            int koniec = zaciatok + dlzka - 1;
            if (vstup[zaciatok] == vstup[koniec]) {
                V[zaciatok][koniec] = V[zaciatok+1][koniec-1];
            } else {
                V[zaciatok][koniec] = 1 + std::min( V[zaciatok+1][koniec], V[zaciatok][koniec-1] );
            }
        }
    }

    // teraz ideme zrekonstruovat jedno optimalne riesenie
    bool vyhodit[MAX];
    for (int i = 0; i < N; i++) vyhodit[i] = false;
    int zaciatok = 0, koniec = N-1;
    while (koniec > zaciatok) {
        if (vstup[zaciatok] == vstup[koniec]) {
            // prvy aj posledny vagon ostavaju
            zaciatok++;
            koniec--;
        } else {
            // odstranime bud zaciatok alebo koniec, podla toho co je vyhodnejsie
            if (V[zaciatok][koniec] == 1+V[zaciatok+1][koniec]) {
                vyhodit[zaciatok] = true;
                zaciatok++;
            } else {
                vyhodit[koniec] = true;
                koniec--;
            }
        }
    }

    // a ideme vypisat najdene riesenie
    printf("%d\n", V[0][N-1]);
    bool medzera = false;
    for (int i = 0; i < N; i++) {
        if (vyhodit[i]) {
            if (medzera) printf(" ");
            medzera = true;
            printf("%d", i + 1); // v programe indexujeme od 0, v zadani od 1
        }
    }
    printf("\n");
}
```

Dodatok: riešenie s lepšou pamäťovou zložitosťou

Existuje aj riešenie úlohy Vlak v čase $O(n^2)$ a pamäti $O(n)$. V nasledujúcom texte si ho ukážeme.

V čom je problém?

Pripomeňme si, že $V[i,j]$ je najmenší počet vagónov, ktoré treba vyradiť z $a_i a_{i+1} \dots a_{j-1} a_j$, aby sme dostali palindróm. Platí, že ak $a_i = a_j$, tak $V[i,j] = V[i+1, j-1]$, inak $V[i,j] = 1 + \min(V[i+1,j], V[i,j-1])$. Všimnite si, že v prvom prípade sa odkazujeme na postupnosť o 2 znaky kratšiu, v druhom prípade na dve postupnosti, ktoré sú obe od pôvodnej o 1 znak kratšie.

Keď budeme teda hodnoty $V[i,j]$ počítať najskôr pre všetky postupnosti dĺžky 1, potom všetky postupnosti dĺžky 2, dĺžky 3, atď., tak nám v každom okamihu stačí pamäť $O(n)$: keď spracúvame postupnosti dĺžky k, stačí

si pamätať optimálne riešenia pre postupnosti dĺžok $k-1$ a $k-2$.

Optimálny počet vagónov, ktoré treba odstrániť, vieme teda v lineárnej pamäti spočítať ľahko. Problém je v samotnom zostrojení riešenia.

Zložitejšia úloha

Namiesto toho, aby sme ukázali, ako tento problém riešiť, si našu úlohu ešte trochu zovšeobecníme: dovolíme, aby náš vlak obsahoval **nanajvýš jeden** špeciálny vozeň „`*`“, ktorý nesmieme odobrať. Táto úloha sa zjavne rieši rovnako ako pôvodná – len v prípade, že máme špeciálny vozeň, neskúšame možnosti, ktoré ho odoberú.

Rozoberáme vlak

Predstavme si, že postupne rozoberáme vlak, ktorý sme dostali na vstupe, a to spôsobom, ktorý zodpovedá (jednému možnému) optimálnemu riešeniu. Pre vlak `ABCXDFABECEDAFDCBYZZA` by tento proces vyzeral takto:

– zober do riešenia `A` zo začiatku aj z konca
– zahoď `Z` z konca
– zahoď `Z` z konca
– zahoď `Y` z konca
– zober do riešenia `B` zo začiatku aj z konca
– ...

Po pár takýchto krokoch sa z reťazca dĺžky n dostaneme k reťazcu polovičnej dĺžky, presnejšie $\lfloor n/2 \rfloor$ alebo $\lfloor n/2 \rfloor + 1$. V našom príklade by to mohol byť reťazec `DFABECEDAFD`.

No a presne tento reťazec (presnejšie: indexy, kde v pôvodnom reťazci začína a končí) vieme nájsť počas toho, ako počítame hodnoty $V[i,j]$: Kým spracúvame úseky dĺžky menšej ako $\lfloor n/2 \rfloor$, len počítame hodnoty $V[i,j]$ ako v predchádzajúcom riešení, nerobíme nič navyše. Akonáhle začneme spracúvať dlhšie úseky, budeme si pre každý z nich pamätať nie len optimálnu hodnotu $V[i,j]$, ale aj to, ktorý reťazec dĺžky $\lfloor n/2 \rfloor$ alebo $\lfloor n/2 \rfloor + 1$ by sme dostali počas optimálneho riešenia úseku $a_i a_{i+1} \ldots a_{j-1} a_j$.

V čase $O(n^2)$ a pamäti $O(n)$ teda vieme okrem hodnoty $V[1,n]$ získať trochu informácie navyše: vieme v pôvodnom reťazci nájsť „strednú časť“ s vyššie uvedeným významom.

A pasca sklapne

A teraz sa ukáže, načo bola dobrá zložitejšia úloha. Nech sme pre pôvodný reťazec $w = \alpha\beta\gamma$ zistili, že počas výroby optimálneho riešenia sa „na pol ceste“ stretneme s reťazcom β. To ale znamená, že úlohu „optimálne zostroj palindróm z w“ vieme rozbiť na dve menšie, nezávislé podúlohy: „optimálne zostroj palindróm z β“ a „optimálne zostroj palindróm z $\alpha * \gamma$“. Obe tieto úlohy vieme vyriešiť rekurzívnym volaním pôvodného algoritmu.

Napríklad uvažujme vlak z vyššie uvedeného príkladu. Najdlhší palindróm, ktorý sa v ňom nachádza, vieme poskladať z najdlhších palindrómov, ktoré sa nachádzajú v reťazcoch `DFABECEDAFD` a `ABCX*CBYZZA`.

Analýza časovej zložitosti

Označme $T(n)$ čas výpočtu tohto algoritmu na reťazci dĺžky n. Z popisu algoritmu dostávame, že $T(n) = \Theta(n^2) + 2T(n/2)$. Dá sa dokázať, že potom $T(n) = \Theta(n^2)$, teda asymptotická časová zložitosť je rovnaká ako pri pôvodnom riešení. Inými slovami, prácu navyše, ktorú toto nové riešenie spraví, môžeme zanedbať pri porovnaní s pôvodným dynamickým programovaním. Intuitívne zdôvodnenie:

– Pre pôvodný reťazec spravíme rádovo n^2 krokov výpočtu.
– Pre každý z dvoch polovičných reťazcov spravíme rádovo $(n/2)^2 = n^2/4$ krokov výpočtu, teda dokopy rádovo $n^2/2$.
– Z každého z nich vzniknú dva reťazce štvrtinovej dĺžky. Pre všetky štyri takéto reťazce dokopy spravíme rádovo $4 \cdot (n/4)^2 = n^2/4$ práce.
– A tak ďalej. Celkový počet krokov je teda rádovo $n^2 + n^2/2 + n^2/4 + n^2/8 + \cdots = 2n^2$.

Formálne sa dá napríklad matematickou indukciou dokázať, že z $T(x) = 1$ pre $x \leq 1$ a $T(n) \leq cn^2 + 2T(n/2)$ vyplýva $T(n) \leq 2cn^2$.

Poznámka na záver

Podobný trik sa dá aplikovať aj v iných úlohách. Vyskúšajte si napríklad v čase $O(n^2)$ a pamäti $O(n)$ zostrojiť jednu najdlhšiu postupnosť, ktorá sa dá vybrať aj z postupnosti $a_1, \ldots, a_n$, aj z postupnosti $b_1, \ldots, b_n$.

Dodatok: nesprávne riešenie a ako ho opraviť

Veľmi často sa pri tejto úlohe vyskytuje nasledujúce **nesprávne** riešenie: najdlhší palindróm v postupnosti $a_1 a_2 \ldots a_n$ nájdem ako najdlhšiu spoločnú podpostupnosť postupností $A = a_1 a_2 \ldots a_n$ a $B = a_n \ldots a_2 a_1$.

Tento prístup nie je uplny zlý (podobá sa predsa na naše vzorové riešenie), ale má niekoľko problémov. **Platí** síce, že najdlhší palindróm v postupnosti A má rovnakú dĺžku ako najdlhšia spoločná podpostupnosť A a B. Ale ani omylom **neplatí**, že každá ich spoločná podpostupnosť je aj palindrómom. Napríklad pre vstup `ACBBAC` je hľadaný palindróm buď `ABBA` alebo `CBBC`, ale nájdená podpostupnosť môže byť aj `ABBC` alebo `CBBA`.

Pozrime sa najskôr na samotný algoritmus na hľadanie najdlhšej spoločnej podpostupnosti postupností $a_1 a_2 \ldots a_n$ a $b_1 b_2 \ldots b_n$. Podobne ako vo vzorovom riešení použijeme dynamické programovanie: $D[i, j]$ bude dĺžka najdlhšej spoločnej podpostupnosti pre postupnosti $a_1 \ldots a_i$ a $b_1 \ldots b_j$. Pre tieto hodnoty platí okrajová podmienka $D[0, j] = D[i, 0] = 0$ a všeobecný vzťah nasledovný: ak $a_i = b_j$, tak $D[i, j] = 1 + D[i-1, j-1]$, inak $D[i, j] = \max(D[i-1, j], D[i, j-1])$.

Predstavme si teda, že sme tieto hodnoty $D[i, j]$ spočítali. My ale v skutočnosti nehľadáme najdlhšiu spoločnú podpostupnosť. Predstavme si, že hľadáme palindróm párnej dĺžky. Potom určite existuje nejaké k také, že z $a_1 \ldots a_k$ vyberieme jeho prvú polovicu a z $a_{k+1} \ldots a_n$ druhú. Inými slovami, polovica hľadaného palindrómu je spoločnou podpostupnosťou postupností $a_1 \ldots a_k$ a $a_n \ldots a_{k+1}$. A jej dĺžku predsa poznáme: je to $D[k, n-k]$. Stačí teda vyskúšať všetky možné k a vybrať si najlepšiu možnosť. Palindrómy nepárnej dĺžky vyriešime analogicky – zoberieme doň a_k a zvyšok nájdeme ako najdlhšiu spoločnú podpostupnosť $a_1 \ldots a_{k-1}$ a $a_n \ldots a_{k+1}$.

Iné, trikové riešenie: Predstavme si, že sme našli najdlhšiu spoločnú podpostupnosť P zadaného reťazca a jeho reverzu. Táto síce nemusí byť palindrómom, vieme z nej však vždy jeden vyhovujúci palindróm rovnakej dĺžky zostrojiť. Určite totiž zafunguje aspoň jeden z postupov „zober prvú polovicu P a jej reverz“ a „zober reverz druhej polovice P a druhú polovicu P“. Rozmyslite si, prečo je to tak.

A-II-2 Jabloňový sad

Hovoríme, že množina bodov v rovine je *konvexná*, ak pre každé dva jej body A a B platí, že obsahuje celú úsečku AB. Preložené do ľudskej reči, konvexný útvar drží pokope a nemá na obvode žiadne preliačenia dovnútra. Pre ľubovoľnú množinu M je jednoznačne definovaný jej *konvexný obal*: najmenšia konvexná množina $\overline{M}$, ktorá obsahuje celú množinu M.

Ak je M tvorená len konečne veľa bodmi, intuitívne si jej konvexný obal môžeme predstaviť nasledovne: body si predstavíme ako klince na doske, okolo všetkých natiahneme gumičku a pustíme ju. Gumička sa bude skracovať, až kým nenarazí na niektoré z klincov, a medzi nimi zostane napnutá. (Pozrite si ešte raz obrázky v zadaní, ak si to neviete predstaviť.)

Asi nikoho teraz neprekvapí, že hľadaný plot je vždy práve obvodom konvexného obalu aktuálnej množiny bodov. Zopakujme si teda niektoré tvrdenia, ktoré pre konvexný obal množiny bodov zjavne platia:

- Konvexný obal je vždy mnohouholník, jeho vrcholy sú niektoré z daných bodov.
- Body najviac vľavo a najviac vpravo (najmenšia a najväčšia súradnica x) ležia na jeho obvode.
- Pre každý vrchol konvexného obalu platí, že vnútorný uhol pri ňom je menší ako 180°.

Nájsť konvexný obal pre zadané množiny bodov v rovine je pomerne známy problém, pre ktorý existujú algoritmy pracujúce v čase $O(n \log n)$. Jeden takýto algoritmus popíšeme.

Konvexný obal môžeme rozdeliť na horný a dolný polobal. Začiatkom oboch polobalov bude najľavejší bod (ak je takých viac, najspodnejší z nich), koncom je najpravejší bod (ak je takých viac, najvrchnejší z nich). V našom algoritme samostatne nájdeme horný a samostatne dolný konvexný polobal.

Na nájdenie horného polobalu si najskôr usporiadame body podľa súradníc (najskôr podľa x a v prípade rovnosti podľa y). Potom si body postupne pridávame do polobalu. Vždy, keď pridáme bod do polobalu, skontrolujeme, či nám tým pri predchádzajúcom bode v polobale nevznikol vnútorný uhol väčší alebo rovný 180°. Ak áno, tak ten predchádzajúci bod práve prestal byť na hornom polobale, preto ho vyhodíme. Toto budeme opakovať, kým v obale neostanú len dva body, alebo kým pri predchádzajúcom bode v polobale nedostaneme vnútorný uhol menší ako 180°. Dolný polobal hľadáme analogicky, len prechádzame body v opačnom poradí.

Ak by stačilo nájsť konvexný obal, tu by sme mohli prestať. Naša úloha však nekončí. Teraz, keď máme hotový konvexný obal, potrebujeme doň vedieť rýchlo pridať ešte jeden bod.

Pozrime sa, ako sa zmení horný polobal. Ak pridaný bod leží pod ním alebo na ňom, polobal zostane nezmenený. Predpokladajme teda, že nový bod $A = [x_A, y_A]$ leží nad horným polobalom. Body na polobale vždy idú za sebou podľa x-ových súradníc (v prípade rovnosti podľa y-ových). Je teda jasne určené miesto, na ktoré bod A patrí. Pridajme ho teda naň. Teraz mohli nastať dva prípady: buď naďalej všetky vrcholy zvierajú vnútorný uhol menší ako 180° a všetko je v poriadku, alebo sa táto vlastnosť na niektorých miestach porušila. Ale jediné (nanajvýš) dva body, u ktorých sa to mohlo stať, sú práve susedia čerstvo pridaného bodu A. Tieto body sa teda ocitli vo vnútri nového konvexného obalu, z nášho polobalu ich teda vyhodíme. Toto budeme opakovať až kým také body v polobale nebudú existovať (pripomíname, že sa stačí stále pozerať na aktuálne susedné body bodu A). Keď už pravý aj ľavý sused bodu A majú správne vnútorné uhly, tak máme správny nový horný polobal. Dolný polobal upravíme analogicky.

Pozrime sa teraz ako tento algoritmus implementovať. Potrebujeme vedieť rýchlo nájsť ľavého a pravého suseda v postupnosti a potrebujeme vedieť rýchlo pridávať a odoberať nové body. Pole alebo zoznam nie sú vhodné: v poli vkladanie/odoberanie prvkov niekde v strede poľa trvá lineárny čas, v zozname zase hľadanie prvku trvá lineárny čas. Dobrou dátovou štruktúrou pre nás je vyvažovaný binárny vyhľadávací strom (napríklad AVL strom, červeno-čierny strom, ...). Takéto stromy majú časovú zložitosť každej operácie, ktorú budeme potrebovať, v najhoršom prípade $O(\log x)$, kde x je aktuálny počet prvkov v strome.

Aby sme nemuseli písať dva krát podobný kód, jeden pre horný polobal a druhý pre dolný polobal, tak použijeme nasledovný trik: pri práci s dolným polobalom všetky súradnice vynásobíme -1 a použijeme kód pre horný polobal. Všimnite si, že takto vyberieme nielen správne body, ale aj každá zvislá hraná bude v práve jednom polobale. Navyše ani nemusíme samostatne zostrojovať začiatočný konvexný obal. Jednoducho začneme s prázdnymi polobalmi a postupne pridáme všetkých $n + m$ bodov.

Ako efektívny je náš algoritmus? Pozrime sa, čo sa stane, keď pridáme nový bod. V polobale môže byť najviac $n + m$ bodov. Nájdenie miesta, kam nový bod patrí, trvá vo vyvažovanom strome $O(\log(n + m))$, toľko isto trvá aj jeho pridanie. Potom niekoľkokrát (nech je to k) vyhodíme z polobalu jedného z jeho susedov. Každú kontrolu a vyhodenie vieme opäť spraviť v čase $O(\log(n + m))$, celé vyhadzovanie teda trvá $O(k \log(n + m))$. Jedno konkrétne pridanie bodu môže byť teda vcelku pomalé – čím viac bodov z obalu zmizne, tým dlhšie nám bude trvať jeho prepočítanie. Teraz si ale treba uvedomiť, že dokopy tento algoritmus musí byť efektívny. Presnejšie, pozrime sa na jeho celkovú časovú zložitosť. Každý bod najviac raz pridáme do daného polobalu a najviac raz ho odtiaľ vyhodíme. Aj pridanie, aj vyradenie spotrebuje čas $O(\log(n + m))$, preto celková časová zložitosť nášho algoritmu je $O((n + m) \log(n + m))$.

Pamäťová zložitosť algoritmu je $O(n + m)$, lebo pre každý polobal si udržujeme jeden vyhľadávací strom, ktorý v najhoršom prípade obsahuje všetky prvky.

Vzorové riešenie je naprogramované v jazyku C++. Aby sme si ušetrili implementáciu vyváženého vyhľadávacieho stromu, použili sme *set* z STL. Premenná typu *set::iterator* je inteligentný ukazovateľ na prvok v strome. Operátory *++* a *--* na iterátore ho posunú na nasledujúci resp. predchádzajúci prvok. Metóda *upper_bound(B)* nájde v strome prvý prvok väčší od B, metóda *begin()* ukazuje na prvý prvok v strome a *end()* ukazuje o jedno za posledný prvok v strome. (Všetky intervaly v STL sú polouzavreté – vľavo uzavreté, vpravo otvorené.) Metóda *erase(a,b)* zoberie polouzavretý interval určený dvomi iterátormi a vymaže ho zo stromu.

Listing programu:

```
#include <iostream>
#include <set>
#include <cmath>
using namespace std;

struct Bod {
    double x, y;
    Bod (double xx, double yy) { x = xx; y = yy; }
    void flip() { x = -x; y = -y; }
    bool operator < (const Bod &a) const {
        if (a.x == this->x) return this->y < a.y; else return this->x < a.x;
    }
};
```

```
double vektorovy_sucin (Bod a, Bod b, Bod c) { return (b.x-a.x)*(c.y-b.y) - (b.y-a.y)*(c.x-b.x); }
double sqr (double a) { return a * a; }
double vzdialenost (Bod a, Bod b) { return sqrt (sqr (a.x - b.x) + sqr (a.y - b.y) ); }

set<Bod> horny_obal, dolny_obal;

double pridaj_do_obalu (set<Bod> &obal, Bod &bod) {
    if (obal.count (bod) > 0) return 0;
    set<Bod>::iterator stred, pred, pred2, za, za2 ;
    pred = stred = obal.upper_bound (bod);
    if (pred != obal.begin() ) pred--;
    if (stred != obal.begin() && stred != obal.end() && vektorovy_sucin(*pred,bod,*stred) <= 0)
        return 0;
    double zmena = 0;
    if (stred != obal.end() ) zmena -= vzdialenost (*pred, *stred);
    //zaciatok
    pred2 = pred;
    if (pred2 != obal.begin() ) pred2--;
    while (pred2 != pred && vektorovy_sucin (*pred2, *pred, bod) <= 0) {
        zmena -= vzdialenost (*pred, *pred2);
        pred = pred2;
        if (pred2 != obal.begin() ) pred2--;
    }
    //koniec
    za = za2 = stred;
    if (za2 != obal.end() ) za2++;
    while (za2 != obal.end() && vektorovy_sucin (bod, *za, *za2) <= 0) {
        zmena -= vzdialenost (*za, *za2);
        za = za2;
        if (za2 != obal.end() ) za2++;
    }
    if (pred != stred) zmena += vzdialenost(*pred,bod);
    if (za != obal.end()) zmena += vzdialenost(bod, *za);
    if (pred != stred) pred++;
    obal.erase (pred, za);
    obal.insert (bod);
    return zmena;
}

double update (double x, double y) {
    Bod b (x, y);
    double zmena = 0;
    zmena += pridaj_do_obalu (horny_obal, b);
    b.flip();
    zmena += pridaj_do_obalu (dolny_obal, b);
    return zmena;
}

int main() {
    int N, M;
    double x, y, obvod=0;
    cin >> N;
    for (int i = 0; i < N; i++) { cin >> x >> y; obvod += update (x, y); }
    cout << obvod << endl;
    cin >> M;
    for (int i = 0; i < M; i++) { cin >> x >> y; cout << update (x, y) << endl; }
}
```

A-II-3 Mažoretky

Na úvod trocha terminológie: Poradia mažoretiek zodpovedajú *permutáciám* čísel od 1 po n. Poradie definované v zadaní sa zvykne nazývať ich *lexikografickým usporiadaním*. Pre jednoduchosť budeme namiesto „permutácia π je v lexikografickom usporiadaní pred permutáciou ρ“ hovoriť „π je menšia ako ρ“.

Nasledujúca permutácia

Našim cieľom v prvej podúlohe je z danej permutácie $a = (a_1, \ldots, a_n)$ vyrobiť najbližšiu väčšiu. Jedinou výnimkou je posledná, najväčšia permutácia $(n, n-1, \ldots, 2, 1)$, po ktorej opäť nasleduje prvá permutácia $(1, 2, \ldots, n-1, n)$. Túto výnimku môžeme v programe ošetriť samostatne. V ďalšom texte teda predpokladáme, že zadaná permutácia a nie je najväčšia.

Pozrime sa na ľubovoľnú permutáciu b, ktorá je väčšia ako a. Čo o nej vieme povedať? Presne to, čo nám hovorí definícia ich poradia: že na prvom mieste, na ktorom sa b a a líšia, musí byť v b väčšia hodnota ako v a.

Spomedzi všetkých permutácií, ktoré sú väčšie ako a, hľadáme tú najmenšiu. Ktorá to bude? Majme dve permutácie b a c, ktoré sú obe väčšie ako a, pričom b sa od a začne líšiť neskôr ako c. Potom je zjavné, že b je menšia ako c – totiž na prvom mieste, kde sa b a c líšia, má b pôvodnú hodnotu z a, zatiaľ čo c tam má inú, väčšiu hodnotu. Ak teda chceme vyrobiť najmenšiu permutáciu väčšiu ako a, musíme sa v prvom rade snažiť nechať na začiatku čo najviac hodnôt nedotknutých.

V rôznych situáciách však bude počet hodnôt, ktoré vieme nechať nedotknuté, rôzny. Všimnime si napríklad permutáciu: $(7, 3, 1, 6, 9, 8, 5, 4, 2)$. Existuje iná permutácia tvaru $(7, 3, 1, 6, \ldots)$, ktorá je od tejto väčšia? Neexistuje – lebo zvyšné prvky $(9, 8, 5, 4, 2)$ sú už usporiadané klesajúco, a teda žiadnu väčšiu permutáciu ich prehádzaním vyrobiť nevieme.

V tomto prípade teda hľadáme permutáciu tvaru $(7, 3, 1, x, \ldots)$, kde $x > 6$. Samozrejme, čím menšie x použijeme, tým menšiu permutáciu dostaneme. V našom prípade máme na výber $x = 8$ alebo $x = 9$, použijeme teda $x = 8$.

Vieme už teda, že hľadáme permutáciu tvaru $(7, 3, 1, 8, \ldots)$. No a teraz si už len stačí uvedomiť, že všetky takéto permutácie sú zaručene väčšie ako tá, ktorú sme dostali na vstupe. Hľadáme teda najmenšiu z nich – tú, kde sú všetky ďalšie čísla uvedené v rastúcom poradí. Po permutácii $(7, 3, 1, 6, 9, 8, 5, 4, 2)$ teda nasleduje permutácia $(7, 3, 1, 8, 2, 4, 5, 6, 9)$.

Celý algoritmus výroby nasledujúcej permutácie teda môžeme sformulovať nasledovne: Nech A je pole, ktoré na začiatku obsahuje vstupnú permutáciu.

1. Idúc od konca nájdi najväčšie x také, že $A[x] < A[x+1]$.
   (Pozícia x je prvá pozícia, ktorá sa bude meniť. Väčšie x nevyhovujú, lebo za touto pozíciou už je naša permutácia usporiadaná klesajúco. Ak také x neexistuje, v poli A je posledná permutácia, zmeníme ju na prvú a skončíme.)
2. Idúc od konca nájdi najväčšie y také, že $A[y] > A[x]$.
   (Keďže y bude najväčšie možné, $A[y]$ bude najmenšie možné – spomedzi hodnôt, ktoré máme na výber, je to teda najmenšia hodnota, ktorá je väčšia ako $A[x]$. Takéto y určite existuje, lebo určite vyhovuje $y = x + 1$.)
3. Vymeň $A[x]$ a $A[y]$.
   (V tejto chvíli sme na pozíciu x dostali hodnotu, ktorá tam patrí. A navyše vieme, že v časti, ktorú meníme, sme vymenili dve po sebe nasledujúce hodnoty. Úsek poľa A od pozície $x + 1$ do konca je teda naďalej usporiadaný klesajúco.)
4. Obráť úsek poľa A od pozície $x + 1$ po koniec.
   (Z klesajúceho úseku, teda najväčšieho možného, urobíme rastúci, teda najmenší možný.)

Príklad: pre vyššie uvedenú permutáciu $(7, 3, 1, 6, 9, 8, 5, 4, 2)$ najskôr nájdeme $x = 4$ (pozícia hodnoty 6) a $y = 6$ (pozícia hodnoty 8). Potom vymeníme dotyčné dve hodnoty, čím dostaneme $(7, 3, 1, 8, 9, 6, 5, 4, 2)$. A na záver otočíme naopak klesajúci úsek $(9, 6, 5, 4, 2)$, čím dostaneme správnu nasledujúcu permutáciu.

Časová zložitosť tohto algoritmu je zjavne $O(n)$. Vieme ju však odhadnúť ešte o čosi presnejšie: keď priamo meníme pole obsahujúce permutáciu tak, ako sme to opísali v našom algoritme, je celkový počet krokov priamo úmerný počtu pozícií, ktoré sa zmenia – a teda optimálny.

Poznámka na záver: Tento algoritmus je v C++ implementovaný vo funkcii `next_permutation`. Navyše sa oplatí vedieť, že tento algoritmus bez zmeny funguje pre ľubovoľné hodnoty vo vstupnom poli, vrátane situácie, keď sú niektoré z hodnôt navzájom rovné.

Listing programu:

```
void dalsia_permutacia(vector<int> &A) {
    int x = A.size()-2;
    while (x>=0 && A[x]>=A[x+1]) --x;
    if (x >= 0) { // nerobime posledna -> prva
        int y = A.size()-1;
        while (A[y] <= A[x]) --y;
        swap( A[x], A[y] );
```

```
    }
    reverse( A.begin()+x+1, A.end() );
}
```

Protiľahlá permutácia

Máme z daného poradia mažoretiek vyrobiť poradie, v ktorom budú o $n!/2$ dní. Táto úloha sa bude ináč riešiť pre párne a ináč pre nepárne n. Pre párne n to bude jednoduchšie, začneme teda týmto prípadom.

Pre každé x od 1 po n zjavne platí, že permutácií, ktoré začínajú číslom x, je presne $(n-1)!$. Ak teda v nejaký deň vidíme prvú z týchto permutácií, vieme, že prvú permutáciu začínajúcu nasledujúcim číslom uvidíme presne o $(n-1)!$ dní.

Pozrime sa teda na našu úlohu: k danej permutácii $(a_1, \ldots, a_n)$ máme zostrojiť tú, ktorú uvidíme o $n!/2$ dní. Túto hodnotu môžeme zapísať ako $(n/2) \cdot (n-1)!$, pričom pre párne n je číslo $n/2$ celé.

Už teda vieme povedať, ako vyzerá prvé číslo hľadanej permutácie: je to jednoducho $b_1 = a_1 + n/2$. (Samozrejme počítané cyklicky, teda po n opäť nasleduje 1.) A vlastne vieme aj to, ako bude vyzerať zvyšok hľadanej permutácie. Totiž ak by napríklad a bola prvá z permutácií začínajúcich číslom a_1, tak hľadáme prvú z permutácií začínajúcich b_1, ak druhá tak druhú, a tak ďalej.

Na pozíciách 2 až n máme teda niekoľkú permutáciu čísel $\{1, 2, \ldots, a_1 - 1, a_1 + 1, \ldots, n\}$ a potrebujeme ju prerobiť na toľkú istú permutáciu čísel $\{1, 2, \ldots, b_1 - 1, b_1 + 1, \ldots, n\}$. To však vieme ľahko urobiť: stačí pre každé k prepísať k-ty najmenší prvok prvej množiny na k-ty najmenší prvok druhej množiny. To znamená, že čísla od 1 po $\min(a_1, b_1) - 1$ aj čísla od $\max(a_1, b_1) + 1$ po n zostanú nedotknuté a tým medzi nimi zmeníme hodnoty o 1.

Príklad: Nech je vstupná permutácia $(2, 7, 4, 1, 6, 8, 3, 5)$. Potom vieme, že výstupná permutácia je tvaru $(6, \ldots)$. Zvyšok dostaneme tak, že v pôvodnej postupnosti $(7, 4, 1, 6, 8, 3, 5)$ premenujeme čísla 1345678 na čísla 1234578. Výsledkom je teda permutácia $(6, 7, 3, 1, 5, 8, 2, 4)$.

Pozrime sa teraz na nepárne n. Môžeme zopakovať pozorovanie, že $n!/2 = (n/2) \cdot (n-1)!$. Teraz je ale n nepárne, a teda $n/2$ nie je celé. Označme si $c = \lfloor n/2 \rfloor$. Potom posun o $n!/2$ dní môžeme zložiť z dvoch posunov: najskôr o $c(n-1)!$ dní a potom o $(n-1)!/2$ dní.

Posun o $c(n-1)!$ dní vyriešime rovnako ako v prípade párneho n: zmeníme prvý prvok z a_1 na $b_1 = a_1 + c$ (opäť počítané cyklicky) a zvyšok permutácie upravíme, aby používal namiesto všetkých prvkov okrem a_1 všetky prvky okrem b_1.

Ostáva nám doriešiť posun o $(n-1)!/2$ dní. No a keďže tento posun v princípe ovplyvňuje posledných $n-1$ pozícii a $n-1$ je párne číslo, môžeme použiť ten istý postup, ktorým sme vyššie vyriešili vstupy s párnym n. Je tu ale jedna vec, na ktorú si musíme dať pozor: ak je hodnota a_2 veľká, nastane niekedy počas týchto $(n-1)!/2$ dní ďalšia zmena na prvom mieste permutácie. Napr. ak by sme z permutácie $(3, 4, 1, 5, 2)$ spravili $(n-1)!/2 = 12$ krokov, tak dostaneme permutáciu $(4, 1, 2, 5, 3)$, ktorá už nezačína hodnotou 3.

Najjednoduchšie riešenie tohto problému je nasledovné: ak je a_2 také veľké, že by nastalo toto pretečenie, tak namiesto $c(n-1)!$ dní sa v prvej fáze posunieme až o $(c+1)(n-1)!$ dní dopredu. Následne sa potrebujeme posunúť o $(n-1)!/2$ dní *späť*. Pri tomto posune máme istotu, že sa počas neho prvá hodnota nezmení. A zároveň vieme, že keď sa dívame len na pozície 2 až n, tak posun o $(n-1)!/2$ dopredu a dozadu vedie na tú istú postupnosť, takže môžeme spokojne použiť vyššie popísaný postup pre párne n.

Naša implementácia má časovú zložitosť lineárnu od počtu prvkov permutácie, teda $O(n)$.

Listing programu:

```
inline int cyklicky(int x, int N) { if (x>N) return x-N; return x; }

void premenuj(vector<int> &A, int stare, int nove) {
    for (int n=1; n<int(A.size()); ++n) {
        if (stare < nove && A[n]>stare && A[n]<=nove) --A[n];
        if (stare > nove && A[n]>=nove && A[n]<stare) ++A[n];
    }
}

void opacna_permutacia(vector<int> &A) {
    int N = A.size();
```

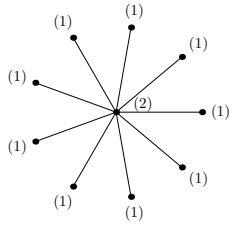
```
    int old=A[0];
    A[0] = cyklicky(A[0]+N/2,N);
    premenuj(A,old,A[0]);
    if (N % 2 == 0) return;

    int velke = ((N+1)/2) + (A[0]<=N/2);
    if (A[1] >= velke) {
        old = A[0];
        A[0] = cyklicky(A[0]+1,N);
        premenuj(A,old,A[0]);
    }

    premenuj(A,A[0],N);
    vector<int> B(A.begin()+1,A.end());
    opacna_permutacia(B);
    for (int n=1; n<N; ++n) A[n]=B[n-1];
    premenuj(A,N,A[0]);
}
```

A-II-4 Grafový počítač a kompletné grafy

Výroba kompletného grafu v lineárnom čase



Na klasickom počítači na výrobu kompletného grafu K_n potrebujeme spraviť $\Theta(n^2)$ krokov, keďže tento graf má až $\binom{n}{2}$ hrán.

Prvé lepšie riešenie na grafovom počítači je postupne vyrobiť a pozliepať dokopy hviezdy s 1 až n vrcholmi. Predstavme si totiž, že máme K_n, v ktorom majú všetky vrcholy značku 1, a hviezdu, v ktorej je n vrcholov so značkou 1 a jeden vrchol so značkou 2. Vhodným volaním `Join` vieme tieto dva grafy spojiť tak, aby sme získali graf K_{n+1}. Následne jeho novému vrcholu zmeníme značku na 1, do hviezdy prídáme nový vrchol a novú hranu a môžeme celý proces začať odznova.

Takéto riešenie má časovú zložitosť $\Theta(n)$, teda lineárnu od počtu vrcholov.

Výroba kompletného grafu pre mocniny dvoch

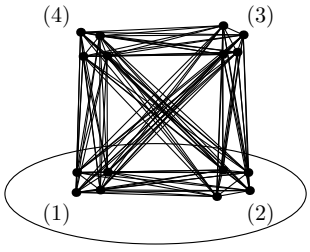
Pre jednoduchosť zatiaľ predpokladajme, že n je mocninou dvoch. Ukážeme, ako celý graf K_n zostrojiť v čase $\Theta(\log n)$.

Ak chceme dosiahnuť takúto časovú zložitosť, musíme v princípe vedieť pomocou konštantne veľa krokov zdvojnásobiť veľkosť zostrojeného kompletného grafu.

Zdvojnásobiť počet vrcholov je ľahké. Keď máme kompletný graf K_n, môžeme použiť `Join` na dve jeho kópie, pričom nastavíme `veq=none`. Tým dostaneme graf s $2n$ vrcholmi. Do kompletného grafu mu však ešte chýba celkom veľa hrán – presne sú to hrany (x, y), kde $x \leq n < y$.

Samozrejme, po jednej tieto hrany pridávať nemôžeme, to by trvalo pridlho. Potrebujeme ich vedieť pridať veľa naraz. My však už máme graf, ktorý obsahuje veľa hrán: samotný K_n. Graf K_{2n} teda „nazliepame“ z niekoľkých kópií grafu K_n.

Teraz prichádza jediný „trik“ v celom riešení. Aby sme vedeli šikovne nazliepať menšie grafy na väčší, vhodne použijeme značky vrcholov.



Majme graf K_n, pričom n je párne a platí, že v K_n má polovica vrcholov značku 1 a polovica značku 2. Vyrobíme si 5 ďalších kópií tohto grafu, ktoré budú namiesto značiek (1,2) používať značky (1,3), (1,4), (2,3), (2,4) a (3,4).

Čo sa stane, keď týchto šesť grafov postupne pospájam volaniami `Join`, pričom `veq=value`? Zjavne dostaneme kompletný graf K_{2n}, v ktorom bude mať po $n/2$ vrcholov každú zo značiek 1, 2, 3 a 4. No a v takomto grafe už len stačí dvoma príkazmi zmeniť všetky značky z 3 na 1 a zo 4 na 2, čím dostaneme správne označený graf K_{2n}.

Na obrázku je príklad grafu K_{16}, zostrojeného týmto postupom zo šiestich kópií grafu K_8. Zakrúžkovaná časť grafu zodpovedá pôvodnému grafu K_8, z ktorého sme konštrukciu začínali.

Listing programu:

```
function kompletny_moc2(n: Integer): Graph;       { funguje len pre n rovne kladnej mocnine dvoch }
var stary, novy : Graph;
begin
    if n = 2 then begin
        novy := EmptyG;
        AddV(novy,1);
        AddV(novy,2);
        AddE(novy,1,2,undef);
        kompletny_moc2 := novy;
    end else begin
        stary := kompletny_moc2(n div 2);
        novy := stary;                                          { zoberieme graf s labelmi (1,2) }
        ReplaceV(stary,2,3); novy := Join(novy,stary,IM_value,IM_any);                  { a (1,3) }
        ReplaceV(stary,3,4); novy := Join(novy,stary,IM_value,IM_any);                  { a (1,4) }
        ReplaceV(stary,1,2); novy := Join(novy,stary,IM_value,IM_any);                  { a (2,4) }
        ReplaceV(stary,4,3); novy := Join(novy,stary,IM_value,IM_any);                  { a (2,3) }
        ReplaceV(stary,2,4); novy := Join(novy,stary,IM_value,IM_any);                  { a (3,4) }
        ReplaceV(novy,3,1); ReplaceV(novy,4,2);  { a na zaver uz len upravime znacky z 1234 na 12 }
        kompletny_moc2 := novy;
    end;
end;
```

Výroba kompletného grafu pre všeobecný počet vrcholov

Najjednoduchšie riešenie pozostáva z dvoch krokov: keď máme vyrobiť K_n, najskôr vyrobíme K_m, kde m je prvá mocnina dvoch väčšia alebo rovná n. Následne z tohto grafu zahodíme $m - n$ vrcholov, spolu s hranami, ktoré do nich vedú.

Ako ale rýchlo zahodiť veľa vrcholov? Pomôžeme si jednoduchým trikom. Najskôr zostrojíme graf s $m - n$ izolovanými vrcholmi, to vieme spraviť ľahko. Všetkým vrcholom K_m dáme značku 1, všetkým vrcholom nášho nového grafu dáme značku 2. Vhodným zavolaním `Join` teraz vieme vyrobiť graf K_m, v ktorom $m - n$ vrcholov bude mať značku 2 a ostatných n značku 1. No a vhodným zavolaním `Common` na pôvodné K_m a označené K_m dostaneme želaný graf K_n.

Akú má tento algoritmus časovú zložitosť? Stačí si všimnúť, že nutne $m \leq 2n$. Preto má zostrojenie kompletného grafu K_m časovú zložitosť $\Theta(\log n)$. Analogicky v nanajvýš logaritmickom čase zostrojíme aj prázdny graf s $m - n$ vrcholmi; zvyšok algoritmu už beží v konštantnom čase. Celková časová zložitosť je teda $\Theta(\log n)$.

Listing programu:

```
{ vyrobi graf s n vrcholmi (kazdy so znackou 2) a 0 hranami }
function prazdny(n : Integer) : Graph;
var tmp : Graph;
begin
    if n=0 then prazdny:=EmptyG else begin
        tmp := prazdny(n div 2);
        tmp := Join(tmp,tmp,IM_none,IM_none);
        if n mod 2 = 1 then AddV(tmp,2);
        prazdny := tmp;
    end;
end;

{ vyrobi kompletny graf s n vrcholmi }
function kompletny(n : Integer) : Graph;
var m : Integer;
    g, h : Graph;
begin
    m := 2; while m<n do m := 2*m;
    g := kompletny_moc2(m);
    SetAllV(g,1);
    h := prazdny(m-n);
    h := Join(h,g,IM_any,IM_any); { znacky sa beru prednostne z h }
    kompletny := Common(g,h,IM_value,IM_any);
end;
```

Výroba kompletného grafu trochu ináč

Bez nejakého zahadzovania (alebo naopak pridávania) vrcholov sa pri konštrukcii všeobecného K_n nezaobídeme – totiž jediné, čo vieme robiť, je, že z vhodne označeného K_{2x} vieme vyrobiť K_{4x}. Grafy, ktorých počet vrcholov nie je deliteľný 4, teda pomocou nášho postupu priamočiaro vyrobiť nevieme.

Namiesto toho, aby sme zahadzovali prebytočné vrcholy všetky naraz na konci, ich však môžeme zahadzovať aj počas algoritmu. Jeden možný spôsob, ako to robiť, vyzerá nasledovne:

Nech máme vyrobiť graf K_n. Pre $n \leq 3$ ho vyrobíme priamo. Inak nájdeme najmenšie m také, že $4m \geq n$. Rekurzívnym volaním vyrobíme graf K_{2m}. Z jeho šiestich kópií vyrobíme graf K_{4m} a z toho následne nanajvýš tri vrcholy zahodíme, čím dostaneme želaný graf K_n.

Všimnite si, že pri rekurzívnom volaní klesol počet vrcholov zostrojovaného grafu približne na polovicu. Poriadnejším sformulovaním tohto pozorovania sa dá dokázať, že aj tento algoritmus má časovú zložitosť $\Theta(\log n)$.

Efektívne hľadanie najväčšej kliky

Otestovať, či graf G obsahuje kliku veľkosti x, vieme ľahko: Zostrojíme kompletný graf K_x a zavoláme `Find`, nech nám nájde jeden jeho výskyt v G. Podľa toho, či `Find` naozaj nájde výskyt alebo vráti `EmptyG`, vieme, či sa v G klika danej veľkosti nachádza alebo nie.

O grafe G, ktorý má $n \geq 1$ vrcholov, priamo vieme, že jeho klikovosť je aspoň 1 (lebo má vrchol) a zároveň je menej ako $n + 1$ (lebo toľko vrcholov nemá). Správnu hodnotu môžeme na tomto intervale nájsť binárnym vyhľadávaním: budeme postupne generovať kompletné grafy rôznych veľkostí a testovať, či sa v G nachádzajú.

Pri binárnom vyhľadávaní potrebujeme spraviť $\log_2 n$ testov. Tieto testy však nie sú „zadarmo“: pri každom z nich treba zostrojiť nejaký kompletný graf.

Ak by zadaný graf G bol hustý (napr. $G = K_n$), tak každý z grafov, ktoré zostrojíme počas binárneho vyhľadávania, bude mať aspoň $n/2$ vrcholov. Preto každá iterácia binárneho vyhľadávania bude mať časovú zložitosť $\Theta(\log n)$, a teda celková časová zložitosť tohto algoritmu bude $\Theta(\log^2 n)$.

Ešte efektívnejšie hľadanie najväčšej kliky

Aby sme predchádzajúci algoritmus zrýchlili, potrebujeme sa zbaviť toho, čo sa zdá byť zbytočne pomalé – zostrojovať graf, ktorý v G hľadáme, vždy úplne odznova.

Ak chceme dosiahnuť optimálnu časovú zložitosť binárneho vyhľadávania, potrebujeme na každú otázku spotrebovať len konštantný čas – a teda nový graf, ktorý chceme vyhľadávať, musíme vedieť v konštantnom čase vyrobiť.

Naše riešenie bude fungovať v dvoch fázach. V prvej fáze budeme postupne zostrojovať grafy K_1, K_2, K_4, K_8, atď., až kým nenájdeme prvý z nich, ktorý sa v danom grafe G nenachádza. V tomto okamihu vieme, že sa klikovosť grafu G nachádza v intervale $\langle 2^a, 2^{a+1}\rangle$, a tiež máme zostrojené kompletné grafy veľkostí 2^0 až 2^{a+1}. Keďže každý graf vieme z predchádzajúceho zostrojiť v konštantnom čase a platí $2^a \leq n$, táto fáza trvá $O(\log n)$.

Od tohto okamihu budeme v intervale, ktorý sme práve našli, binárne vyhľadávať. Všimnite si, že po každej iterácii bude dĺžka intervalu, v ktorom hľadáme, rovná mocnine dvojky. Totiž majme interval $\langle x, x + 2^y)$ pre $y > 0$. V jeho strede leží hodnota $x + 2^{y-1}$ a keď sa opýtame na ňu, dostaneme tak či onak interval presne polovičnej dĺžky. Navyše si všimnite, že stále bude platiť $x \geq 2^y$, lebo na začiatku to platí a počas hľadania sa y bude zmenšovať, zatiaľ čo x nie.

Počas binárneho vyhľadávania si budeme v jednej premennej udržiavať kompletný graf K_x, ktorého počet vrcholov bude rovný spodnej hranici intervalu, v ktorom práve hľadáme.

Každá iterácia binárneho vyhľadávania teda bude vyzerať nasledovne: Hľadáme v intervale $\langle x, x + 2^y)$ pre nejaké $y > 0$. Nech $z = 2^{y-1}$. Potom hodnota uprostred intervalu, v ktorom hľadáme, je práve $x + z$. Zoberieme graf K_x a graf K_z (ten máme ešte z prvej fázy, keďže z je mocnina dvoch) a vyrobíme z nich graf K_{x+z}. Na ten sa opýtame. Ak ho graf G obsahuje, ďalej pokračujeme s intervalom $\langle x + z, x + 2z)$, ak nie, tak s intervalom $\langle x, x + z)$.

Jediné, čo potrebujeme ukázať, je, ako z grafov K_x a K_z vyrobiť v konštantnom čase graf K_{x+z}. Konštrukcia bude podobná ako keď sme z viacerých kópií K_n vyrábali K_{2n}. Tentokrát využijeme, že platí $x \geq 2^y$. Graf K_x má teda aspoň dvakrát toľko vrcholov ako graf K_z. Konštrukciu grafu K_{x+z} rozpíšeme po myšlienkových krokoch.

- Na začiatku sú v premenných `K_x` a `K_z` grafy K_x a K_z, všetky ich vrcholy majú značku 1.
- Vyrobíme graf C_{12}, čo je K_x, v ktorom má z vrcholov značku 2 a ostatné značku 1.

```
K_z_2 := SetAllV(K_z,2); C_12 := Join(K_z_2,K_x,any,any);
```

- Vyrobíme graf D s $x + z$ vrcholmi, z ktorých z má značku 2, z má značku 3 a ostatné značku 1. Do kompletného grafu budú grafu D chýbať hrany medzi vrcholmi so značkami 2 a 3.

  ```
  C_13 := ReplaceV(C_12,2,3); D := Join(C_12,C_13,value,any);
  ```

- Preznačíme vrcholy grafu C_{12}, aby sme mali istotu, že vrcholy, ktoré majú id 1 až z, majú značku 1.

  ```
  tmp := Join(K_z,C_{12},value,any);
  ```

- Vyrobíme z práve preznačeného grafu graf C_{123}, v ktorom vrcholy s id 1 až z preznačíme na značku 3. V tomto grafe máme teda po z vrcholoch so značkami 2 a 3, ostatné vrcholy (ak tam ešte nejaké sú) majú značku 1.

  ```
  K_z_3 := SetAllV(K_z,3); C_123 := Join(K_z_3,tmp,id,any);
  ```

- Spojením grafov D a C_{123} dostávame hľadaný graf K_{x+z}.

  ```
  vysledok := Join(D,C_123,value,any);
  ```

Takto dostávame riešenie, ktorého celková časová zložitosť je $O(\log n)$. Poznámka na záver: aj pamäťová zložitosť tohto riešenia je $O(\log n)$, presnejšie, potrebujeme si súčasne pamätať $O(\log n)$ grafov. Existuje aj riešenie s časovou zložitosťou $O(\log n)$, ktoré si v každom okamihu pamätá len konštantne veľa grafov. Jedna možnosť je namiesto postupne idúcich mocnín dvojky použiť Fibonacciho čísla. Najskôr nájdeme také x, že hľadané číslo leží v intervale $\langle F_x, F_{x+1})$ a od tejto chvíle si stačí pamätať tri kompletné grafy: keď máme interval $\langle a, a + F_y)$, tak si pamätáme kompletné grafy veľkosti a, F_y a F_{y-1}.

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY ŠIESTY ROČNÍK OLYMPIÁDY V INFORMATIKE

Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX