Korešpondenčný seminár z programovania
XXXII. ročník, 2014/15
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola letnej časti

Baklažán

1. Zerg Bot 2

(max. 0 b za popis, 15 b za program)

Tento vzorák bude používať mnohé pojmy a techniky vysvetlené vo vzoráku k predošlej sérii, preto pred jeho čítaním odporúčam, aby ste si prečítali vzorové riešenia k úlohe Z1 z predošlej série KSP (ak ste ich náhodou ešte nečítali).

1 - Tam a späť

Na prejdenie levelu stačí, aby robot chodil stále rovno a vždy, keď príde na rozsvietený prepínač, prepínač vypol a otočil sa. Napíšeme si teda funkciu, ktorá urobí jeden krok a v prípade potreby vypne prepínač a otočí robota. Túto funkciu potom robota necháme vykonávať dookola v nekonečnej rekurzii.

Narazíme však na drobný problém. Robot počas svojho pohybu chodí aj po už vypnutých prepínačoch, preto ak po vstupe na rozsvietený prepínač najprv prepínač vypneme, robot nemá ako zistiť, že sa má otáčať. To môžeme vyriešiť napríklad tak, že si napíšeme pomocnú funkciu, ktorá prepne prepínač a otočí robota. Túto funkciu potom zavoláme, keď vstúpime na rozsvietený prepínač. Tiež by fungovalo, keby sme robota po vstupe na rozsvietený prepínač najskôr otočili a až potom prepínač vypli.

Kód:

```
funkcia0

aha funkcia0:
krok
funkcia1 ak svieti
funkcia0

aha funkcia1:
prepni
dolava
dolava
```

2 - Tam, kde sa skrýval bazilisk

V tomto leveli sa najprv bolo treba trochu pohrať s prepínačmi a zistiť, ako fungujú. Keďže väčšina chodbičiek má dĺžku 3, určite sa nám zíde funkcia, ktorá urobí 3 kroky:

```
aha funkcia0:
krok
krok
krok
```

Po troche experimentovania nájdeme spôsob, ako level prejsť. Jednou z možností je obísť celú komnatu v smere hodinových ručičiek začnúc v juhovýchodnom rohu a cestou rozsvietiť všetky prepínače okrem prepínača v severovýchodnom rohu. Môžeme si všimnúť, že tri steny komnaty vyzerajú úplne rovnako, preto sa nám oplatí napísať si funkciu, ktorá začínajúc v rohu prepne prepínač v danom rohu, prejde popri jednej stene (cestou prepne aj prepínač "vo výklenku") a skončí v ďalšom rohu. Aby sme túto funkciu mohli volať bezprostredne viackrát po sebe, v koncovom rohu sa ešte otočíme doprava:

```
aha funkcia1:
prepni
funkcia0
dolava
```

```
krok
prepni
dolava
dolava
krok
dolava
funkcia0
doprava
```

S pomocou tejto funkcie potom ľahko napíšeme celý program:

Hlavný kód:

```
funkcia0
doprava
funkcia0
doprava
funkcia1
funkcia1
funkcia1
funkcia0
dolava
funkcia0
krok
```

3 - Komprimácia

Najprv si spočítame, že rovné úseky majú postupne dĺžky 9, 4, 7, 13, 4, 6, 15, 6 a 12 krokov. Keďže medzi týmito dĺžkami je dosť veľa násobkov trojky, zíde sa nám funkcia, ktorá urobí 3 kroky. Tiež sa nám tam dvakrát opakuje štvorka, preto si definujeme aj funkciu na urobenie 4 krokov (samozrejme s využitím funkcie pre 3 kroky). S pomocou týchto dvoch funkcií budeme vedieť urobiť 7 krokov na dva príkazy. Keďže máme dva úseky dlhé 6, oplatí sa nám napísať si aj funkciu na 6 krokov. Nakoniec sa nám zíde aj funkcia na 9 krokov – nielenže máme úsek dlhý 9, ale skombinovaním deväť- a štvorkrokovej funkcie vieme na dva príkazy urobiť 13 krokov a skombinovaním deväť- a šesťkrokovej vieme urobiť 15 krokov. Pri definícii každej z týchto funkcií môžeme, samozrejme, využívať tie predchádzajúce. Naše funkcie teda vyzerajú takto:

```
aha funkcia3:
krok
krok
krok

aha funkcia4:
funkcia3
krok

aha funkcia6:
funkcia3
funkcia3

aha funkcia9:
funkcia6
funkcia3
```

S pomocou takýchto funkcií vieme ľahko napísať program, ktorý prejde level a spolu s definíciami funkcií má presne 30 príkazov, teda sa akurát zmestí do limitu. Rovné úseky z našich funkcií poskladáme ako 9, 4, 4+3, 9+4, 4, 6, 9+6, 6 a 9+3. Toto riešenie sa však dá ešte zlepšiť. Môžeme si totiž všimnúť, že po všetkých úsekoch, kde sme použili `funkciu4`, nasledovala zákruta doprava. Preto môžeme príkaz `doprava` pridať na koniec `funkcie4` a túto funkciu vždy volať až na konci úseku. Podobne po každom úseku obsahujúcom `funkciu6` nasleduje zákruta doľava, preto môžeme príkaz `dolava` pridať na koniec `funkcie6`. Tým si ale pokazíme `funkciu9` (keďže tá vo svojej definícii používa `funkciu6`), preto ju musíme prepísať, napríklad takto:

```
aha funkcia9:
funkcia3
funkcia3
funkcia3
```

S pomocou takýchto funkcií potom vieme napísať program, ktorý má spolu s definíciami iba 26 príkazov:

Hlavný kód:

```
funkcia9
dolava

funkcia4

funkcia3
funkcia4

funkcia9
funkcia4

funkcia4

funkcia6

funkcia9
funkcia6

funkcia6

funkcia9
funkcia3
```

4 - Svetielka

Potrebujeme rozsvietiť 5 prepínačov na východnom konci chodby a zhasnúť 5 prepínačov na západnom konci chodby. Celé to môžeme urobiť napríklad takto:

1. Najprv prepneme prvých 5 prepínačov.
2. Potom pôjdeme rovno, až kým nenarazíme na stenu, pričom cestou budeme vypínať všetky svietiace prepínače.
3. Pri stene sa otočíme.
4. Pôjdeme až do konca levelu iba rovno.

Prvý bod urobíme veľmi jednoducho: napíšeme si funkciu, ktorá prepne prepínač a urobí krok, túto funkciu potom zavoláme päťkrát.

```
aha funkcia0:
prepni
krok
```

Na druhú časť si napíšeme takúto funkciu:

```
aha funkcia1:
krok
prepni ak svieti
funkcia1 ak nie je stena vpredu
```

Kým pred robotom nebude stena, táto funkcia sa bude rekurzívne volať dookola. Po príchode k stene sa volať prestane a všetky jej zavolania postupne skončia.

Tretia časť je jednoduchá a na štvrtú časť nám stačí funkcia, ktorá bude v nekonečnej rekurzii robiť kroky:

```
aha funkcia2:
krok
funkcia2
```

Hlavný program teda bude vyzerať nasledovne:

Hlavný kód:

```
funkcia0
funkcia0
funkcia0
funkcia0
funkcia0
funkcia1
dolava
dolava
funkcia2
```

5 - Tajná kombinácia

Keďže o tajnej kombinácii nič nevieme, nezostáva nám nič iné, ako začať postupne skúšať všetky možné kombinácie. Nemusíme to však skúšať ručne. Namiesto toho si môžeme napísať program, ktorý na hracom pláne postupne vytvorí všetky možné kombinácie, tento program spustiť a potom sa už len pozerať, pri ktorej kombinácii sa jet torch vypne.

Napíšme si `funkciu0`, ktorá za predpokladu, že je robot otočený smerom na východ, vytvorí na prepínačoch, ktoré sú medzi robotom a východným koncom chodby (vrátane prepínača, na ktorom robotom práve stojí) postupne všetky možné kombinácie, pričom na konci nechá vypínače v rovnakom stave ako boli na začiatku a robot bude stáť na rovnakom políčku ako na začiatku, ale otočený na západ. Tento popis sa vám môže zdať trochu krkolomný, ale neskôr sa nám zíde. Pri programovaní tejto funkcie využijeme princíp vysvetľovania významu slova pomocou toho istého slova.

Ako by takáto funkcia mala vyzerať?

- V prípade, že je robot na začiatku volania funkcie na východnom konci chodby, je to jednoduché. Máme vytvoriť všetky možné kombinácie na prepínači, na ktorom robot práve stojí. Takéto kombinácie sú iba dve – buď je prepínač vypnutý alebo je zapnutý. V jednom z týchto stavov je na začiatku a do druhého ho jednoducho prepneme. Chceme, aby robot skončil otočený smerom na západ a aby prepínače boli v pôvodnom stave, preto robota ešte otočíme a prepínač prepneme naspäť.

```
prepni
dolava
dolava
prepni
```

- Prípad, keď robot na začiatku volania funkcie nie je na východnom konci chodby, je trochu zložitejší, keďže sa musíme postarať o väčší úsek prepínačov a teda musíme vytvoriť viac kombinácií. Počet prepínačov navyše dopredu nevieme. Všetky kombinácie však môžeme rozdeliť na dve skupiny: tie, pri ktorých je prepínač, na ktorom robot práve stojí, zhasnutý a tie, pri ktorých je rozsvietený. Najprv vytvoríme všetky kombinácie z prvej skupiny: posunieme robota o jeden krok na východ a zavoláme `funkciu0`. Tým sa na všetkých prepínačoch východne od začiatočného vytvoria všetky možné kombinácie, teda sme vytvorili všetky kombinácie, pri ktorých je začiatočný prepínač vypnutý. Následne sa vrátime na políčko, kde sme začínali, rozsvietime ho a podobným spôsobom vytvoríme všetky kombinácie, pri ktorých je toto políčko rozsvietené. Nakoniec sa vrátime na začiatočné políčko a opäť ho zhasneme, aby boli na konci volania funkcie políčka v pôvodnom stave.

```
krok
funkcia0
krok
prepni
dolava
dolava
krok
```

```
funkcia0
krok
prepni
```

Ešte nejako potrebujeme docieliť, aby sa vždy vykonala tá správna z týchto dvoch možností. To vieme urobiť napríklad tak, že každú z týchto možností dáme do inej pomocnej funkcie a vo `funkcii0` sa iba rozhodneme, ktorú z týchto dvoch funkcií zavoláme. Pritom si, samozrejme, musíme dať pozor, aby sa nám nestalo, že po zavolaní prvej (a s tým súvisiacej zmene situácie) sa zavolá aj druhá.

Kód:

```
funkcia0

aha funkcia0:
funkcia1 ak je stena vpredu
funkcia2 ak je stena vlavo

aha funkcia1:
prepni
dolava
dolava
prepni

aha funkcia2:
krok
funkcia0
krok
prepni
dolava
dolava
krok
funkcia0
krok
prepni
```

Po tom, čo nájdeme správnu kombináciu, je už riešenie jednoduché:

Kód:

```
krok
prepni
krok
prepni
krok
krok
prepni
krok
prepni
krok
doprava
krok
krok
```

1 - Svietiaca cesta

Napíšeme si funkciu, ktorá urobí krok správnym smerom a potom zavolá sama seba. Ako urobiť krok správnym smerom? Jednoducho vyskúšame všetky tri možné smery. Najprv urobíme krok dopredu a skontrolujeme, či sme na svietiacom prepínači. Ak áno, potom to bol krok správnym smerom a môžeme znovu zavolať našu

funkciu. V opačnom prípade sa vrátime naspäť a skúsime nejaký iný smer, napríklad doľava (vzhľadom na smer, ktorým sme boli otočení na začiatku volania funkcie). Ak ani tým smerom nebude svietiaci prepínač, vrátime sa a pôjdeme posledným možným smerom, teda doprava. Keďže používame nekonečnú rekurziu, funkcia sa vždy vykoná iba po miesto, kde prvý raz zavolá sama seba, teda sa nám nemôže stať, že by po prvom zavolaní urobila ešte nejaké nežiadúce príkazy navyše. Keďže sa na viacerých miestach programu budeme chcieť vrátiť na políčko, odkiaľ sme prišli, môžeme si na to napísať pomocnú funkciu.

Kód:

```
funkcia0

aha funkcia0:
krok
funkcia0 ak svieti
funkcia1
doprava
krok
funkcia0 ak svieti
funkcia1
krok
funkcia0

aha funkcia1:
dolava
dolava
krok
```

2 - Rozsvecovanie

Najjednoduchší spôsob ako zaručene nájsť všetky stĺpy, je prehľadať celú mapu. Skúsme teda najprv napísať program, ktorý by po riadkoch prešiel po celom plániku (podobne ako v leveli 4-Šachovnica z úlohy Prask 1.4 z predošlej série), ak by tam neboli stĺpy. Robotovu cestu môžeme rozdeliť na 4 druhy úsekov: úseky, keď ide na východ, úseky, keď ide na západ, otáčanie pri východnom okraji miestnosti a otáčanie pri západnom okraji miestnosti. Pre každý z týchto druhov si napíšeme jednu funkciu a tieto štyri funkcie vhodným spôsobom necháme volať sa navzájom v nekonečnej rekurzii:

```
funkcia0

aha funkcia0:
krok
funkcia0 ak nie je stena vpredu
funkcia2

aha funkcia1:
krok
funkcia1 ak nie je stena vpredu
funkcia3

aha funkcia2:
doprava
krok
doprava
funkcia1

aha funkcia3:
dolava
krok
dolava
funkcia0
```

`funkcia0` slúži na cestu na východ: táto funkcia bude hýbať robotom dopredu a volať sama seba dookola, až kým robot nepríde k stene. Potom zavolá `funkciu2`, ktorá robota otočí (a posunie do ďalšieho riadku) a následne zavolá `funkciu1`, ktorá slúži na cestu na západ. `funkcia1` podobným spôsobom dovedie robota k západnému okraju miestnosti a zavolá `funkciu3`, ktorá robota otočí (a posunie o jedno políčko na juh) a zavolá opäť `funkciu0`. Takto sa budú tieto štyri funkcie volať dookola, až kým robot nepríde na spodok mapy.

To je síce všetko pekné, ale na mape sú aj stĺpy, okolo ktorých potrebujeme rozsvietiť prepínače. Najprv, samozrejme, budeme potrebovať vedieť rozoznať stĺpy od okrajov miestnosti. Na to môžeme využiť, že stĺpy sa, na rozdiel od okrajov miestnosti, dajú obísť. Napíšme si teda funkciu, ktorá obíde jeden stĺp:

```
aha funkcia4:
doprava
krok
dolava
krok
krok
dolava
krok
doprava
```

Peter

2. Zabezpečenie

(max. 6 b za popis, 4 b za program)

Vo všeobecných šifrách potrebujeme na dešifrovanie poznať šifrovací algoritmus a kľúč na dešifrovanie. Šifrovací algoritmus už poznáme – je popísaný v zadaní. Pravdepodobne nebude problém celý proces obrátiť a dešifrovať text. Avšak na to potrebujeme kľúč – v našom prípade číslo udávajúce posun v abecede. Tento kľúč zatiaľ nepoznáme, preto ho musíme zistiť z jednotlivých správ.

Na začiatok, keď nevieme prísť na nejaký prefíkaný spôsob, ako odhaliť kľúč, môžeme skúšať všetky možné kľúče, kým použitie jedného nevráti zmysluplný dešifrovaný text. V našom jednoduchom príklade spoznáme "zmysluplný" text tak, že na jeho začiatku sa nachádza *V MENE KSP*. Môžeme teda v cykle postupne prechádzať všetky možné posuny. Pre každý posun "zmenšíme" každé písmeno o 1, a v prípade, že by sme sa mali dostať pod A, tak pokračujeme od Z. Po takomto pokuse o dešifrovanie overíme, či text začína v *V MENE KSP*.

Predošlé riešenie má časovú zložitosť $O(n \cdot k)$, kde n je maximálna dĺžka textu a k je počet možných kľúčov. V našom prípade riešenie nie je až také zlé, keďže texty sú pomerne krátke a písmen v abecede je len 26.

Keby však správa mohla pozostávať zo všetkých ASCII znakov, alebo nebodaj z UTF-8 znakov a bola by veľká niekoľko gigabajtov, bol by to už trochu problém. Existuje však aj rýchlejšie a jednoduchšie riešenie.

Vieme, že prvý znak textu (označme ho e) musel vzniknúť zašifrovaním V (lebo to je prvé písmeno *V MENE KSP*). Kľúč môžeme jednoducho vypočítať ako hodnotu $e - V$.

V niektorých programovacích jazykoch ako napríklad C++ môžeme pracovať so znakmi priamo ako s číslami, napríklad `'D'-'A'` bude číslo 3. V iných jazykoch musíme najprv konvertovať znaky na čísla. `ord('D')-ord('A')` bude mať hodnotu 3.

Tiež ak nám vadí, keď vyjde záporná hodnota ($F - V$ je -16), môžeme k tomuto číslu pričítať hodnotu 26.

Akonáhle vypočítame k, stačí dešifrovať text podobne ako v predošlom riešení, a keďže už vieme správny kľúč, stačí odčítavať od každého znaku iba raz. Takto dostaneme časovú zložitosť $O(n)$. Pamäťovú zložitosť vieme mať dokonca $O(1)$, keby sme vstup načítavali postupne znak po znaku a rovno vypisovali výsledok, ale pre jednoduchosť budeme načítavať celý riadok do pamäte naraz.

Listing programu (C++)

```
#include <iostream>
#include <string>
using namespace std;

int main(){
        int n; //počet riadkov vstupu
        int k; //kľúč
        string s; //string do ktorého budem načítavať
        cin >> n;
        /* Ako si niekto všimol, kombinovaním cin a getline vznikajú problémy.
         * Nasledujúci getline načítava ešte z prvého riadku vstupu. */
        getline(cin,s);
        for (int riadok = 0; riadok < n; riadok++){
                getline(cin,s); //Načítam celý jeden riadok vstupu.
                k = (s[0] - 'V' + 26)%26; //Spočítam kľúč podľa prvého znaku stringu (pôvodne 'V').
                for (int i = 0; i < s.size(); i++){
                        if (s[i] == ' ') continue; //Medzery nedešifrujem.
                        s[i] = (s[i] - 'A' - k + 26)%26 + 'A'; //Dešifrujem každé písmeno.
                        //+'A' a -'A' používam na prevod medzi ASCII a hodnotou znaku od 0 po 25
                }
```

```
                cout << s << endl; //Dešifrovaný text vypíšem.
        }
}
```

Listing programu (Python)

```
t = int(input())
for i in range(t):
    line = input()
    kluc = (ord(line[0])-ord('V')) % 26
    line = [x if x == '␣' else chr((ord(x)-ord('A')-kluc) % 26 + ord('A')) for x in line]
    print(''.join(line))
```

Kubo

3. Zatúlané cukríky

(max. 0 b za popis, 15 b za program)

Pozrime sa najprv na trochu jednoduchšiu úlohu, v ktorej nebudeme musieť vypisovať celú Kukovu cestu, ale len počet cukríkov, ktoré nazbieral.

Rekurzia

Asi najjednoduchšie funkčné riešenie, aké sa dalo vymyslieť, je rekurzia.

Cheme zistiť, koľko najviac cukríkov mohol Kuko zobrať po ceste z ľavého horného rohu do pravého dolného rohu mapy. Dostať sa na pravé dolné políčko mohol buď zhora alebo zľava. Rekurzívne si zistíme, koľko najviac cukríkov vedel zozbierať na ceste zo začiatku do jedného z týchto políčok (túto otázku sa pýtam dvakrát, postupne pre obe políčka) a vyberieme si lepšiu možnosť. Tieto kratšie cesty vyrátame rovnakým spôsobom, buď sa na posledné políčko na nej prišiel zľava alebo zhora. Jedinou výnimkou sú políčka na ľavom a hornom okraji mapy, kde mám vždy len jednu možnosť, ako som sa mohol na ne dostať a ľavé horné políčko, kde nie je ani jedna možnosť.

Aká je časová zložitosť? Prvé volanie funkcie sa zavolá dvakrát, každé z týchto volaní sa však tiež zavolá dvakrát, čo je dokopy už štyrikrát, a čo nevidieť tu máme exponenciálnu časovú zložitosť. To nám zbehne tak na vstupoch do veľkosti 10, ale na ostatné potrebujeme niečo rýchlejšie.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <cmath>
using namespace std;

vector<vector<int>> mapa;

int zrataj(int x, int y){
        if (x<0 || y<0) return 0; //ak sme mimo mapy, vratime nulu

        int vlavo = zrataj(x, y-1); //vzdy sa rekurzivne zavolame na predchadzajuce policka
        int hore = zrataj(x-1, y);

        return mapa[x][y] + max(vlavo, hore); //vratime hodnotu pre toto policko
}

int main(){
        int n, m;
        cin>>n>>m;
        mapa.resize(n, vector<int> (m)); //nastavime velkost pola a nacitame mapu
        for (int i=0; i<n; i++){
                for (int j=0; j<m; j++){
                        cin>>mapa[i][j];
                }
        }
        int vysledok = zrataj(n-1, m-1); //spytame sa na policko, ktore nas zaujima
        cout<<vysledok<<endl;
}
```

Rekurzia s memoizáciou

Riešenie obyčajnou rekurziou je pomalé preto, lebo veľakrát hľadáme cestu na to isté políčko. Algoritmus preto upravíme: namiesto toho, aby sme vždy danú cestu počítali nanovo, si ju spočítame len raz, výsledok uložíme do pamäte a pri ďalších otázkach vrátime rovno túto už vypočítanú hodnotu.

Vďaka tomu, že cestu na každé políčko prepočítavame najviac raz a pýtame sa ňu najviac dvakrát[1], sa časová zložitosť rekurzie zlepšila na počet políčok, teda $O(nm)$. Táto zložitosť je naviac najlepšia možná, lebo musíme načítať hodnotu všetkých políčok, čo nám zaberie čas $O(nm)$.

[1] raz keď počítame cestu na políčko pod ním a raz na cestu na políčko vpravo

Vypisovanie cesty

Vypisovanie cesty je pri vyššie uvedených algoritmoch už len čerešničkou na torte a dá sa doprogramovať veľmi jednoducho. Pri všetkých sme sa totiž pre danú cestu na políčko pýtali, či je lepšie ísť doňho zvrchu alebo zľava. Zapamätáme si teda, ktorá z týchto možností bola výhodnejšia, napríklad do dvojrozmerného poľa znakov. Z neho budeme na konci vedieť priamo vyskladať cestu – stačí, ak pôjdeme odzadu.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <cmath>
using namespace std;

vector<vector<int>> mapa;
vector<vector<int>> memo;
vector<vector<char>>path;

int zrataj(int x, int y){
    if (x<0 || y<0) return 0; //ak sme mimo mapy, vratime nulu

    if (memo[x][y] == -1) {     //ak sme si toto policko nevyratali, tak ideme na to
        int vlavo = zrataj(x, y-1);
        int hore = zrataj(x-1, y);

        memo[x][y] =  mapa[x][y] + max(vlavo, hore); //zapiseme vysledok pre policko

        if(vlavo > hore){ //zapiseme si odkial sa oplatilo prist, aby sme vedeli spravit cestu
            path[x][y] = 'R';
        } else {
            path[x][y] = 'D';
        }
    }

    return memo[x][y]; //vratime vysledok pre policko, ci sme ho zratali teraz alebo uz kedysi predtym
}

int main(){
    int n, m;
    cin>>n>>m;

    mapa.resize(n, vector<int> (m)); //nacitame mapu a poinicializujeme polia
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
                cin>>mapa[i][j];

    memo.resize(n, vector<int> (m, -1));
    path.resize(n, vector<char> (m));

    zrataj(n-1, m-1);

    cout<<memo[n-1][m-1]<<endl; //vypiseme posledne policko, na ktorom je odpoved

    string cesta;

    int i = n-1;
    int j = m-1;
    while(i != 0 || j != 0){ //vyskladame cestu od konca
        cesta += path[i][j];
        if (path[i][j] == 'R')
            j--;
        else
            i--;
    }

    reverse(cesta.begin(), cesta.end()); //cestu sme vyskladali od konca, tak ju prevratime
    cout<<cesta<<endl;
}
```

Dynamické programovanie

Táto úloha sa dala riešiť aj principiálne inak, než rekurziou, a to pomocou dynamického programovania. Pri rekurzii sme sa snažili zrátať rovno výsledok a postupne si dopočítavať zvyšné informácie, ako sme ich potrebovali. Dynamické programovanie sa na problém pozerá z opačného konca – z údajov, ktoré už poznáme vyráta ďalšie, až kým sa nedopracuje k výsledku. Postupne preto ráta najväčší počet cukríkov, ktoré vieme zozbierať na ceste na dané políčko, ale začína rátať v ľavom hornom rohu a od neho ide postupne doprava a dole.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <cmath>
```

```cpp
using namespace std;

int main(){
    int n, m;
    cin>>n>>m;

    vector<vector<int>> mapa;
    vector<vector<int>> memo;
    vector<vector<char>>path;

    mapa.resize(n, vector<int> (m)); //nacitame mapu a poinicializujeme polia (rovnako ako v rekurzii)
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
            cin>>mapa[i][j];

    memo.resize(n, vector<int> (m, -1));
    path.resize(n, vector<char> (m));

    //zacneme ratat

    for(int i=0; i<n; i++){ //prejdeme postupne od laveho horneho rohu po riadkoch
            for(int j=0; j<m; j++){
                    int vlavo, hore;
                    vlavo = hore = 0;
                    if (i != 0) hore = memo[i-1][j]; //kontrolujeme, či nie sme na okraji mapy
                    if (j != 0) vlavo = memo[i][j-1]; //kedze ideme po mape postupne
                    // tak vieme ze policka vlavo a hore mame uz zratane

                    memo[i][j] = mapa[i][j] + max(vlavo, hore);

                    if(vlavo > hore){
                            path[i][j] = 'R';
                    } else {
                            path[i][j] = 'D';
                    }
            }
    }

    string cesta;

    int i = n-1;
    int j = m-1;
    while(i != 0 || j != 0){ //vyskladame cestu od konca, rovnako ako pri rekurzii
            cesta += path[i][j];
            if (path[i][j] == 'R')
                    j--;
            else
                    i--;
    }

    reverse(cesta.begin(), cesta.end());
    cout<<cesta<<endl;
}
```

Jaro

4. Zaspatý krtko

(max. 7 b za popis, 8 b za program)

V tejto úlohe sa dalo spraviť celkom priamočiare riešenie, ktoré by potrebovalo čas $O(n^2)$, jednoducho tak, že vyskúšame súčet čísel v každom súvislom úseku. Napríklad pre každý začiatok úseku skúsime postupne pričítavať prvky sprava a popritom všetkom si budeme udržiavať, aký najvyšší súčet sme doposiaľ videli, kde začínal úsek s týmto súčtom a aký bol dlhý.

Takéto riešenie síce nedostane plný počet bodov, ale je lepšie mať aspoň niečo.

Listing programu (Pascal)

```pascal
var     i, j, sum, n                    : int64;
        bestsum, bestbegin, bestlen     : int64;
        cisla                           : array [1..1000009] of int64;

begin

    //Nacitame a nastavime vsetko potrebne
    read(n);
    bestsum := 0;
    bestbegin := 1;
    bestlen := 0;
    for i:=1 to n do read(cisla [i]);

    //Vonkajsim cyklom si urcime zaciatok
    for i:=1 to n do begin

        //Novy zaciatok znamena sucet 0
        sum := 0;

        //Vnutornym cyklom prejdeme konce
        for j:=i to n do begin

            //Sucet po tento koniec je rovny predoslemu suctu
            //navysenemu o cislo na aktualnej pozicii
```

```
            sum := sum + cisla [j];

            //ak mame lepsi vysledok, poznamename si ho
            if sum > bestsum then begin
                bestsum := sum;
                bestbegin := i;
                bestlen := j - i + 1;
            end;
        end;
    end;

    writeln(bestbegin, ' ', bestlen);
end.
```

Jeden prístup

Jedna spásonosná myšlienka, ktorá vám mohla napadnúť, je pozrieť sa na úlohu ako na úseky kladných[2] a záporných čísel. Správny úsek bude začínať na začiatku nejakého kladného úseku a končiť na konci nejakého kladného úseku.

Ďalšie úvahy teda mohli sledovať, kedy sa nám oplatí prepojiť dva kladné úseky, aj keď sú oddelené úsekom záporným. Netreba však vymýšľať ťažké podmienky, môžme si napríklad pre každé číslo zistiť, aký najlepší výsledok vieme dosiahnuť, keď ním budeme končiť, s tým, že povolíme, aby tento výsledok bol nula, pokiaľ každý, ktorý končí našim číslom, bude záporný [3].

Tak sa nám úloha totiž zjednoduší, ak chceme výsledok končiaci nejakým číslom, bude to buď nula, alebo súčet tohto čísla s výsledkom pre predošlé číslo. Nakoniec si už stačí len všimnúť, že nepotrebujeme všetky medzivýsledky ukladať, ale stačí nám udržiavať si akési lokálne a akési globálne najlepšie riešenie.

Dostali sme sa teda k optimálnemu riešeniu s časovou zložitosťou $O(n)$ a pamäťou $O(1)$.

Listing programu (Pascal)

```
var i, n, locsum, globsum, locbegin, globbegin, globnum, curr   : int64;

begin
    //nacitame n a poinicializujeme vsetky premenne
    //na rozumne hodnoty, znamenajuce prazdny usek
    read(n);
    globsum := 0;
    globbegin := 1;
    globnum := 0;
    locbegin := 1;
    locsum := 0;

    //tu spracujeme vsetky cisla
    for i:=1 to n do begin
        read(curr);

        //k lokalnemu najlepsiemu vysledku priratame aktualne cislo
        locsum := locsum + curr;

        //ak sme dostali doteraz najlepsi vysledok
        if (locsum > globsum) then begin
            globsum := locsum;
            globbegin := locbegin;
            globnum := i - globbegin + 1;
        end

        //a ak mame zaporny lokalny vysledok, oplati sa nam namiesto
        //neho zacinat usek na nasledujucom cisle
        else if locsum < 0 then begin
            locsum := 0;
            locbegin := i+1;
        end;
    end;

    writeln(globbegin, ' ', globnum);
end.
```

Druhý prístup

Napriek tomu si ešte rozoberieme druhé, možno menej intuitívne, ale za to lepšie zovšeobecniteľné riešenie. Pozrieme sa, čo nám pomôže, ak si pre každú pozíciu vyrátame súčet čísel od začiatku po túto pozíciu. Takéto čísla budeme volať prefixové súčty. Keď sa nad tým zamyslíme poriadnejšie, súčet každého súvislého úseku vieme popísať rozdielom dvoch prefixových súčtov.

Ak by sme takto chceli spočítať najväčší súčet, ktorý končí na nejakej nami určenej pozícii, stačí nám zistiť, aký najmenší prefixový súčet sme videli naľavo od seba, lebo najväčší výsledok dostaneme samozrejme, pokiaľ od nášho nemenného čísla odčítame čo najmenšie číslo.

[2] resp. nezáporných

[3] Vtedy predsa môžme zobrať prázdny úsek.

Čo teda spravíme: Prejdeme pole zľava doprava, budeme si počítať prefixové súčty a pamätať si doteraz najmenší videný prefixový súčet a doteraz najlepší videný výsledok. Ten prípadne upravíme, ak je rozdiel nášho aktuálneho prefixového súčtu a najmenšieho videného súčtu lepší, ako najlepší doterajší výsledok. Zložitosti budú rovnaké ako v predošlom prístupe.

Listing programu (Pascal)

```
var i, n, minprefix, minend, bestsum, bestbegin, bestnum    : int64;
    curr, currprefix                                        : int64;

begin
    read(n);
    bestsum := 0;       //najlepsi doteraz videny sucet useku
    bestbegin := 1;     //jeho zaciatok
    bestnum := 0;       //a pocet cisel v nom
    minprefix := 0;     //najmensi prefixovy sucet
    minend := 0;        //a kde konci
    currprefix := 0;    //aktualny prefixovy sucet

    //tu si spracujeme vsetky cisla
    for i:= 1 to n do begin
        read(curr);

        //upravime aktualny prefixovy sucet
        currprefix := currprefix + curr;

        //ak sme nasli novy najlepsi vysledok
        if (currprefix - minprefix) > bestsum then begin
            bestsum := currprefix - minprefix;
            bestbegin := minend + 1;
            bestnum := i - minend;
        end;

        //aktualizujeme najlepsi (najmensi) prefixovy sucet
        //rozmyslite si, ze ak by tato podmienka isla prva, vysledok
        //by bol stale spravny
        if currprefix < minprefix then begin
            minprefix := currprefix;
            minend := i;
        end;
    end;

    writeln(bestbegin, ' ', bestnum);
end.
```

mišof

5. Otravné farmárčenie

(max. 3 b za popis, 15 b za program)

Už z prvého pohľadu na túto úlohu by malo byť jasné, že si môžeme nechať zájsť chuť na akékoľvek exaktné optimálne riešenie. Spôsob, akým vyrobíme výsledný obrázok z jeho komprimovanej podoby, nemá žiadne dobré vlastnosti, ktoré by sme vedeli využiť.

Čo nám teda ostáva? Prezerať nejaké možné riešenia a vybrať si najlepšie, ktoré sa nám podarí nájsť.

Hm, ale ani to nevyzerá veľmi nádejne. Keď máme obrázok rozmerov 500×500 a v ňom môžeme zvoliť 1000 referenčných bodov, existuje približne $1000^{250\,000}$ možností ako obrázok skomprimovať. Skúšanie všetkých možností hrubou silou teda tiež nevyzerá reálne. Vyzerá to, že predsa len budeme musieť aj trochu myslieť.

Jedna z vecí, ktoré si môžeme všimnúť, je samotný vzorec zo zadania. Keď si rozmyslíme, ako funguje, prídeme na to, že síce každý referenčný bod ovplyvňuje celý obrázok, ale najväčší vplyv má vo svojom bezprostrednom okolí – a naopak, ďaleko od neho je už jeho vplyv pomerne malý. Asi teda príliš nepokazíme, keď si to predstavíme tak, že každý referenčný bod ofarbí len svoje okolie, v každom smere po najbližšie iné referenčné body.

Takéto pozorovanie nám už umožní vyrobiť nejaké prvé jednoduché riešenia. Napríklad keď máme obrázok rozmerov $n \times n$, tak môžeme referenčné body rozmiestniť pravidelne do mriežky rozmerov $\sqrt{2n} \times \sqrt{2n}$. Ich intenzity určíme napríklad tak, že pre každý referenčný bod zoberieme priemer jeho okolia. Výsledný obrázok síce nebude žiadna výhra, ale programuje sa to rozumne ľahko a nejaké body za to budú.

O čosi lepšie riešenie zrejme vieme dosiahnuť tak, že referenčné body nebudeme rozmiestňovať rovnomerne, ale budeme ich dávať *pažravo* vždy tam, kde ich je najviac treba. Ako na to? Budeme referenčné body pridávať postupne po jednom. Vždy, keď pridávame nový, vyskúšame všetkých $256n^2$ možností, kam ho dať. Zakaždým si prepočítame, akú chybu bude mať výsledné riešenie. No a následne zoberieme tú polohu, ktorá viedla k najmenšej chybe.

Takéto riešenie má síce polynomiálnu časovú zložitosť, je však stále dosť pomalé. Na to, aby bolo aspoň trochu použiteľné aj pre obrázky väčšie ako 10×10, ho treba implementovať šikovne. Urobíme preto jeden trik, ktorý odteraz budú používať aj všetky ďalšie riešenia: Keď už máme nejaké referenčné body, budeme si pamätať pre každý pixel obrázka aj súčet hodnôt (intenzita referenčného bodu krát jeho váha), aj súčet samotných váh. Z nich vieme aktuálnu intenzitu pixelu vypočítať v konštantnom čase. Čo je však dôležitejšie,

keď pridáme (alebo odoberieme alebo presunieme) jeden nový referenčný bod, vieme každý pixel v konštantnom čase prepočítať. Nemusíme už teda strácať čas tým, že po každej zmene našej komprimovanej reprezentácie budeme vyhodnocovať celé riešenie od začiatku.

S týmto trikom potrebujeme na vyššie popísané riešenie zhruba $512n^5$ krokov výpočtu. To je OK pre menšie obrázky, ale pre 500×500 by takéto riešenie bežalo rádovo roky. Niekde teda treba ušetriť. Kde? Môžeme napríklad prezerať menej možných polôh nového referenčného bodu, z intenzít pixelov v jeho okolí môžeme odhadnúť malý interval intenzít, ktoré má zmysel skúšať pre referenčný bod... fantázii sa medze nekladú.

Lezenie na kopec

My by sme ale chceli nejaký šikovnejší postup, ktorý bude výstupy dávať rýchlejšie a navyše budú mať lepšiu kvalitu. Ako na to? Jednou zo základných techník číselnej optimalizácie je tzv. *hill climbing*, v preklade teda lezenie na kopec.

Komprimovaný obrázok je tvorený $6n$ číslami: pre každý z $2n$ referenčných bodov potrebujeme povedať jeho súradnice a jeho intenzitu. Predstavme si teraz týchto $6n$ čísel ako akési "GPS súradnice" nejakého "bodu" vo "svete". Samozrejme, tento svet nie je dvojrozmerný (ako u klasických GPS súradníc) ale je až $6n$-rozmerný – to však v našich predstavách spokojne odignorujeme.

Každému "bodu" teraz ešte priradíme "nadmorskú výšku". Tá bude zodpovedať kvalite príslušného riešenia: čím lepšie riešenie je tvorené súradnicami daného bodu, tým vyššia bude nadmorská výška v ňom.

V našej úlohe môžeme urobiť jedno dôležité pozorovanie: **keď spravíme malú zmenu** "súradníc bodu" (teda keď buď trošku posunieme jeden referenčný bod alebo trošku zmeníme jeho intenzitu), **spôsobíme tým len malú zmenu** kvality riešenia. Celá naša krajina, ktorú si predstavujeme, je teda v podstate pekná spojitá.

To, čo by sme v ideálnom prípade chceli, je nájsť najvyšší bod celej krajiny – globálne maximum výšok, teda optimálne riešenie. Toto môže byť veľmi ťažké, a tak sa uspokojíme s o čosi horším výsledkom: nájdeme aspoň nejaké kopce, nájdeme vrchol každého z nich (lokálne maximum) a spomedzi nich si vyberieme ten najlepší.

Ako na to? Začneme vždy v náhodnom bode našej krajiny. (Zvolíme si teda náhodne súradnice referenčných bodov aj ich intenzity.) Tento bod zrejme leží na úbočí nejakého kopca. Teda existujú nejaké smery, ktorými keď sa po krajine pohneme, pôjdeme hore kopcom. Dobrý spôsob, ako liezť hore kopcom, je taký, že v každom kroku si vyberieme ten smer, ktorý vedie najstrmšie dohora. Čo to znamená v našej úlohe? Vyskúšame nejaké malé zmeny (trochu posunúť alebo prefarbiť niektorý referenčný bod). Ak žiadna z nich riešenie nezlepší, sme už na vrchole kopca. A ak ho nejaké zmeny zlepšia, tak si vyberieme tú z nich, ktorá ho zlepší najviac. Tú aplikujeme a postup lezenia na kopec opakujeme z nového bodu.

Simulované žíhanie

Naše výstupy sme generovali ešte o čosi šikovnejším postupom. V angličtine je známy pod názvom *simulated annealing*. Ide o podobnú *heuristiku* ako lezenie na kopec, len o čosi zložitejšiu, ale zato často o čosi účinnejšiu. (Mimochodom, ide o postup, ktorý bol spoluvynájdený u nás – v roku 1985 ho vymyslel Vlado Černý a aplikoval ho na riešenie problému obchodného cestujúceho.)

Na rozdiel od lezenia na kopec, ktoré sa po krajine pohybovalo "spojito", pri simulovanom žíhaní po krajine viac "skáčeme". Na začiatku je povolená veľkosť skoku veľká (teda v našom prípade môžeme napr. zobrať jeden referenčný bod a ľubovoľne mu zmeniť všetky parametre). Po každom skoku vyhodnotíme, či je nové riešenie lepšie alebo horšie ako to staré. Lepšie riešenie si necháme s veľkou pravdepodobnosťou, horšie s malou. Teda ak je napr. nové riešenie horšie od toho čo práve máme, s pravdepodobnosťou 90 percent ostaneme kde sme, ale s pravdepodobnosťou 10 percent aj tak prejdeme do nového bodu. (Toto je dôležité aby sme mali šancu dostať sa preč z nejakého "malého miestneho pohoria". Pravdepodobnosti samozrejme nemusia byť 10-90, môžu napr. závisieť aj od rozdielu kvality oboch riešení.)

Ako proces hľadania riešenia pokračuje, postupne zmenšujeme povolenú veľkosť skoku. Keďže preferujeme lepšie riešenia pred horšími, časom sa s veľkou pravdepodobnosťou dostaneme do "hornatej oblasti" a zároveň veľkosť skoku klesne natoľko, že už z nej neodídeme – ak by sme aj skočili mimo, ďalší skok nás zase skoro určite vráti späť. A tam sa vlastne stále deje to isté: v rámci oblasti nájdeme nejaké miesto, kde sú kopce ešte vyššie, a keď časom veľkosť skoku ešte viac klesne, už tam ostaneme.

Ak to príliš nedáva zmysel, pozrite si animovaný gif na Wikipédii.

No a celý tento proces môžeme samozrejme veľakrát nezávisle na sebe spustiť a vybrať najlepšie z takto zostrojených riešení. Samozrejme, ani táto technika nie je všemocná a nemáme žiadnu záruku, že nájde globálne optimum, ani záruku, ako ďaleko bude nami nájdené riešenie od globálneho optima.

Listing programu (C++)

```
#include <cmath>
#include <cstdlib>
```

```cpp
#include <iostream>
#include <utility>
#include <vector>
using namespace std;

const int PHASE_COUNT = 6;
const int STARTING_MAX_JUMP = 1 << PHASE_COUNT;
const int STARTING_MAX_FADE = 1 << PHASE_COUNT;
const int STEP_COUNT = 5000;

typedef pair<int,int> point;
typedef pair<point,int> refpoint;

int R, C;
vector< vector<int> > input;

vector<refpoint> best;
double best_score;

vector< vector<double> > isum, wsum;

void load() {
    cin >> C >> R;
    input.resize(R, vector<int>(C,0));
    for (int r=0; r<R; ++r) for (int c=0; c<C; ++c) cin >> input[r][c];
    isum.resize( R, vector<double>(C,0) );
    wsum.resize( R, vector<double>(C,0) );
}

int square_dist(const point &A, const point &B) {
    return (A.first-B.first)*(A.first-B.first) + (A.second-B.second)*(A.second-B.second);
}

void add_reference_point(const refpoint &rp, int weight=1) {
    for (int r=0; r<R; ++r) for (int c=0; c<C; ++c) {
        int sd = square_dist( point(r,c), rp.first );
        double w = (sd == 0 ? 1000 : 1./sd);
        isum[r][c] += rp.second * w * weight;
        wsum[r][c] += w * weight;
    }
}

void get_intensities(const vector<refpoint> &attempt) {
    for (int r=0; r<R; ++r) for (int c=0; c<C; ++c) { isum[r][c] = 0., wsum[r][c] = 0.; }
    for (auto rp : attempt) add_reference_point(rp);
}

double score() {
    double diff = 0.;
    for (int r=0; r<R; ++r) for (int c=0; c<C; ++c) {
        double in = isum[r][c] / wsum[r][c];
        diff += (in - input[r][c])*(in - input[r][c]);
    }
    return sqrt( diff / R / C );
}

int main() {
    srand(time(NULL));
    load();

    vector<refpoint> attempt;
    for (int n=0; n<R+C; ++n) attempt.push_back( refpoint( point( rand()%R, rand()%C ), rand()%256 ) );

    get_intensities(attempt);
    best_score = score();
    best = attempt;

    int max_jump = STARTING_MAX_JUMP, max_fade = STARTING_MAX_FADE;

    for (int phase=0; phase<PHASE_COUNT; ++phase) {
        attempt = best;
        double att_score = best_score;
        get_intensities(attempt);

        for (int step=0; step<STEP_COUNT; ++step) {

            vector<refpoint> new_attempt = attempt;

            int n = rand() % new_attempt.size();
            while (true) {
                int nr = attempt[n].first.first  - max_jump + rand()%(2*max_jump+1);
                int nc = attempt[n].first.second - max_jump + rand()%(2*max_jump+1);
                int ni = attempt[n].second - max_fade + rand()%(2*max_fade+1);
                new_attempt[n] = refpoint( point(nr,nc), ni );
                if (0 <= nr && nr < R && 0 <= nc && nc < C && 0 <= ni && ni < 256) break;
            }

            add_reference_point( attempt[n], -1 );
            add_reference_point( new_attempt[n], +1 );
            double new_score = score();

            if (new_score < best_score) {
                best_score = new_score;
                best = new_attempt;
            }

            bool revert = false;
            if (new_score < att_score && rand()%10 >= 9) revert = true;
            if (new_score >= att_score && rand()%10 >= 0) revert = true;
```

```
            if (revert) {
                add_reference_point( attempt[n], +1 );
                add_reference_point( new_attempt[n], -1 );
            } else {
                att_score = new_score;
                attempt = new_attempt;
            }
        }

        max_jump /= 2;
        max_fade /= 2;
    }

    cout << (R+C) << endl;
    for (auto rp : best) cout << (1+rp.first.second) << " " << (1+rp.first.first) << " " << rp.second << endl;
}
```

Hodnotenie

Bodovanie praktickej časti úlohy je nasledovné. Každý vstup sa hodnotí samostatne a dá sa zaň získať 1 bod. Referenčná odchýlka r (počet bodov potrebný na plný počet) sa vypočíta ako aritmetický priemer najlepšej účastníckej a najlepšej vedúcovskej odchýlky. Keď vaša odchýlka v konkrétnom vstupe bola o, tak ste dostali $(\max(0, \min(50, r - o + 50)) \cdot 0.02)^2$ bodov.

6. Overené informácie

Vlejd
(max. 10 b za popis, 10 b za program)

Najhrubšia sila

Najprv si potrebujeme uvedomiť, kedy tri body tvoria trojuholník. Vždy, keď nie sú na jednej priamke. Ako sa to dá otestovať? Môžeme si vyjadriť rovnice všetkých strán potenciálneho trujuholníka a porovnať ich. Nejde to aj ľahšie? Samozrejme, že ide. Stačí nám totiž vyrátať len smernice daných úsečiek (tangens uhlu, ktorý zvierajú s osou x) a ak sú rovnaké, tak sú dané úsečky rovnobežné a teda nemôžu tvoriť trojuholník. Ale ide to ešte ľahšie.

Môžme si uvedomiť, že trojuholník je len na jednej priamke (teda to nie je trojuholník) práve vtedy, keď má nulový obsah. Najjednoduchší a najlepší spôsob, ako zistiť, či tri body A, B, C ležia na jednej priamke, je spočítať *vektorový súčin* vektorov $(C - A)$ a $(B - A)$ a overiť či je nulový. V dvoch rozmeroch vypočítame vektorový súčin $v \times u = vx \cdot uy - ux \cdot vy$, kde napr. vx je x-ová zložka vektora v.

Výhody vektorového súčinu:

- Je rýchly a jednoduchý.
- Pokiaľ body mali celočíselné súradnice, tak vektorový súčin môžeme spočítať v celých číslach, čím sa vyhneme nepresnostiam.
- Funguje aj pre všelijaké patologické prípady, kedy je viac bodov na rovnakom mieste a podobne.

No a už máme prvé riešenie. Prejdeme všetky trojice bodov (dávame si ale pozor na opakovania) a pre každú trojicu si zistíme, či tvorí korektný trojuholník. Dostávame riešenie v čase $O(n^3)$.

Listing programu (C++)

```
#include<cstdio>
#define For(i, n) for(int i = 0; i<(n); ++i)
typedef long long ll;
int n, X[4007], Y[4007];

int main() {
    scanf("%d",&n);
    For(i, n) scanf("%d%d", X+i, Y+i);
    int pocet = 0;
    For(i, n) For(j, i) For(k, j) {
        ll x1 = X[i]-X[j], y1 = Y[i]-Y[j];
        ll x2 = X[i]-X[k], y2 = Y[i]-Y[k];
        // vektorový súčin
        if (x1*y2 - x2*y1 != 0) pocet++;
    }
    printf("%d\n",pocet);
}
```

O hľadaní v priamkách

Dôležitá myšlienka na ceste k lepšiemu riešeniu je, že to môžeme počítať aj naopak. Keby každá trojica bodov tvorila trojuholník, tak by sme mali $\frac{n(n-1)(n-2)}{6}$ trojuholníkov. Potom, ak by sme vedeli zrátať počet

trojíc, ktoré netvoria trojuholníky, tak ich len odrátame a máme riešenie. Budeme teda rátať trojice bodov, ktoré netvoria trojuholník, resp. ktoré ležia na jedne priamke. Od teraz uvažujme len priamky, na ktorých sú aspoň 2 body z našej množiny. Potom maximálny počet rôznych priamok je $\frac{n(n-1)}{2}$, čo nie je až tak veľa.

V podstate by sme chceli pre každú priamku zistiť, koľko bodov na nej leží. Ale potrebujeme to spraviť rýchlo. Napríklad to môžeme spočítať tak, že najprv preskúmame priamky, ktoré prechádzajú prvým bodom, potom priamky, ktoré prechádzajú druhým, atď.

Ku každému bodu si vieme vyrátať smernice (uhol, aký zviera s osou x) všetkých priamok, ktoré ním prechádzajú. Prejdeme teda všetky ostatné body, vyrátame smernicu a uložíme si ju. Takýchto smerníc potom budeme mať $n-1$. Niektoré smernice budú rôzne, ale budú tam nejaké zhluky rovnakých smerníc.

Čo ale znamená, že pre dva body nám vyjde rovnaká smernica? No to, že sú spolu s pôvodným bodom na tej istej priamke a teda netvoria trojuholník. Ak nám pre m bodov vyjde tá istá smernica, tak aj s pôvodným máme $m+1$ bodov, ktoré ležia na jednej priamke a netvoria trojuholníky. Dokopy to je $\frac{m(m-1)}{2}$ trojíc, ktoré netvoria trojuholníky a je v nich pôvodný bod. Na to, aby sme vedeli zistiť, koľkokrát sme dostali ktorú smernicu si ich môžeme jednoducho usporiadať a prejsť. (Prípadne sa dajú pamätať v tzv. mape.) Oba postupy nás stoja čas $O(n\log(n))$.

Nakoľko tento postup musíme urobiť pre každý bod, dostávame riešenie s časovou zložitosťou $O(n^2\log(n))$. Pamätať si musíme len pôvodné body a pre jeden aj všetky smernice. Pamäťová zložitosť je teda $O(n)$. Pri celom tomto postupe si ale treba dávať pozor na to, že každú trojicu, ktorá netvorí trojuholník zarátame trikrát. Druhá nepríjemnosť je, že smernice nie sú celé čísla. Tu pomôže, že smernica je len podiel dvoch celých čísel (lebo súradnice trojuholníkov sú celočíselné) a takéto racionálne číslo si vieme pamätať ako dvojicu (`pair<>` v C++). Tieto zlomky však treba previesť do základného tvaru, teda vydeliť najväčším spoločným deliteľom (funkcia `__gcd()` v C++).

Listing programu (C++)

```
#include<cstdio>
#include<vector>
#include<cmath>
#include<algorithm>
#define For(Q,W) for(long long Q=0; Q<W; Q++)
using namespace std;
typedef long long L;

bool cmp(pair<L,L> a,pair<L,L> b){
    return a.second*b.first < b.second*a.first;
}

int main() {
    long long n;
    scanf("%lld",&n);
    vector<pair<L,L> > body;
    For(i,n){
      L a, b;
      scanf("%lld%lld",&a,&b);
      body.push_back(make_pair(a,b));
    }
    L vsetkych = (n*(n-1)*(n-2))/6;
    L zlych =0 ;
    For(i,n){
      vector<pair<L,L> > priamky;
      pair<L,L> a = body[i];

      For(j,n){
        if(i==j) continue;
        pair<L,L> b = body[j];
        pair<L,L> c = make_pair(b.first - a.first, b.second - a.second);
        if( c.first < 0) {
          c.first = 0ll-c.first;
          c.second = 0ll-c.second;
        }
        else{
          if (c.first ==0) c.second = abs(c.second);
      }

        priamky.push_back(c);
      }

      sort(priamky.begin(), priamky.end(),cmp);

      if(priamky.size()>0){
        pair<L,L> smernica= priamky[0];
        L pocet_na_nej=1;
        for(L j=1;j<priamky.size();j++){
          if(priamky[j].first*smernica.second == priamky[j].second*smernica.first){
              pocet_na_nej+=1;
          }
          else{
              zlych += (pocet_na_nej*(pocet_na_nej-1))/2;
              pocet_na_nej =1;
              smernica=priamky[j];
          }
        }
```

```
            zlych += (pocet_na_nej*(pocet_na_nej-1))/2;
        }

    }
    printf("%lld\n",vsetkych-zlych/3);
}
```

Jano

7. Obsadenie toboganov plavčíkmi

(max. 15 b za popis, 10 b za program)

Ľahko by sme napísali program, ktorý by vyskúšal všetky možné ofarbenia (tých je 2^n), pre každé ofarbenie by sme len cyklom prešli hrany, vynásobili bezpečnosti a na konci to celé sčítali.

Tento algoritmus by bol príliš pomalý, pretože 2^{40} je dosť veľa (približne tisíc miliárd) a to ešte treba zakaždým prechádzať hrany.

To, čo by sme však zvládli je skúšať 2^{20} možností, to je predsa len milión. Ale čoho je najviac $n/2$? No predsa vrcholov v menšej polovici vrcholov[4].

Dôležitá informácia v zadaní bola, že graf je bipartitný. To znamená, že jeho vrcholy sa dajú rozdeliť do dvoch množín A, B tak, aby medzi žiadnymi vrcholmi v A neviedla hrana a tiež medzi žiadnymi vrcholmi v B neviedla hrana (hrany teda idú len z A do B a naopak). Tieto množiny voláme partície.

Tým pádom vrcholy v jednej partícii (napríklad v B) sú navzájom nezávislé. Keď zmeníme farbu vrchola x z B, tak to nijako neovplyvní bezpečnosti hrán, ani súčiny bezpečností pri iných vrcholoch z B.

Predstavme si, že nejako zafixujeme farby všetkých vrcholov v A, napríklad nech sú všetky biele. Potom sa môžeme zaujímať, aká je čiastočná odpoveď pre takéto ofarbenia, teda súčet celkových bezpečností, pre ktoré sú všetky vrcholy z A biele. Toto číslo označíme $G(B)$. Keby sme z grafu zahodili všetky vrcholy z partície B aj s ich hranami a nechali len jeden vrchol x, mohli by sme nazvať čiastočnú odpoveď pre tento vrchol $G(x)$.

Túto hodnotu vieme vypočítať pomerne ľahko, buď je x biely (vtedy dostaneme nejaký súčin hodnôt hrán w_{xb}), alebo je x čierny (a dostaneme súčin w_{xc}). $G(x) = w_{xb} + w_{xc}$.

Po úvahách o nezávislosti zistíme, že $G(B) = G(x_1) \cdot G(x_2) \cdot \ldots \cdot G(x_b)$, teda súčin čiastkových hodnôt pre všetky vrcholy z B. Môžete si aj sami rozmyslieť, prečo je to tak.

Tým pádom, pre jedno konkrétne ofarbenie A vieme rýchlym prechodom cez všetky vrcholy B zistiť $G(B)$ pre dané A. Tieto hodnoty stačí sčítať pre všetky možné ofarbenia množiny A.

Pred týmto počítaním ešte musíme rozdeliť graf na partície a nezabudnúť pomenovať A tu menšiu z nich. Na rozdelenie postačí DFS prechod.

Celková časová zložitosť bude $O(2^{n/2}(n+m))$. Pre veľké n je $2^{n/2}$ je oveľa, oveľa menšie ako 2^n, takže robiť optimalizáciu, ktorá by nám toto zabezpečila má zmysel.

Listing programu (C++)

```
#include<cstdio>
#include<algorithm>
#include<vector>
using namespace std;
#define For(i, n) for(int i = 0; i<(n); ++i)
#define mp make_pair
#define MOD 1000000007LL

typedef long long ll;
typedef pair<pair<int,int>,vector<int> > vertex;

vector<vector<int> > G;
vector<vector<vertex> > E;
vector<bool> T;
vector<int> C,B;

void dfs(int v, int f) {
    T[v]=true;
    if(f==0) C.push_back(v);
    else B.push_back(v);
    For(i,G[v].size()) {
        int w=G[v][i];
        if(T[w]) continue;
        dfs(w,(f+1)%2);
    }
}

int main(){
    int n,m;
    scanf("%d%d",&n,&m);
    G.resize(n);
    E.resize(n);
    For(i,m) {
        int x,y,a,b,c,d;
```

[4] Namiesto menšej polovice sme mali napísať menšiu partíciu ale bolo by to zrozumiteľnejšie?

```
        scanf("%d%d%d%d%d%d",&x,&y,&a,&b,&c,&d);
        x--; y--;
        G[x].push_back(y);
        G[y].push_back(x);
        vector<int> pom; pom.push_back(a); pom.push_back(b); pom.push_back(c); pom.push_back(d);
        E[x].push_back(mp(mp(x,y),pom));
        E[y].push_back(mp(mp(x,y),pom));
    }
    T.resize(n,false);
    For(i,n)
        if(!T[i]) dfs(i,0);
    if(C.size() > B.size()) swap(C,B);
    vector<int> F; F.resize(n,-1);
    int p=C.size();
    ll res=0;
    For(i,1<<p) {
        For(j,p) {
            if(i&(1<<j)) F[C[j]]=0;
            else F[C[j]]=1;
        }
        ll zat=1;
        For(j,n) {
            if(F[j]!=-1) continue;
            ll suc0=1,suc1=1;
            For(k,E[j].size()) {
                if(E[j][k].first.first==j && F[E[j][k].first.second]==0)
                    {suc0*=E[j][k].second[0]; suc1*=E[j][k].second[2];}
                if(E[j][k].first.first==j && F[E[j][k].first.second]==1)
                    {suc0*=E[j][k].second[1]; suc1*=E[j][k].second[3];}
                if(E[j][k].first.second==j && F[E[j][k].first.first]==0)
                    {suc0*=E[j][k].second[0]; suc1*=E[j][k].second[1];}
                if(E[j][k].first.second==j && F[E[j][k].first.first]==1)
                    {suc0*=E[j][k].second[2]; suc1*=E[j][k].second[3];}
                suc0%=MOD; suc1%=MOD;
            }
            zat=(suc0*zat+suc1*zat)%MOD;
        }
        res=(res+zat)%MOD;
    }
    printf("%d\n",(int)res);
}
```

Žaba

8. Okázalá svadba

(max. 15 b za popis, 10 b za program)

Osobne sa musím priznať, že sa mi táto úloha veľmi páčila. Jej riešenie je totiž také skladačkové, treba spraviť zopár správnych aj nesprávnych pozorovaní, ale výsledok je o to krajší. A ako ste mali začať? No predsa tak, že ste naprogramovali jednoduché n^2 riešenie, ktoré postupne skúšalo všetky možné úseky a zisťovalo ich maximum a bitový or. A aj keď to vyzerá ako riešenie so zložitosťou $O(n^3)$, určite ho dokážete ľahko zredukovať, keď si uvedomíte, že z výsledku pre jeden úsek viete jednoducho vypočítať výsledok pre úsek o jedno dlhší.

Samozrejme, takéto jednoduché riešenie nie je až také zaujímavé. Potrebujeme vymyslieť niečo rádovo rýchlejšie. Dobrým postupom je hľadať, čoho môže byť v zadaní málo, a poprípade si fixovať nejaké prvky napevno. Začnime druhým prípadom, skúsime si dopredu určiť maximum úseku. Prečo práve maximum? Lebo to sa aj naozaj nachádza v poli. Pre ľubovoľný úsek bude jeho maximum len jeden z prvkov poľa, zatiaľčo bitový or môže byť takmer ľubovoľné číslo.

Vyberieme si teda nejaký prvok na m-tej pozícii a budeme sa pozerať na všetky úseky, ktoré ho obsahujú. Naviac budeme predpokladať, že tento prvok je maximum pre všetky tieto úseky. Samozrejme, niečo také platí iba pre najväčší prvok poľa. Ak si ako náš prvok zvolíme inú hodnotu, pravdepodobne niektoré úseky budú obsahovať väčšie číslo. To budeme ale potichu ignorovať a budeme sa tváriť, že nech je na m-tej pozícii hocijaké číslo, je maximom pre všetky úseky, ktoré ho obsahujú. Naším cieľom bude pre tieto úseky rátať or čísel v nich a pomocou toho zisťovať výslednú hodnotu.

A hoci naše vybrané číslo nemusí byť skutočné maximum každého podúseku, hodnota, ktorú vďaka tomu vypočítame, nebude väčšia ako naozajstné maximum súčtu maxima a bitového or-u v tomto úseku. A tú správnu hodnotu zarátame, keď si ako m-tý prvok zvolíme to správne číslo[5].

Maximum máme určené, pozerajme sa teraz na bitový or. Ten má tú peknú vlastnosť, že toto číslo neklesá, keď sa k úseku pridávajú ďalšie prvky, pretože sa môže len zväčšiť priorovaním nového bitu, ktorý sa v ňom dovtedy nevyskytoval.

Zoberme si napríklad všetky úseky, kde je m-tý prvok najľavším v tomto úseku a pozrime sa na bitové ory ich čísel. Hodnota tohoto oru sa zmení vždy len vtedy, keď sa pridá prvok, ktorý obsahuje nejaký "nový" bit, taký, ktorý doposiaľ nebol v našom ore. Ale toto sa môže stať najviac 31 krát, lebo práve toľko bitov má najväčšie možné číslo 10^9.

[5] Možno sa pokúste trochu rozdýchať to, čo som sa tu snažil naznačiť a uvedomte si, že naozaj je to v pohode.

Je len 31 zaujímavých úsekov, čo sú také úseky, kde sa zmení hodnota oru. To platilo pre úseky zľava zarovnané na m-tý prvok. Ale to isté sa dá povedať aj opačným smerom a dokopy nám to dá najviac 31^2 zaujímavých úsekov, čo dostaneme ako všetky možnosti začiatkov a koncov zarovnaných zaujímavých úsekov. A to je pomerne málo.

Otázka ale je, že čo s úsekmi, ktoré nie sú zaujímavé. A čo s ostatnými dĺžkami postupností? Všimnime si, že naše zaujímavé úseky sú naviac najkratšie možné, lebo zaujímavé sú práve tým, že sa niečo zmenilo. A druhá dôležitá vlastnosť je, že ak máme úsek dĺžky d so súčtom maxima a oru rovným l, tak všetky úseky dĺžky viac ako d majú súčet maxima a oru rovný aspoň l. Prečo? Lebo zoberieme ten úsek dĺžky d, pridáme nejaké prvky a je jasné, že ani maximum ani or prvkov sa nám zmenšiť nemohol.

Zhrnieme si teraz riešenie. Pre každý prvok m poľa vyrátame dve pomocné polia (v programe označené ako Ml a Mr). Tie hovoria o každom možnom bite, kde naľavo (poprípade napravo) sa nachádza najbližšie číslo obsahujúce tento bit. Pričom prvok je sám sebe naľavo (aj napravo). Toto vieme spraviť jedným prechodom. Tieto dve polia si utriedime podľa vzdialenosti. Každá dvojica, kde jeden prvok je zľava a druhý sprava, nám určuje jeden zaujímavý úsek obsahujúci daný prvok m. O prvku m predpokladáme, že je maximom svojich zaujímavých úsekov, a or čísel zaujímavého úseku vyrátame ľahko, lebo presne vieme, ktoré bity boli pridané dovtedy (predchádzajúce prvky v Ml a Mr). Do poľa výsledkov si preto zaznačíme, že úsek tejto dĺžky má aspoň takýto súčet maxima a oru, pričom si pamätáme zatiaľ videné maximum.

Niektoré prvky poľa výsledkov sú nevyplnené, lebo neexistuje zaujímavý úsek danej dĺžky. V takom prípade bude odpoveď maximum súčtu pre nejaký menší zaujímavý úsek. Môže sa dokonca stať, že máme dlhý zaujímavý úsek, ktorý má menší súčet ako nejaký iný menší zaujímavý úsek. Aby sme toto všetko napravili, toto pole výsledkov upravíme tak, že na i-tom políčku je maximum z úsekov dlhých najviac i. Toto pole určuje výsledok.

Čo sa týka zložitosti, označme si M maximálny prvok postupnosti zo vstupu. Pamäťová zložitosť je potom $O(n \log M)$, lebo pre každý prvok si pamätáme informáciu o $\log M$ bitoch. A časová zložiťosť je $O(n \log^2 M)$, lebo musíme vyskúšať všetky zaujímavé úseky.

Listing programu (C++)

```
#include <cstdio>
#include <cassert>
#include <algorithm>
#include <vector>
#include <iostream>
#include <string>
#include <map>
#include <set>
#include <queue>
#include <cstring>
#include <cstdlib>
#include <cmath>
#define For(i, n) for (int i = 0; i < (int) n; ++i)
#define SIZE(x) ((int) (x).size())
#define mp(a, b) make_pair(a, b)
using namespace std;

typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;

int main() {
    int n;
    scanf("%d",&n);
    vector<int> A; A.resize(n);
    For(i,n) scanf("_%d",&A[i]);
    int moc=1;
    For(i,n) while((1<<moc)<=A[i]) moc++;
    vector<int> P; P.resize(moc,-1);
    vector<vector<pii> > Ml; Ml.resize(n);
    vector<vector<pii> > Mr; Mr.resize(n);
    For(i,n) {
        For(j,moc) if((A[i]&(1<<j))!=0) P[j]=i;
        //popridavaj do Ml
        For(j,moc) {
            if(P[j]==-1) continue;
            Ml[i].push_back(mp(i-P[j],j));
        }
    }
    P.clear(); P.resize(moc,-1);
    for(int i=n-1; i>=0; i--) {
        For(j,moc) if((A[i]&(1<<j))!=0) P[j]=i;
        //popridavaj do Mr
        For(j,moc) {
            if(P[j]==-1) continue;
            Mr[i].push_back(mp(P[j]-i,j));
        }
    }
    For(i,n) {
        sort(Ml[i].begin(),Ml[i].end());
        sort(Mr[i].begin(),Mr[i].end());
    }
    vector<int> Vys;
    Vys.resize(n,-1);
```

```
    //prejdi vsetko
    For(i,n) {
        if(A[i]==0) continue;
        int p1=0;
        For(i1,Ml[i].size()+1) {
            if(i1!=0) p1|=(1<<Ml[i][i1-1].second);
            int p2=0;
            For(i2,Mr[i].size()+1) {
                if(i2!=0) p2|=(1<<Mr[i][i2-1].second);
                int vzd=0;
                if(i1!=0) vzd+=Ml[i][i1-1].first;
                if(i2!=0) vzd+=Mr[i][i2-1].first;
                Vys[vzd]=max(Vys[vzd],A[i]+(p1|p2));
            }
        }
    }
    if(Vys[0]==-1) Vys[0]=0;
    for(int i=1; i<n; i++) Vys[i]=max(Vys[i],Vys[i-1]);
    For(i,n) printf("%d\n",Vys[i]);
}
```