B-II-1 Letenka

Pomalšie ale funkčné riešenia

Jednou možnosťou, ako úlohu vyriešiť, je vyskúšať všetky lety, potom všetky dvojice letov a nakoniec všetky trojice letov. Z nich vyberieme len tie, ktoré zodpovedajú našim požiadavkam. (Lety musia na seba nadväzovať a musí sedieť začiatočné aj koncové letisko.) Spomedzi tých, ktoré vyhovujú, si vyberieme najlacnejšiu možnosť. Takto postupne nájdeme najskôr najlacnejší priamy let, potom najlacnejší let na jeden prestup a potom najlacnejší let na dva prestupy.

Takéto riešenie má časovú zložitosť až kubickú od počtu letov, teda $O(\ell^3)$. A zjavne nie je optimálne – robíme pri ňom veľa zbytočnej práce. Napríklad keď sme si vybrali prvý let z Viedne do Mníchova, nemá zmysel pre druhý let skúšať všetky možné lety po celom svete. Omnoho šikovnejšie je skúsiť len lety z Mníchova. A keď sme si už odtiaľ vybrali druhý let do Frankfurtu a našim cieľom je Zurich, pre tretí let máme ešte lepšiu situáciu. Nemusíme predsa prezerať všetky lety z Frankfurtu, keď jediné, čo potrebujeme, je najlacnejší priamy let do Zurichu.

Tieto pozorovania nás privádzajú k záveru, že si potrebujeme lety uložiť v pamäti tak, aby sme vedeli efektívne robiť dva typy operácii: O1: „nájsť všetky lety z daného letiska“ a O2: „pre danú dvojicu letísk povedať, či sú spojené priamym letom, a ak áno, akým najlacnejším“. Takýto spôsob uloženia údajov existuje, ale je zbytočne komplikovaný, preto nebudeme zachádzať do detailov.

Iný možný prístup vedúci k trocha horšiemu riešeniu: Stačí nám uložiť informácie o letoch tak, aby sme vedeli robiť operáciu O2. Potom najlacnejší let z A do B na dva prestupy nájdeme tak, že vyskúšame všetky možné prestupové letiská C a D a zakaždým si zistíme najlacnejšiu cenu cesty $A \to C \to D \to B$. Ak si počet rôznych letísk na vstupe označíme n, vieme o tomto riešení povedať, že rádovo n^2-krát budeme potrebovať pre nejakú dvojicu letísk spraviť operáciu O2. Čím lepšie ju budeme vedieť robiť, tým rýchlejšie toto riešenie bude.

Pri všetkých týchto postupoch ešte stále skúšame aj niektoré možnosti, ktoré by sme skúšať nemuseli. Napríklad v poslednom uvedenom riešení nemá zmysel skúšať všetky dvojice prestupných letísk (C, D): Akonáhle vieme, že medzi A a C nevedie priamy let, nemusíme skúšať žiadne D. A pre konkrétne C stačí skúšať tie D, do ktorých sa z C vieme dostať priamym letom.

Riešenie v čase priamo úmernom počtu letov

V prvom rade si môžeme uvedomiť, že keďže každé letisko má trojpísmenný identifikátor, je len $26^3 = 17576$ možností pre názov letiska. Teda nie je problém spraviť si pole veľkosti 26^3 a v ňom si niečo uložiť o každom z letísk, ktoré sa na vstupe vyskytnú.

Začneme tým, že načítame začiatočné letisko A a cieľové letisko B. Následne načítame celý zoznam letov do jedného poľa. A už počas načítavania letov (alebo hneď po ňom) si viem vyplniť dve ďalšie polia: pre každé letisko C cenu najlacnejšieho letu $A \to C$, a pre každé letisko D cenu najlacnejšieho letu $D \to B$.

Akonáhle teda skončilo načítanie, viem už cenu najlacnejšieho priameho letu $A \to B$ (mám ju uloženú hneď na dvoch miestach).

Následne spočítam cenu najlacnejšieho letu s jedným prestupom: pre každé letisko C, ktoré sa na vstupe aspoň raz vyskytlo, sa pozriem na súčet cien najlacnejšieho letu $A \to C$ a najlacnejšieho letu $C \to B$.

Na záver spočítam cenu najlacnejšieho letu s dvomi prestupmi: Prejdem celý zoznam letov. Pre každý let vyskúšam možnosť: „čo ak je toto optimálny let z C do D?“ Zistím teda cenu práve skúšaného letu, plus cenu letu z A do miesta, kde práve skúšaný let začína, plus cenu letu z miesta, kde práve skúšaný let končí, do B.

Listing programu:

```
const NEKONECNO = 987654321;
      POCET_LETISK = 26*26*26;

type let = record odkial, kam, cena : longint; end;
var zaciatok, koniec : longint;
    lety : array of let;
    L : longint;
```

```
{ pomocna funkcia: skonvertuje trojpismenovy retazec na cislo z [0,26*26*26) }
function retazec_na_cislo(s : string) : longint;
begin retazec_na_cislo := 26*26*(ord(s[1])-65) + 26*(ord(s[2])-65) + ord(s[3])-65; end;

{ pomocna procedura: zmensi hodnotu v premennej, ak vies }
procedure vylepsi(var premenna : longint; nova_hodnota : longint);
begin if nova_hodnota < premenna then premenna := nova_hodnota; end;

{ procedura na nacitanie kodu letiska a vyroby cisla letiska z neho }
procedure nacitaj_letisko(var kod : longint);
var c : char;
    letisko : string;
begin
    letisko := '';
    read(c); letisko += c; read(c); letisko += c; read(c); letisko += c;
    read(c); { aj whitespace za nazvom nacitame }
    kod := retazec_na_cislo(letisko);
end;

procedure nacitaj_vstup;
var i : longint;
begin
    nacitaj_letisko(zaciatok); nacitaj_letisko(koniec);
    readln(L);
    setlength(lety,L);
    for i:=0 to L-1 do begin
        nacitaj_letisko(lety[i].odkial); nacitaj_letisko(lety[i].kam); readln(lety[i].cena);
    end;
end;

procedure vypocitaj_vystup;
var najlepsi_prilet, najlepsi_odlet : array[0..POCET_LETISK-1] of longint;
    i, odpoved1, odpoved2, odpoved3 : longint;
begin
    for i:=0 to POCET_LETISK-1 do najlepsi_prilet[i] := NEKONECNO;
    for i:=0 to POCET_LETISK-1 do najlepsi_odlet[i]  := NEKONECNO;

    { vypocitame ceny priamych letov zo zaciatku a do konca trasy }
    for i:=0 to L-1 do begin
        if lety[i].odkial = zaciatok then vylepsi( najlepsi_prilet[ lety[i].kam ],   lety[i].cena );
        if lety[i].kam    = koniec   then vylepsi( najlepsi_odlet[ lety[i].odkial ], lety[i].cena );
    end;
    odpoved1 := najlepsi_odlet[zaciatok];

    { pre kazde letisko vyskusame letiet zo zaciatku na koniec cez neho }
    odpoved2 := odpoved1;
    for i:=0 to POCET_LETISK-1 do vylepsi( odpoved2, najlepsi_prilet[i]+najlepsi_odlet[i] );

    { a pre kazdy let vyskusame letiet zo zaciatku na koniec cez neho }
    odpoved3 := odpoved2;
    for i:=0 to L-1 do
        vylepsi(odpoved3, najlepsi_prilet[lety[i].odkial]+lety[i].cena+najlepsi_odlet[lety[i].kam]);

    { vypiseme odpovede }
    if odpoved1 >= NEKONECNO then writeln('neexistuje') else writeln(odpoved1);
    if odpoved2 >= NEKONECNO then writeln('neexistuje') else writeln(odpoved2);
    if odpoved3 >= NEKONECNO then writeln('neexistuje') else writeln(odpoved3);
end;

begin
    nacitaj_vstup;
    vypocitaj_vystup;
end.
```

Iné vzorové riešenie

Ukážeme si ešte jedno riešenie s optimálnou časovou zložitosťou $O(\ell)$. Toto riešenie je myšlienkovo o trochu ťažšie, ale má dve nesporné výhody: veľmi ľahko sa implementuje a navyše ostane efektívne aj vtedy, keď by sme sa zaujímali o lety s tromi, štyrmi, či piatimi prestupmi.

Hlavná myšlienka: Ako vyzerá najlacnejší let z A do B s dvomi prestupmi? Tak, že najskôr s jedným prestupom prídeme z A na nejaké letisko C, a odtiaľ najlacnejším letom na letisko B. A ktorú možnosť použijeme pri ceste z A na C? No predsa najlacnejšiu.

Celé riešenie bude teraz úplne priamočiare: Trikrát prejdeme cez zoznam letov. Pri prvom prechode si pre *každé letisko* zistíme cenu najlacnejšieho priameho letu naň. Pri druhom prechode použijeme tieto údaje na

to, aby sme pre každé letisko zistili cenu najlacnejšieho letu naň pomocou dvoch letov. A v treťom prechode spravíme ešte raz to isté: z cien pre dva lety spočítame ceny pre tri lety.

(Ide vlastne o jednoduchú aplikáciu postupu nazývaného *dynamické programovanie*. Navyše si pozorný čitateľ určite všimol, že v treťom prechode si už stačilo pamätať optimálnu cenu len pre naše cieľové letisko. Keďže nám to však nepokazí zložitosť, budeme si to pamätať pre všetky, bude sa to ľahšie implementovať. Časová zložitosť bude naďalej lineárna od počtu letov.)

Nasleduje mierne lenivá implementácia tohto postupu v C++: aby sme nemuseli prečíslovať letiská, použijeme namiesto toho hešovacie tabuľky.

Listing programu:

```
#include <iostream>
#include <vector>
#include <tr1/unordered_set>
#include <tr1/unordered_map>
using namespace std;
using namespace std::tr1;

const int NEKONECNO = 987654321;

struct let { string odkial, kam; int cena; };

string zaciatok, koniec;       // z ktoreho letiska a na ktore letime
vector<let> lety;              // zoznam letov zo vstupu
unordered_set<string> letiska; // zoznam letisk zo vstupu

// najlepsia_cena[x][YYY]: najlepsia cena, za ktoru vieme prist na letisko YYY pomocou x letov
unordered_map<string,int> najlepsia_cena[4];

void nacitaj_vstup() {
    cin >> zaciatok >> koniec;
    int L; cin >> L;
    while (L--) {
        let T; cin >> T.odkial >> T.kam >> T.cena;
        lety.push_back(T);
        letiska.insert(T.odkial); letiska.insert(T.kam);
    }
}

void inicializuj() {
    for (int let=0; let<=3; ++let)
        for (unordered_set<string>::iterator it=letiska.begin(); it!=letiska.end(); ++it)
            najlepsia_cena[let][*it] = NEKONECNO;
    najlepsia_cena[0][zaciatok] = 0;
}

void vypocitaj() {
    for (int let=1; let<=3; ++let)
        // pre kazde letisko vieme najlepsi sposob na (let-1) letov, chceme najlepsi na (let) letov
        for (unsigned i=0; i<lety.size(); ++i) {
            int tento_sposob = najlepsia_cena[let-1][lety[i].odkial] + lety[i].cena;
            if (tento_sposob < najlepsia_cena[let][lety[i].kam])
                najlepsia_cena[let][lety[i].kam] = tento_sposob;
        }
}

void vypis_vystup() {
    int odpoved1 = najlepsia_cena[1][koniec];
    if (odpoved1 == NEKONECNO) cout << "neexistuje" << endl; else cout << odpoved1 << endl;
    int odpoved2 = min( odpoved1, najlepsia_cena[2][koniec] );
    if (odpoved2 == NEKONECNO) cout << "neexistuje" << endl; else cout << odpoved2 << endl;
    int odpoved3 = min( odpoved2, najlepsia_cena[3][koniec] );
    if (odpoved3 == NEKONECNO) cout << "neexistuje" << endl; else cout << odpoved3 << endl;
}

int main() {
    nacitaj_vstup();
    inicializuj();
    vypocitaj();
    vypis_vystup();
}
```

B-II-2 Benátky

Pomalšie riešenie

Jedno možné jednoduché riešenie vyzerá nasledovne: Samostatne pre každý počet zatopených poschodí p prejdeme celú ulicu a spočítame počet ostrovov. Presnejšie, keďže každý ostrov práve na jednom mieste začína, stačí spočítať všetky začiatky ostrovov. Začiatok ostrovu je taký dom, ktorý má na ľavej strane vodu (teda naľavo od neho je dom, ktorý už je celý pod vodou). Výnimkou je prvý dom: ak ešte nie je celý pod vodou, tak aj on je začiatkom ostrova.

V programe to ľahko implementujeme napr. tak, že ku výškam domov, ktoré si uložíme ako $H[1]$ až $H[n]$, doplníme „dom nulovej výšky“: $H[0] = 0$. Teraz pre každé $i = 1 \ldots n$ platí: Ak je zatopených presne p poschodí, tak dom i je začiatkom ostrova vtedy a len vtedy, ak $H[i] \geq p$ a zároveň $H[i-1] < p$.

Listing programu:

```
var vysky: array[0..10000] of longint;
    pocet,voda,i,n:longint;
begin
    readln(n);
    for i:=1 to n do read(vysky[i]);
    vysky[0]:=0;
    for voda:=1 to n do begin
        pocet:=0;
        for i:=1 to n do if (vysky[i]>=voda) and (vysky[i-1]<voda) then inc(pocet);
        write(pocet, ' ');
    end;
    writeln;
end.
```

Takéto počítanie však nie je príliš efektívne – spravíme pri ňom spolu až rádovo n^2 krokov. Existujú však aj lepšie postupy. Jeden z nich si ukážeme.

Vzorové riešenie

Budeme postupne simulovať dvíhanie sa hladiny. Dá sa všimnúť, že keď sa domy ponoria o jedno poschodie, nezmení sa toho príliš veľa. Presnejšie, zmena nastane len vtedy, keď sa strecha nejakého domu zaplaví. Vtedy sa možno nejak (najviac o 1) zmení počet ostrovov.

Vzorové riešenie teda bude nasledovné. Rozdelíme si domy do n chlievikov podľa ich výšky. (V prvom chlieviku budú jednoposchodové, v druhom dvojposchodové...) V cykle budeme postupne zvyšovať počet zatopených poschodí p od 1 po n. Vždy, keď zatopíme nové poschodie p, tak postupne po jednom „ponoríme“ domy v $(p-1)$-tom chlieviku, pričom si neustále pamätáme a upravujeme aktuálny počet ostrovov.

Ponoriť dom znamená, že ho akoby odstránime z ulice. T.j., do poľa boolovských premenných si o ňom zaznačíme, že už je celý pod vodou. Následne sa pozrieme, či sú už domy, ktoré s ním bezprostredne susedia, celé ponorené alebo nie. Mohli nastať tri prípady:

- Ak pred ním aj za ním už je voda, počet ostrovov sa zníži o 1.
  (Tento dom bol samostatný ostrov a ten práve zanikol.)
- Ak pred ním aj za ním je nepotopený dom, počet sa zvýši o 1.
  (Potopením tohto domu sa jeden ostrov rozpojil na dva.)
- Ak na jednej strane je voda a na druhej nepotopený dom, počet ostrovov sa nezmení.
  (Potopením tohto domu sa len zmenšila šírka jedného z ostrovov.)

Časová zložitosť algoritmu bude lineárna od počtu domov, teda $O(n)$. To preto, že rozdelenie domov do chlievikov vieme spraviť v lineárnom čase a následne už len každý dom raz spracujeme.

Listing programu:

```
var vyska: array [1..1000007] of longint;
    vynoreny: array [0..1000008] of boolean;
```

```
    chlievik: array [0..1000007] of array of longint;
    pocet:array [0..1000007] of longint;
    n,i,voda,pocetOstrovov,cisloDomu:longint;
begin
    readln(n);
    { zistim velkosti chlievikov, nacitam vstup a nastavim pole vynoreny }
    for i:=0 to n do pocet[i] := 0;
    for i:=1 to n do begin
        read(vyska[i]);
        vynoreny[i] := true;
        inc(pocet[vyska[i]]);
    end;
    vynoreny[0] := false;
    vynoreny[n+1] := false;
    { podla zistenych velkosti nastavim velkosti chlievikov }
    for i:=0 to n do begin
        setlength(chlievik[i], pocet[i]);
        pocet[i] := 0;
    end;
    { naplnim chlieviky cislami budov}
    for i:=1 to n do begin
        chlievik[vyska[i],pocet[vyska[i]]]:=i;
        inc(pocet[vyska[i]]);
    end;
    { na zaciatku je jeden ostrov }
    pocetOstrovov := 1;
    for voda:=1 to n do begin
        { postupne zaplavujeme poschodia: pre domy so spravnou vyskou potopime }
        for i:=1 to pocet[voda-1] do begin
            cisloDomu := chlievik[voda-1,i-1];
            { vsimnite si, ze sa pocet ostrovov zmeni podla podmienok}
            dec(pocetOstrovov);
            if (vynoreny[cisloDomu-1]) then inc(pocetOstrovov);
            if (vynoreny[cisloDomu+1]) then inc(pocetOstrovov);
            vynoreny[cisloDomu] := false;
        end;
        write(pocetOstrovov,' ');
    end;
    writeln;
end.
```

B-II-3 Aničkine darčeky

Podúloha a)

Algoritmus, ktorý ste programovali pre Aničku, sa volá binárne vyhľadávanie (v angličtine binary search).

Stručne si pripomeňme hlavnú myšlienku, ktorá bola podrobnejšie popísaná v domácom kole: Máme usporiadanú postupnosť čísel, medzi ktorými chceme nájsť niektoré konkrétne. Tie si pamätáme v poli, prípadne ich vieme zistiť aj bez poľa (ako dátumy v decembri v domácom kole). Na začiatku hľadáme v intervale celého rozsahu poľa a postupne ho vhodne zmenšujeme na polovičnú veľkosť, až kým nehľadáme na intervale dĺžky 1 – teda kým nám neostal už len jediný kandidát.

Myšlienka binárneho vyhľadávania je síce ľahká, ale pri implementácii je veľmi ľahké urobiť kopu chýb „o plus mínus jedna“ a tak stvoriť program, ktorý dá občas nesprávnu odpoveď, či sa dokonca zacyklí. Týmto typickým chybám sa dá vyhnúť vhodným myšlienkovým prístupom, ktorý si teraz ukážeme.

Interval, v ktorom hľadáme, si budeme pamätať ako polootvorený interval, konkrétne $\langle z, k)$. Tento zápis znamená, že na pozícii z (začiatok) je prvá hodnota, ktorá už do intervalu patrí, a k (koniec) je prvá hodnota, ktorá už do intervalu nepatrí. Teda napr. polootvorený interval $\langle 5, 8)$ obsahuje celé čísla 5, 6 a 7, ale už nie 8.

Táto reprezentácia má oproti iným reprezentáciám (napr. uzavretý interval) veľa výhod. Skúste si rozmyslieť ako by ste vyjadrili dĺžku intervalu jedným a druhým spôsobom alebo ako by ste zapísali prázdny interval.

Ako tieto intervaly využiť pri binárnom vyhľadávaní? Jednoducho – rozdelíme si prvky v poli na dva typy: *zlé a dobré*. V našom prípade zlé prvky sú tie darčeky, ktoré sú prilacné, a dobré prvky sú tie darčeky, ktoré sú už dostatočne drahé. To, čo hľadáme, je prvý dobrý prvok v poli. Urobíme to tak, že nájdeme súčasne posledný zlý aj prvý dobrý prvok. Počas hľadania si budeme udržiavať vyššie spomenuté dve premenné z a k. Budeme si pritom dávať pozor, aby vždy platilo: prvok na pozícii z je zlý a prvok na pozícii k je dobrý.

Keďže pole je usporiadané, vieme, že všetky prvky naľavo od pozície z sú nutne tiež zlé. (Ak je siedmy darček príliš lacný, je aj každý z prvých šiestich darčekov príliš lacný.) A podobne, všetky prvky napravo od pozície k sú tiež dobré. Jedine prvky medzi z a k sú teda neznáme.

Ktorý prvok môže teda byť posledným zlým prvkom v poli? Do úvahy pripadajú práve pozície z polootvoreného intervalu $\langle z, k)$. V tejto chvíli je jasné, kedy binárneho vyhľadávanie skončí: vtedy, keď už ostane len jeden kandidát. Teda vtedy, keď $k - z = 1$.

Aj krok binárneho vyhľadávania je teraz priamočiary: pozrieme sa do stredu intervalu, na pozíciu[1] $m = \lfloor (z + k)/2 \rfloor$. Ak je na tejto pozícii dobrý prvok (dostatočne drahý darček), môžeme zmeniť k na m. V opačnom prípade zmeníme z na m. V oboch prípadoch sme takto skrátili interval, v ktorom hľadáme, približne na polovicu.[2]

Posledné, čo potrebujeme vyriešiť, je inicializácia: na začiatku potrebujeme nastaviť hodnoty z a k tak, aby platila vyššie uvedená podmienka (odborne nazývaná *invariant*).

Majme teda v poli A na pozíciách 0 až $n - 1$ uložené ceny darčekov od najmenšej po najväčšiu. Jedna možnosť je začať vyhľadávanie tým, že sa pozrieme na pozície 0 a $n - 1$, či nenastal nejaký zo špeciálnych prípadov: všetky prvky zlé alebo naopak všetky dobré – v našom prípade teda všetky darčeky prilacné alebo všetky dostatočne drahé. Ak nastal špeciálny prípad, ošetríme ho, a ak nie, môžeme nastaviť $z = 0$, $k = n - 1$ a spustiť binárne vyhľadávanie.

Ešte šikovnejšie je však pomôcť si drobným trikom: predstavme si, že pred poľom (teda na pozícii -1) je darček, ktorý je určite príliš lacný, a za poľom (na pozícii n) darček, ktorý je dostatočne drahý. Nastavíme teda $z = -1$, $k = n$ a rovno spustíme binárne vyhľadávanie.

A kedy teda spoznáme, či nenastal špeciálny prípad? To je jednoduché: úplne na konci. Ak náhodou skončíme s tým, že k ostalo rovné n, sú všetky skutočné darčeky príliš lacné, a teda vypíšeme -1. V opačnom prípade (a to vrátane situácie, kedy ostalo z rovné -1) ukazuje číslo k na pozíciu v poli, kde je prvý dobrý darček. A to je už úplne všetko.

Listing programu:

```
{ ========= toto je funkcia, ktoru ste mali implementovat ======== }

function najdi_darcek(var A: array of longint; n : longint; p : longint) : longint;
var z, k, m : longint;
begin
    { inicializujeme z a k tak, aby ukazovali na urcite zly a urcite dobry darcek }
    z := -1;
    k := n;

    { kym mame viac ako jednu moznost, zmensujeme interval na polovicu }
    while k-z > 1 do begin
        m := (z+k) div 2;
        if A[m] < p then z := m else k := m;
    end;

    { zistime, ci mame riesenie, a ak ano, vratime ho }
    if k=n then najdi_darcek := -1 else najdi_darcek := A[k];
end;

{ ======== a takto by sa dala tato funkcia pouzit v programe ======== }

var pocet_darcekov : longint = 12;
    ceny_darcekov : array[0..11] of longint = (2, 3, 3, 3, 5, 8, 9, 10, 23, 32, 47, 99);
begin
    { vyskusame vo vyssie definovanom poli vyhladavat pre rozne ceny }
    writeln( najdi_darcek( ceny_darcekov, pocet_darcekov,   4 ) );   { vypise 5 }
    writeln( najdi_darcek( ceny_darcekov, pocet_darcekov,   5 ) );   { vypise 5 }
    writeln( najdi_darcek( ceny_darcekov, pocet_darcekov,  17 ) );   { vypise 23 }
    writeln( najdi_darcek( ceny_darcekov, pocet_darcekov, 100 ) );   { vypise -1 }
end.
```

[1] Symbolmi $\lfloor c \rfloor$ a $\lceil c \rceil$ značíme dolnú a hornú celú časť čísla c: $\lfloor c \rfloor$ je najväčšie celé číslo ktoré je najviac c, a $\lceil c \rceil$ je najmenšie celé číslo, ktoré je aspoň c.

[2] Povšimnite si tiež, že krok robíme len vtedy, ak $k - z > 1$. V takomto prípade platí $z < m < k$, a teda určite spravíme aspoň nejaký pokrok aj pre malé intervaly – vždy pozeráme na nové, neznáme políčko a nikdy sa nezacyklíme.

Podúloha b)

Na koľko políčok poľa A sa naše riešenie pozrie? Vždy, keď nejaké políčko poľa A porovnáme s hodnotou p, zmenší sa počet možností, ktoré nám ostávajú, približne na polovicu. Mohli by sme sa teda pýtať otázku: Koľkokrát musíme interval danej dĺžky „rozpoliť“, kým nám ostane len jeden prvok? Pre lepšiu predstavu sa môžeme tú istú otázku opýtať aj opačným smerom: Koľkokrát musíme dĺžku intervalu zdvojnásobiť, aby z 1 narástla až na pôvodnú dĺžku?

Presne na túto otázku nám odpovedajú logaritmy. Logaritmus so základom 2 z čísla a zapíšeme ako $\log_2 a$. Hodnota $\log_2 a$ je také číslo b, že platí $2^b = a$. Napríklad $\log_2 8 = 3$, pretože $2^3 = 8$.

Naše riešenie teda spraví rádovo $\log_2 n$ krokov a v každom z nich sa pozrie na jedno políčko poľa A.

Príklad: Ak by pole A malo milión prvkov, pozrieme sa len na 20 z nich (lebo $\log_2 1\,000\,000 \approx 20$).

Podúloha c)

Dokážeme, že ľubovoľný *deterministický*[3] algoritmus riešiaci zadanú úlohu sa v najhoršom možnom prípade musí pozrieť na aspoň $\lceil \log_2(n+1) \rceil$ políčok poľa A.

Základná myšlienka dôkazu je jednoduchá: Existuje $n+1$ možných výstupov a každým prístupom do poľa A vieme zaručene vylúčiť najviac polovicu z nich, preto potrebný počet prístupov do poľa musí byť v najhoršom možnom prípade aspoň rovný dvojkovému logaritmu čísla $n+1$.

Vo zvyšku tohto vzorového riešenia tento dôkaz rozpíšeme poriadnejšie.

V prvom rade si uvedomme, že keď sa algoritmus pozrie na nejaké políčko x poľa A, jediné rozumné, čo môže spraviť, je porovnať ho s hľadanou hodnotou p. Toto porovnanie nám dá jeden z dvoch možných výsledkov: buď sa dozvieme, že $A[x] < p$, teda dotyčný darček je prilacný, alebo sa dozvieme, že $A[x] \geq p$, teda dotyčný darček je dostatočne drahý.

Na začiatku ešte o poli A nič nevieme. To, čo chceme zistiť, je vlastne počet prvkov v poli A, ktoré sú menšie ako p. (Inými slovami, chceme zistiť polohu z posledného prilacného darčeka.) Je $n+1$ možností: prilacných darčekov môže byť 0, 1, 2, ..., až všetkých n. A ku každej z týchto $n+1$ možností patrí iný výstup algoritmu: v prvých n prípadoch vypíšeme cenu 1., 2., ..., n. darčeka, v poslednom prípade vypíšeme hodnotu -1.

Na začiatku sme teda v situácii, že do úvahy pripadá $n+1$ rôznych možností pre hodnotu z. Tieto možnosti budeme volať *kandidátmi*. Počas behu programu sa tento občas pozrie na niektoré políčko poľa A. V tomto okamihu sa nám množina kandidátov môže zmenšiť.

Príklad 1: Kandidátmi pre z boli čísla 4 až 11. Program sa pozrel na políčko 7 a bolo $A[7] < p$. Od tohto okamihu sú už kandidátmi len čísla od 7 po 11.

Príklad 2: Kandidátmi pre z boli čísla 4 až 11. Program sa pozrel na políčko 13 a samozrejme sa dozvedel, že $A[13] \geq p$. Jeho chyba, že sa tam pozeral, predsa to už mal vedieť. Množina kandidátov sa nezmenila.

No a už sme skoro hotoví. Stačí si len uvedomiť, že pri každom prístupe do poľa A sa množina kandidátov *rozdelí z jedného intervalu na dva*. Ak boli kandidátmi čísla od a po $b-1$ vrátane a opýtali sme sa na c, tak sa stane nasledovné: Ak $A[c] < p$, kandidátmi ostanú čísla c až $b-1$, inak kandidátmi ostanú čísla a až $c-1$. Ak bolo teda pred prístupom do poľa A kandidátov q, po prístupe je ich *v tej pre nás horšej vetve* aspoň $\lceil q/2 \rceil$.

Nech teda 2^k je najväčšia mocnina dvoch ostro menšia ako $n+1$. (Čiže platí $2^k < n+1 \leq 2^{k+1}$.) Tvrdíme, že ak má pole A veľkosť n, neexistuje algoritmus, ktorému vždy stačí pozrieť sa na k políčok poľa A. Totiž na začiatku má pred sebou tento algoritmus $n+1$ kandidátov, a keď mu budeme na otázky odpovedať tak, aby zakaždým nastala tá pre neho horšia alternatíva, aj po k otázkach mu ešte ostanú aspoň dvaja kandidáti, a teda si nebude istý odpoveďou.

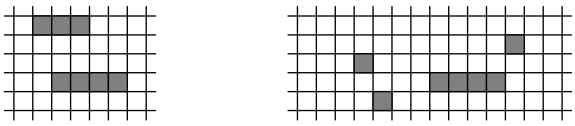
[3] Slovo „deterministický“ znamená, že ďalší krok algoritmu je vždy jednoznačne určený tým, čo sa udialo dovtedy. Inými slovami, je to algoritmus, ktorý sa nikdy nerozhoduje náhodne. Ona by mu tu tá náhoda aj tak nepomohla – keďže sa pozeráme na najhorší možný prípad, tak či tak by museli všetky možné postupy hľadania byť efektívne. A načo si potom náhodne vyberať jeden z nich?

B-II-4 Čierne štvorce sú tu opäť

Podúloha a)

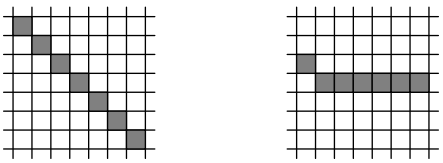
Prvé rozmiestnenie, ktoré sme mali nájsť, malo tvoriť 7 čiernych štvorcov, pre ktoré platí: O 3 minúty ešte bude aspoň jeden štvorec čierny, ale o 4 minúty už budú úplne všetky štvorce v celej rovine biele.

My už vieme vyrobiť rozmiestnenie 4 čiernych štvorcov, ktoré presne 3 minúty vydrží: stačí ich dať do radu vedľa seba. A keď ich máme mať na začiatku namiesto štyroch sedem, stačí ešte pridať tri iné niekde „dostatočne ďaleko“. Na nasledujúcom obrázku vidíme dve rôzne vyhovujúce rozmiestnenia. Treťou možnosťou bolo použiť riešenie podúlohy d) domáceho kola pre $n = 7$ a $k = 4$.



Druhé hľadané rozmiestnenie malo tvoriť 7 čiernych štvorcov, pre ktoré platí: O 6 minút bude práve jeden štvorec v celej rovine čierny. Tento štvorec nesmie ležať na žiadnej z pozícií, kde boli čierne štvorce na začiatku.

Najjednoduchšie je všimnúť si, čo sa stane, ak čierne štvorce tvoria rad idúci „šikmo doprava dodola“. Ten bude postupne postupovať doprava dohora a skracovať sa. Toto riešenie je na nasledujúcom obrázku vľavo. Na obrázku vpravo je iné riešenie tejto podúlohy. Skúste si odsimulovať jeho prvý krok a uvidíte, ako funguje.

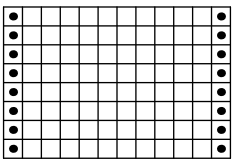


Podúloha b)

Štvorce na obvode Marikinho papiera ostanú navždy biele.

Uvedomte si, že na to, aby sme to dokázali, stačí dokázať, že ak niekedy je celý obvod Marikinho papiera biely, tak aj o minútu neskôr bude celý biely.

Všimnime si najskôr tie štvorce na zvislých okrajoch (teda tie, ktoré sú zvýraznené na nasledujúcom obrázku):



Pre každý z nich platí, že aj on je biely, aj štvorec bezprostredne pod ním je biely. A teda všetky tieto štvorce budú biele aj v nasledujúcom kroku. A rovnakú úvahu môžeme spraviť aj pre štvorce na vodorovných hranách – každý z nich je biely a má naľavo od seba biely štvorec, preto aj v nasledujúcej minúte bude biely.

Podúloha c)

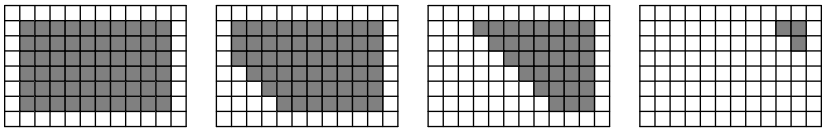
Žiadne začiatočné zafarbenie nevydrží ani len 20 minút, nie to ešte 100. Ukážeme dva rôzne (aj keď podobné) spôsoby, ako toto tvrdenie dokázať.

Prvý dôkaz bude založený na nasledujúcom pozorovaní: Predstavme si dve rôzne štvorcové siete. V prvej označme množinu čiernych štvorcov $\mathcal{A}$, v druhej $\mathcal{B}$. Nech $\mathcal{A} \subseteq \mathcal{B}$, teda každý štvorec, ktorý je čierny v prvej sieti, je čierny aj v druhej.

Zamyslime sa teraz, ako bude situácia v našich dvoch sieťach vyzerať o minútu. V prvej sieti sa množina čiernych štvorcov zmení na $\mathcal{A}'$, v druhej na $\mathcal{B}'$. A bude naďalej platiť vzťah $\mathcal{A}' \subseteq \mathcal{B}'$? Ľahko sa presvedčíme, že áno. Ak je nejaký štvorec s čierny v $\mathcal{A}'$, tým skôr je čierny aj v $\mathcal{B}'$ – všetky čierne štvorce z $\mathcal{A}$, ktoré „hlasovali“ za to, aby s bol čierny v $\mathcal{A}'$, máme aj v $\mathcal{B}$.

Preto vôbec nemusíme skúšať všetkých $2^{6.10}$ možných začiatočných rozložení čiernych štvorcov v Marikinej štvorcovej sieti. Stačí sa nám sústrediť na jediné z nich – to, v ktorom sú na začiatku čierne úplne všetky štvorce. Ak totiž toto rozloženie po niekoľkých krokoch „vyhynie“, vieme, že aj hocijaké iné začiatočné rozloženie by po toľko isto krokoch (a možno aj skôr) už malo len samé biele štvorce.

A pre čierny obdĺžnik už ľahko odsimulujeme zmeny počas nasledujúcich minút. Na nasledujúcom obrázku je znázornené, ako vyzerá po 0, 3, 7 a 13 minútach od začiatku.



Iný možným riešením je priamo si všimnúť (a dokázať), že každé rozmiestnenie čiernych štvorcov sa bude postupne meniť na biele podobne ako náš obdĺžnik – po „uhlopriečkach“, začínajúc v ľavom dolnom rohu.

Aby sme mohli toto tvrdenie presnejšie sformulovať, očíslujme si políčka vo vnútri Marikinho plánu tak ako na nasledujúcom obrázku.

	6	7	8	9	10	11	12	13	14	15	
	5	6	7	8	9	10	11	12	13	14	
	4	5	6	7	8	9	10	11	12	13	
	3	4	5	6	7	8	9	10	11	12	
	2	3	4	5	6	7	8	9	10	11	
	1	2	3	4	5	6	7	8	9	10	

A teraz už ľahko postupne zdôvodňujeme:

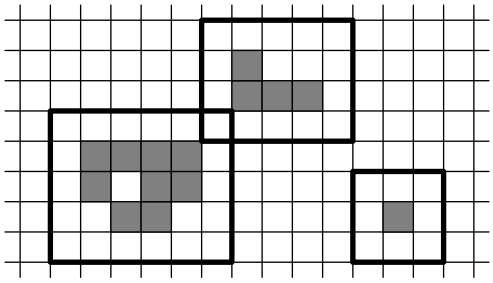
- Po prvej minúte bude štvorec s číslom 1 už navždy biely – aj pod ním, aj naľavo od neho je biely štvorec.
- Po druhej minúte budú aj štvorce s číslom 2 už navždy biele – lebo všetky štvorce, ktoré sú bezprostredne dodola a naľavo od nich sú po prvej minúte navždy biele.
- Po tretej minúte sa k navždy bielym štvorcom pridajú aj štvorce s číslom 3, a tak ďalej.

Teda bez ohľadu na to, ako Marika rozmiestni čierne štvorce, bude o 15 minút celý jej obdĺžnik biely. Vo všeobecnosti, ak máme obdĺžnik rozmerov $r \times c$ (kde $r, c \geq 3$), ktorého okraj je celý biely, tak vieme, že po $r + c - 5$ minútach už bude celé biele aj vnútro.

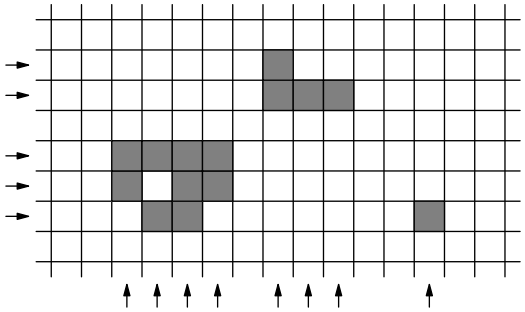
Podúloha d)

Základná myšlienka bude jednoduchá: Pozrieme sa na čierne štvorce a zvolíme nejaký obdĺžnik, ktorý ich obsahuje a má celý okraj biely. Z jeho rozmerov potom vieme, rovnako ako v predchádzajúcej podúlohe, vypočítať, po koľkých krokoch už bude zaručene celý biely.

Poriadny dôkaz ale treba predsa len robiť trochu poriadnejšie. To, na čo si potrebujeme dať pozor, je situácia, kedy sú čierne štvorce príliš „roztrúsené“ – vtedy totiž musíme namiesto jedného veľkého obdĺžnika použiť viacero malých. Na nasledujúcom obrázku je príklad toho, ako sme takto „uzavreli“ 14 čiernych štvorcov do troch obdĺžnikov. Pre situáciu na obrázku vieme (podľa rozmerov najväčšieho obdĺžnika) zaručiť, že o nanajvýš 6 sekúnd už budú všetky štvorce biele.

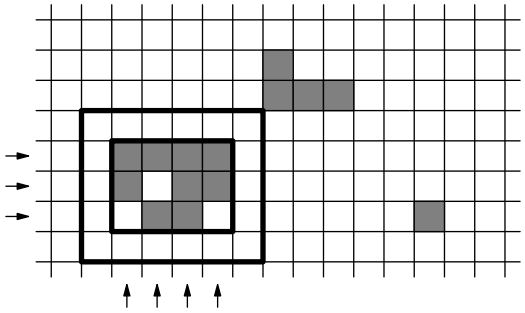


Ako takéto uzavretie do obdĺžnikov vyrobiť systematicky a zaručiť, že budú všetky z nich malé? Môžeme napríklad začať tým, že si vyznačíme riadky a stĺpce, v ktorých leží aspoň jeden čierny štvorec. (Ak teda máme 47 čiernych štvorcov, tak dokopy vyznačíme najviac 47 riadkov a najviac 47 stĺpcov.)

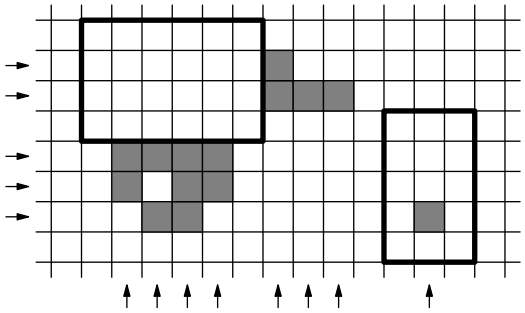


Vyznačené riadky aj stĺpce sa nám rozpadnú na niekoľko súvislých skupín. Napríklad na obrázku nad týmto textom sú dve skupiny vyznačených riadkov a tri skupiny vyznačených stĺpcov.

No a keď si vyberieme jednu skupinu vyznačených riadkov a jednu skupinu vyznačených stĺpcov, dostaneme jeden z obdĺžnikov:



Tu sú ďalšie dva zo šiestich obdĺžnikov, ktoré našim postupom vznikli. Všimnite si, že môžeme dostať aj prázdne alebo zbytočne veľké obdĺžniky, to nám ale vôbec neprekáža.



A kedy že to prefarbovanie vlastne skončí? Každý obdĺžnik sa bude prefarbovať nezávisle od zvyšku roviny, preto môžeme uvažovať každý zvlášť. Najhorší možný prípad je, že budeme mať presne jeden obdĺžnik a ten bude mať rozmery 49×49. No a podľa vzťahu, ktorý sme si odvodili v podúlohe c), o 93 minút bude už aj takto veľký obdĺžnik celý biely. Jedným možným riešením podúlohy d) je teda číslo $k = 93$.

(Toto riešenie nie je najlepšie možné. Vedeli by ste ho zlepšiť?)

DVADSIATY SIEDMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Autori úloh: Michal Forišek, Ján Hozza, Mária Mrocková
Recenzent: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2012