A-III-1 Urýchľovač

Získať 3 body nebolo ťažké. Keďže rezistorov je nanajvýš 2000, môžeme vyskúšať každú dvojicu a zistiť, či súčet ich odporov je správny. Takéto riešenie funguje v čase $O(n^2)$, ale nedáva správny výsledok, pokiaľ treba rezistorov viac. Keby sme chceli podobným spôsobom riešiť úlohu naozaj správne, skúšali by sme všetky štvorice, ktorých je už ale až $O(n^4)$.

Dynamické programovanie

Pre získanie zhruba polovice bodov ste mohli použiť úplne iný prístup. Tu si bolo treba uvedomiť, že kombinácie rezistorov, ktoré majú privysoký súčet odporov, nás nemusia zaujímať. Stačí nám pre každý odpor z rozsahu $0 \ldots R$ zistiť najmenší počet rezistorov ktorým ho vieme dosiahnúť. Pre jednoduchosť si budeme pamätať aj to, ktoré odpory dávajú daný súčet.

Túto informáciu budeme zisťovať postupným pridávaním rezistorov. Keby sme nemali žiadne rezistory, vedeli by sme (pomocou 0 rezistorov) dosiahnuť odpor 0. Ostatné odpory by sme nevedeli dosiahnuť vôbec, poznačíme si teda, že na ne potrebujeme nekonečne veľa rezistorov.

Teraz budeme postupne po jednom pridávať rezistory zo vstupu a zakaždým prepočítame, čo sa zmenilo. Čo sa stane, ak pridáme rezistor s odporom x? Ak sme doteraz vedeli nejaký celkový odpor y vyrobiť pomocou z rezistorov, teraz už vieme vyrobiť aj celkový odpor $x + y$ pomocou $z + 1$ rezistorov. Stačí teda raz prejsť celé pole so starými informáciami, vypočítať z nich nové a pozrieť sa, čo sme zlepšili.

Takto dostaneme riešenie v čase $O(R \cdot n)$.

Vzorové riešenie

Naše vzorové riešenie sa bude viacej podobať na úplne prvotné. Čo nám robilo problém je príliš veľa kombinácií. Inšpirujeme sa teda taktikou „meet in the middle“ a pokúsime sa problém vyriešiť tak, že nájdeme dve dvojice, ktoré budú dokopy dávať správny súčet. Ak totižto máme nejakú jednu dvojicu rezistorov, presne vieme povedať, aký súčet má mať druhá dvojica, ktorú ku nej treba doplniť. Ak by sme teda vedeli povedať, ktoré všetky odpory sú dosiahnuteľné dvojicou rezistorov, vedeli by sme sa potom proste pozrieť, či k danému odporu r vieme dosiahnuť aj $R - r$.

Musíme si ale dať pozor, aby sme nevyužili žiaden rezistor dvakrát. To dosiahneme tak, že v množine dosiahnuteľných odporov budeme mať iba odpory dosiahnuteľné pomocou niekoľkých rezistorov zo začiatku a druhú dvojicu k nej budeme vyberať zo zvyšných, takže budú naše dvojice rezistorov zaručene disjunktné.

Pozrime sa na to teda poriadnejšie: Postupne budeme spracovávať rezistory. V okamihu, keď ideme spracovať i-ty rezistor, si budeme pamätať množinu M odporov, ktoré vieme dosiahnuť pomocou dvoch rezistorov s číslom menším ako i. Teraz i-ty rezistor spracujeme:

- Pozrieme sa, či je v M hodnota rovná hľadanému R mínus odpor i-teho rezistora.
- Pre každé $j > i$ sa pozrieme, či je v M hodnota rovná hľadanému R mínus celkový odpor i-teho a j-teho rezistora.
- Ak ani jedna z vyššie skúšaných možností nezafungovala, tak do množiny M pridáme pre každé $j < i$ odpor dvojice tvorenej j-tym a i-tym rezistorom, a ideme spracovať rezistor s číslom $i + 1$.

Ako množinu budeme používať buďto binárny vyvažovací strom, alebo hashovaciu tabuľku, ktorá súčtu odporov bude vedieť rýchlo priradiť dvojicu (alebo jediný) rezistor s daným súčtom. Oboje máme predimplementované v niektorých jazykoch (napr. ako `set` a `unordered_set` v C++).

V závislosti od použitej množinovej implementácie potom dostaneme buďto riešenie v čase $O(n^2 \log n)$ pre stromy, alebo $O(n^2)$ v priemernom prípade pre hashovaciu tabuľku. V nižšie uvedenom programe sme si vybrali druhú možnosť. Upozorníme ešte na drobný detail: pre každú sumu, ktorú vieme dosiahnuť pomocou dvoch už spracovaných rezistorov, si pamätáme len jeden konkrétny spôsob, ako to spraviť – hoci tých spôsobov môže existovať viac, nám stačí poznať ľubovoľný jeden.

Implementáciu si ešte môžeme uľahčiť drobným trikom: Medzi rezistory zo vstupu pridáme tri nové rezistory s odporom 0. Pre takto upravený vstup nám stačí nájsť riešenie tvorené *presne* štyrmi rezistormi. Ak obsahuje nejaké nuly, tie jednoducho nevypíšeme na výstup.

Listing programu (C++)

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

int main() {
    int X, N;
    cin >> X >> N;
    vector<int> A(N+3,0);
    for (int n=0; n<N; ++n) cin >> A[n];
    N += 3;

    unordered_map< int, pair<int,int> > seen;
    for (int i=0; i<N; ++i) {
        // skúsime nájsť štvoricu tvaru (a,b,i,j) v ktorej a<b<i<j
        for (int j=i+1; j<N; ++j)
            if (seen.count( X-A[i]-A[j] )) {
                auto tmp = seen[ X-A[i]-A[j] ];
                for (int x : {tmp.first,tmp.second,i,j})
                    if (A[x]) cout << A[x] << " ";
                return 0;
            }
        // pridáme nové dvojice odporov obsahujúce i-ty
        for (int j=0; j<i; ++j) seen[ A[j]+A[i] ] = {j,i};
    }

    cout << "Koniec sveta sa odklada\n";
}
```

Iné riešenie

Existujú aj riešenia, ktoré dosiahnu rovnakú časovú zložitosť bez použitia zložitých dátových štruktúr. Jedno také teraz popíšeme.

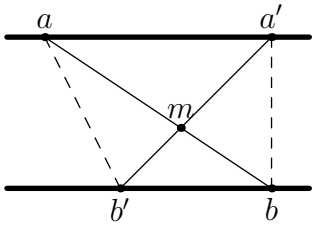
Začneme vyššie popísaným trikom s tromi nulami. Následne do poľa vygenerujeme záznamy tvaru (odpor i + odpor j, i, j) a usporiadame ich podľa odporu. Na to môžeme použiť buď všeobecné triedenie, čím dostaneme časovú zložitosť $O(n^2 \log n)$, alebo RadixSort v $O(n^2)$.

Následne budeme postupne toto pole prechádzať a pre každú dvojicu (s, i, j) budeme hľadať, či v ňom existuje dvojica $(R - s, k, l)$ taká, že i, j, k, l sú štyri rôzne indexy. Opäť máme dve jednoduché možnosti ako to robiť. A opäť jedna vedie k riešeniu v čase $O(n^2)$ a druhá k $O(n^2 \log n)$. Popíšeme si tú druhú z nich. Pre každý konkrétny záznam (s, i, j) pomocou binárneho vyhľadávania nájdeme prvý záznam so súčtom $R - s$. A už len prejdeme cez všetky záznamy, ktoré majú tento súčet, a pre každý z nich sa pozrieme, či sú jemu zodpovedajúce indexy k, l rôzne od i, j.

Prečo toto riešenie funguje efektívne? Nemohlo by sa stať, že bude pomalé v situáciách, keď tú istú sumu vieme dosiahnuť veľa spôsobmi? Nemohlo. A to preto, že keď hľadáme dvojicu indexov so súčtom $R - s$ a narazíme (prvýkrát počas behu programu) na situáciu, že sú takéto dvojice aspoň tri, určite aspoň jedna z nich bude disjunktná s i, j, a teda sme našli riešenie a môžeme rovno skončiť.

A-III-5 Kompy

Na úvod začneme pozorovaním, že v optimálnom riešení sa trasy žiadnych dvoch kômp nekrížia. Ak by sa krížili, tak podľa obrázku existuje preusporiadanie trás s celkovým súčtom dĺžok menším, čo však nesedí s tvrdením, že riešenie je optimálne.



$$\begin{aligned}|ab| + |a'b'| &= (|am| + |mb|) + (|a'm| + |mb'|)\\ &= (|am| + |mb'|) + (|a'm| + |mb'|) > |ab'| + |a'b|\end{aligned}$$

Z vyššie uvedeného pozorovania vyplýva, že najpravejšie miesto na jednej strane kanála bude určite spojené kompou s najpravejším miestom na druhej strane. Prečo? Lebo keby to tak nebolo, každé z nich by muselo byť spojené s nejakým iným miestom na protiľahlej strane, a tieto dve spojenia by sa križovali.

Označme teraz $\mathrm{OPT}(i, j)$ ako najmenší možný súčet dĺžok trás kômp, ktoré spájajú prvých i miest na ľavej strane a prvých j miest na pravej strane. Vieme už, že v tomto optimálnom riešení máme kompu spájajúcu i-te ľavé a j-te pravé miesto. Čo vieme ďalej povedať o tom, ako toto optimálne riešenie vyzerá?

Vo všeobecnosti sú tri možnosti. Prvou je, že ani z i-teho ľavého, ani z j-teho pravého miesta už žiadna ďalšia kompa nevedie. V takomto prípade teda ešte potrebujeme pospájať prvých $i - 1$ miest vľavo a prvých $j - 1$ vpravo, čo vieme najlepšie spraviť so súčtom dĺžok $\mathrm{OPT}(i - 1, j - 1)$.

Zvyšné dve možnosti sú symetrické: buď idú ešte nejaké kompy z i-teho ľavého miesta, alebo idú nejaké z j-teho pravého. (Nemôžu ísť z oboch, lebo by sa križovali.) Ak ešte majú ísť nejaké kompy z i-teho ľavého miesta, tak vlastne chceme prepojiť prvých i miest vľavo a prvých $j - 1$ vpravo, čo vedie k celkovej dĺžke $\mathrm{OPT}(i - 1, j)$. Analogicky na opačnej strane.

Pre $i, j \geq 2$ teda platí nasledujúce tvrdenie:

$$\mathrm{OPT}(i, j) = d(i, j) + \min\left\{\, \mathrm{OPT}(i - 1, j),\ \mathrm{OPT}(i, j - 1),\ \mathrm{OPT}(i - 1, j - 1) \,\right\}$$

(Hodnota $d(i, j)$ je vzdialenosť i-tého miesta na ľavej strane a j-tého miesta na pravej strane.)

Spoločne s $\mathrm{OPT}(1, 1) = d(1, 1)$, $\mathrm{OPT}(i, 1) = d(i, 1) + \mathrm{OPT}(i - 1, 1)$ a $\mathrm{OPT}(1, j) = d(1, j) + \mathrm{OPT}(1, j - 1)$ tak máme rekurzívny predpis pre výpočet všetkých hodnôt $\mathrm{OPT}(i, j)$, $1 \leq i \leq n$ a $1 \leq j \leq m$.

Pomocou vyššie uvedeného vzťahu si teda vieme vypočítať hodnotu $\mathrm{OPT}(n, m)$: súčet dĺžok kômp v optimálnom riešení. My však potrebujeme jedno optimálne riešenie aj zostrojiť.

Na chvíľu predpokladajme, že pamäťový limit umožňuje uloženie poľa čísel o veľkosti $n \times m$. Kvôli nutnosti vypísať optimálne riešenie potrebujeme okrem hodnôt OPT aj pole REK, ktoré bude pre každé (i, j) obsahovať informáciu, pre ktorú z vyššie uvedených troch možností hľadané minimum nastalo. Pomocou tohto poľa potom vieme spätne, začínajúc kompou $n - m$ a končiac kompou $1 - 1$, zostrojiť jedno optimálne riešenie.

Takéto riešenie má časovú aj pamäťovú zložitosť $O(nm)$ a získa približne 10 bodov z 15 možných.

Problém je v tom, že pole OPT sa nám pre veľké vstupy do pamäte nezmestí. Pri použití štandardného dátového typu `double` by sme naň potrebovali až 3 GB pamäte.

Na samotný výpočet výpočet hodnôt v poli OPT nám však postačí pamäť veľkosti $O(m)$: hodnoty v $OPT(i, j)$ sa dajú vypočítať iba pomocou hodnôt v predchádzajúcom riadku, takže si budeme uchovávať iba predchádzajúci a aktuálny riadok tohto poľa.

No a čo s poľom REK? To sa nám do pamäte zmestí. K uloženiu poľa REK použijeme nasledujúci trik: do jednej 32-bitovej (4-bajtovej) premennej uložíme až 16 hodnôt poľa REK, každú hodnotu pomocou 2 bitov. Takto potrebujeme k uloženiu poľa REK nanajvýš $(20\,000)^2/4$ bajtov, teda zhruba 100 MB pamäte.

Listing programu (C++)

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

double best[2][20000];
unsigned how[20000][1260];

int main() {
    int L, S, A, B;
    cin >> L >> S >> A >> B;

    vector<int> acka(A), becka(B);
    for (int &x:acka) cin >> x;
    for (int &x:becka) cin >> x;

    for (int a=0; a<A; ++a) {
        int cur=a%2, prev=1-cur;

        for (int b=0; b<B; ++b) {
            double   bestopt = 1e30;
            double   cdist   = sqrt(1.*(acka[a]-becka[b])*(acka[a]-becka[b]) + S*S);
            unsigned besthow = 0;

            if (a==0 && b==0)                          { bestopt=0;               besthow=0; }
            if (a>0 && best[prev][b] < bestopt)        { bestopt=best[prev][b];   besthow=1; }
            if (b>0 && best[cur][b-1] < bestopt)       { bestopt=best[cur][b-1];  besthow=2; }
            if (a>0 && b>0 && best[prev][b-1] < bestopt) { bestopt=best[prev][b-1]; besthow=3; }
            best[cur][b] = cdist + bestopt;
            how[a][b/16] |= besthow << ((b&15)<<1);
        }
    }

    int a=A-1, b=B-1, step;
    vector< pair<int,int> > answer;
    do {
        answer.push_back({a+1,b+1});
        step = (how[a][b/16] >> ((b&15)<<1)) & 3;
        a -= step&1;
        b -= step>>1;
    } while (step);

    cout << answer.size() << "\n";
    for (auto e:answer) cout << e.first << " " << e.second << "\n";
}
```

TRIDSIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Jozef Brandys, Michal Forišek, Askar Gafurov, Jaroslav Petrucha, Michaela Šandalová
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2015