A-III-1 Trojnohý beh

Podúloha A: najväčší súčet rýchlostí

Všimnime si najpomalšie zo všetkých detí, bez ujmy na všeobecnosti nech je to chlapec Jožko. S ktorým dievčaťom ho dať dokopy? Zjavne nič nepokazíme, keď bude tvoriť dvojicu s najpomalším dievčaťom. Prečo? Nech je Zuzka najpomalšie z dievčat. Majme ľubovoľné riešenie, v ktorom Jožko a Zuzka nebežia spolu, ale beží napríklad Jožko s Tamarou a Zuzka s Mirkom. Čo sa stane, keď to vymeníme a pobeží Jožko so Zuzkou a Tamara s Mirkom? Jožkova dvojica bude stále rovnako rýchla – totiž Jožko je najpomalší zo všetkých. No a Mirkova dvojica bude aspoň tak rýchla ako bola, lebo Tamara beží aspoň tak rýchlo ako Zuzka. Z toho teda vyplýva, že takouto výmenou nikdy nič nepokazíme. No a preto určite existuje optimálne riešenie, v ktorom Jožko a Zuzka bežia spolu.
Opakovaním tejto úvahy dostávame veľmi jednoduchý algoritmus: Na to, aby sme vyrobili dvojice, pre ktoré bude súčet rýchlostí maximálny, stačí každé pohlavie usporiadať podľa rýchlosti a potom popárovať do dvojíc, začínajúc od najpomalších. Takéto riešenie ľahko implementujeme v čase $O(n \log n)$.

Podúloha B: najmenší súčet rýchlostí

Aj v tejto podúlohe vieme spraviť skoro rovnakú úvahu ako v predchádzajúcej. Opäť, nech je Jožko najpomalšie zo všetkých detí. S kým má bežať, ak chceme, aby bol celkový súčet rýchlostí čo najmenší? Intuitívne, Jožka chceme „uviazať na krk“ čo najrýchlejšiemu dievčaťu. No a správnosť tejto intuície si vieme dokázať podobným argumentom ako v podúlohe A: výmenou. Tentokrát nech Katka je najrýchlejšie zo všetkých dievčat. Uvažujme ľubovoľné riešenie, v ktorom Jožko s Katkou nebežia spolu. Čo spraví výmena, ktorá ich dá dokopy?
Nech teraz beží Jožko s Tamarou a Mirko s Katkou. Keď to prehodíme a necháme bežať Jožka s Katkou, pobeží táto dvojica rovnako rýchlo ako Jožkova pôvodná, lebo Jožko to brzdí. No a dvojica Tamara-Mirko pobeží nanajvýš tak rýchlo ako dvojica Katka-Mirko, takže celkový súčet rýchlostí určite nestúpol.
Rovnako ako v predchádzajúcej podúlohe teda aj tu vieme spraviť riešenie v čase $O(n \log n)$. Opäť začneme tým, že zvlášť usporiadame chlapcov a zvlášť dievčatá, ale tentokrát ich budeme párovať naopak: i-ty najpomalší chlapec dostane k sebe i-te najrýchlejšie dievča.

Podúloha B šikovnejšie

Napriek tomu, že na prvý pohľad vyzerajú obe podúlohy rovnako, je medzi nimi zásadný rozdiel. V podúlohe A sa často môže stať, že existuje len jedno jediné optimálne popárovanie detí do dvojíc – to, ktoré vyrobí náš algoritmus. Toto v podúlohe B nie je pravda. Tam v každom veľkom vstupe existuje optimálnych riešení veľa. Dokonca tak veľa, že na nájdenie jedného z nich nebudeme potrebovať nič usporadúvať.
Všimnime si, že nech rozdelíme deti do dvojíc ľubovoľne, vždy bude platiť, že rýchlosti tých n dvojíc sú rovné rýchlostiam *niektorých* n detí. Určite preto nemôže existovať riešenie, v ktorom je súčet rýchlostí dvojíc menší ako súčet rýchlostí *najpomalších* n detí.
Predstavme si teraz, že sme všetkých $2n$ detí dokopy usporiadali podľa rýchlosti. Prvú polovicu poradia budeme volať pomalé deti a druhú rýchle deti. Všimnime si, že ak je pomalých chlapcov k, je pomalých dievčat $n - k$, a teda máme k rýchlych dievčat a $n - k$ rýchlych chlapcov. Určite teda vieme spraviť k dvojíc (pomalý chlapec)-(rýchle dievča) a $n - k$ dvojíc (rýchly chlapec)-(pomalé dievča). Nech to spravíme akokoľvek, súčet rýchlostí výsledných n dvojíc bude vždy rovnaký: bude rovný súčtu rýchlostí n pomalých detí. A ako sme si vyššie zdôvodnili, toto riešenie je určite optimálne.
Aby sme našli jedno optimálne riešenie, stačí teda vedieť rozdeliť deti na pomalú a rýchlu polovicu. Na to nepotrebujeme použiť triedenie, existujú efektívnejšie algoritmy, ktoré to vedia spraviť v lineárnom čase.
Asi najjednoduchší na implementáciu je algoritmus QuickSelect, ktorý beží v *očakávanom* náhodnom čase. Algoritmus funguje podobne ako triedenie QuickSort: v každej iterácii vyberieme jeden náhodný prvok ako pivota a v lineárnom čase preusporiadame prvky tak, aby všetky menšie od pivota boli skôr ako všetky väčšie od neho. Rozdiel bude v tom, že zatiaľ čo QuickSort vždy robil dve rekurzívne volania (jedno na prvky menšie od pivota, druhé na väčšie), QuickSelect vždy spraví len nanajvýš jedno z nich: v našom prípade by sme sa vždy zavolali len na tú časť poľa, v ktorej leží hranica medzi pomalými a rýchlymi deťami.

Príklad: Majme $n = 100$, teda dokopy 200 detí. Zvolíme jedno z nich ako pivota, preusporiadame pole a zistíme, že zvolené dieťa je v usporiadanom poradí na 47. mieste. QuickSort by teraz samostatne usporiadal najskôr prvých 46 detí a potom posledných 153. My to ale nemusíme robiť. Vieme totiž, že všetkých 46 prvých detí je pomalých a na ich presnom poradí nám nezáleží. Preto sa rekurzívne zavoláme len na posledných 153 detí.
(Dôkaz časovej zložitosti tohto algoritmu je náročný, preto ho neuvádzame. Intuitívne ide o to, že v priemere sa v každej iterácii algoritmu spracúvaná časť poľa dostatočne skráti.)
Existujú aj ďalšie jeho zlepšenia. Napríklad na `https://en.wikipedia.org/wiki/Median_of_medians` je popísaná úprava tohto algoritmu na deterministický algoritmus, ktorý aj v najhoršom prípade beží v lineárnom čase. Úprava spočíva v tom, že namiesto náhodného výberu budeme pivota vyberať šikovnejšie – tak, aby sme zaručili, že nám prvky rozdelí na dve podobne veľké časti.

Listing programu (C++)

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

int main() {
    int N;
    cin >> N;
    vector<int> chlapci(N);
    for (int &x : chlapci) cin >> x;
    vector<int> dievcata(N);
    for (int &x : dievcata) cin >> x;

    // najdeme najvacsi mozny sucet rychlosti v case O(n log n)
    sort( chlapci.begin(), chlapci.end() );
    sort( dievcata.begin(), dievcata.end() );
    int odpoved = 0;
    for (unsigned i=0; i<chlapci.size(); ++i) odpoved += min( chlapci[i], dievcata[i] );
    cout << odpoved << endl;
}
```

Listing programu (C++)

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

int main() {
    int N;
    cin >> N;
    vector<int> deti(2*N);
    for (int &x : deti) cin >> x;

    // najdeme najmensi mozny sucet rychlosti v case O(n)
    nth_element( deti.begin(), deti.begin()+N, deti.end() );
    int odpoved = accumulate( deti.begin(), deti.begin()+N, 0 );
    cout << odpoved << endl;
}
```

A-III-2 Elektromobil

Je zjavné, že všetky riešenia tejto úlohy budú založené na nejakom prehľadávaní nejakého grafu.
Úplne základné riešenie úlohy môžeme založiť na pozorovaní, že keď si tak jazdíme autom po krajine, v ľubovoľnom meste vieme popísať našu aktuálnu situáciu dvomi číslami. Jedno je číslo dotyčného mesta a druhé hovorí, na koľko presunov máme ešte nabitú batériu.
Na jazdenie po krajine sa teda môžeme dívať ako na prechádzanie sa po grafe, ktorého vrcholmi sú všetky možné stavy, v ktorých sa môžeme nachádzať. (Každý vrchol je teda nejaká dvojica čísel.) Hrany v tomto grafe zodpovedajú akciám, ktoré môžeme robiť. Z každého stavu, v ktorom nemáme vybitú batériu, teda vedú hrany predstavujúce akciu „prejdi do susedného mesta a zmenši o 1 nabitosť batérie“. Zo stavov, v ktorých sme v meste s dobíjacou stanicou, vedú navyše hrany predstavujúce akciu „zostaň v tomto meste a dobi si batériu doplna“.
Takýto graf má $O(nk)$ vrcholov a $O(mk + dk)$ hrán. Úlohu vieme vyriešiť tým, že jeho prehľadávaním nájdeme všetky stavy dosiahnuteľné zo začiatočného (t.j. zo stavu „sme v meste a, máme plnú nádrž“) a zistíme, či je medzi nimi ľubovoľný stav, v ktorom sme v meste b. Takéto riešenie má časovú zložitosť $O(k(n + m + d)) = O(k(n + m))$.

Existujú rôzne optimalizácie tohto riešenia. Jednou z možností je uvedomiť si, že niektoré stavy sú zbytočné: ak vieme byť v meste c a mať batériu nabitú na ešte 7 presunov, vôbec nás nezaujíma, že sa iným spôsobom vieme dostať do mesta c a mať batériu nabitú na 3 presuny. Totiž čokoľvek, čo vieme spraviť z toho druhého stavu, vieme spraviť aj z prvého. Toto pozorovanie vedie k riešeniu, pri ktorom si pre každé mesto postupne počítame s ako najviac nabitou batériou v ňom vieme byť. Ak navyše počas prehľadávania vždy spracujeme ten nespracovaný stav, v ktorom máme práve najviac nabitú batériu, dá sa ukázať, že prehľadáme iba $O(n \min(\sqrt{n}, k))$ stavov a časová zložitosť algoritmu pri šikovnej implementácii bude $O((m+n)\min(\sqrt{n}, k))$.
Iná optimalizácia sa dá použiť, ak je počet d dobíjacích staníc malý. V takomto prípade vieme samostatne spustiť prehľadávanie do šírky z každej dobíjacej stanice a zistiť, kam všade sa z nej vieme dostať bez dobíjania. Takto si zostrojíme nový graf, ktorého vrcholmi už budú len dobíjacie stanice. Hrany tohto nového grafu budú hovoriť, odkiaľ kam sa vieme dostať bez dobíjania. Pôvodnú úlohu vieme vyriešiť na tri ďalšie prehľadávania: V pôvodnom grafe zistíme, na ktoré stanice (množina A) sa vieme dostať z mesta a a následne z ktorých staníc (množina B) sa vieme dostať do mesta b. Na novom grafe čerpacích staníc následne zistíme, či sa vieme dostať z množiny A do množiny B. Takéto riešenie má časovú zložitosť $O(d(n+m))$.

Vzorové riešenie

Na záver si ukážeme optimálne riešenie úlohy. Jeho časová zložitosť bude lineárna od veľkosti grafu, teda $O(n+m)$. Podotýkame, že časová zložitosť nebude závisieť od d ani k.
Základná myšlienka riešenia je nasledovná. Predstavme si, že samostatne pre každú dobíjaciu stanicu (a tiež pre mesto a, kde začíname) niekto zobral vedro s nejakou novou farbou a ofarbil ňou všetko, čo leží vo vzdialenosti nanajvýš $k/2$ od dotyčného miesta. Niektoré cesty teda môžu byť ofarbené len do polovice, ak je k nepárne. Niektoré cesty môžu mať naraz aj viacero farieb. Zjednotenie všetkých oblastí ofarbených jednotlivými farbami budeme volať *ofarbená oblasť*.
Akonáhle takto ofarbíme našu krajinu, vieme ľahko povedať, kedy sa dá prejsť z jednej dobíjacej stanice priamo na druhú: zjavne je to práve vtedy, keď sa ich farebné oblasti dotýkajú alebo prekrývajú.
Tak isto by malo byť zjavné, že ak niekedy počas nášho cestovania po krajine prídeme na cestu, ktorá nemá žiadnu farbu, už sa nikdy nedostaneme na žiadnu dobíjaciu stanicu. (Museli sme od poslednej stanice prejsť viac ako $k/2$, aby sme sa dostali von z *jej* ofarbenej oblasti, takže už máme batériu nabitú na menej ako $k/2$. A keďže sme na neofarbenej ceste, v okolí $k/2$ okolo nás nie je žiadna dobíjacia stanica.
Ak teda existuje cesta z a do b, musí vyzerať nasledovne: Vyrazím z a, jazdím po krajine a navštevujem dobíjacie stanice bez toho, aby som opustil ofarbenú oblasť. Po poslednom dobití batérie prejdem do b, pričom už môžem ofarbenú oblasť opustiť.

V tomto okamihu spravíme druhé pozorovanie, ktoré bude dôležité pre riešenie našej úlohy: Predstavme si, že sme začali vo vrchole a nášho grafu a ľubovoľne sme sa po grafe prechádzali, pričom sme ale nikdy neopustili ofarbenú oblasť. Potom ľubovoľné miesto, ktoré vieme dosiahnuť takouto prechádzkou, vieme dosiahnuť aj naším autom (ktoré treba pravidelne nabíjať).
Dôkaz: V každom okamihu prechádzky ideme po hrane nejakej farby. (Ak máme hranu, ktorá leží vo viacerých farebných oblastiach, vyberieme si pre ňu ľubovoľnú jednu z tých farieb.) Prechádzke teda zodpovedá nejaká postupnosť farieb. Nech teraz f_1, f_2 je ľubovoľná dvojica po sebe idúcich farieb v tejto postupnosti. Keďže sme vedeli prejsť z farby f_1 na farbu f_2, oblasti farieb f_1 a f_2 musia spolu susediť (prípadne sa prekrývať). Určite teda vieme sadnúť do nášho auta a postupne navštevovať dobíjacie stanice v poradí zodpovedajúcom poradiu farieb na našej prechádzke. No a na záver nech prechádzka skončila kdekoľvek v oblasti poslednej farby, dotyčné miesto je určite dosiahnuteľné z príslušnej dobíjacej stanice.

Práve dokázané tvrdenie je veľmi dôležité, lebo práve vďaka nemu budeme vedieť zlepšiť časovú zložitosť. Toto tvrdenie nám totiž hovorí, že vôbec nezáleží na jednotlivých farbách. Ak chceme zistiť, ktoré miesta v našom grafe sú a ktoré nie sú dosiahnuteľné, stačí nám naraz zostrojiť celú ofarbenú oblasť.
A toto je práve miesto, kde vieme ušetriť čas. Keby sme naozaj zvlášť pre každú dobíjaciu stanicu ofarbili jej oblasť, v najhoršom prípade by sme až d-krát ofarbili celý graf, časová zložitosť by teda bola $\Theta(d(n+m))$. Keď však vieme, že na jednotlivých farbách nezáleží, máme omnoho lepšiu možnosť. Na ofarbovanie použijeme len jedno jediné prehľadávanie do šírky, ktoré však spustíme naraz zo všetkých dobíjacích staníc. Pritom si dáme

pozor na to, aby sme každú cestu ofarbili len raz. Dokopy teda nanajvýš raz spracujeme každú hranu nášho grafu, čiže celé ofarbovanie stihneme v čase $O(n + m)$.

Na záver už uvedieme len jeden implementačný detail: Aby sme nemuseli pre nepárne k hrany ofarbovať do polovice, vložíme si do stredu každej hrany jeden nový vrchol a hodnotu k vynásobíme dvomi. Takto dostaneme graf s $n + m$ vrcholmi a $2m$ hranami, takže časovú zložitosť nám to nepokazí.

Listing programu (C++)

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

struct hrana { int x,y; bool ofarbena; };

int N, M, D, K, A, B;
vector<int> stanice;
vector<hrana> hrany;
vector< vector<int> > G; // pre každý vrchol zoznam čísel hrán idúcich z neho

// pomocná funkcia: ak má hrana [h] jeden koniec vo vrchole [kde], kde je druhý koniec?
int druhy_koniec(int h, int kde) { return hrany[h].x ^ hrany[h].y ^ kde; }

// začínajúc vo vrcholoch [odkial], ofarbi všetky hrany do vzdialenosti [maxdist]
void ofarbi(const vector<int> &odkial, int maxdist) {
    vector<int> vzdialenost( M+N, maxdist+1 );
    queue<int> Q;
    for (int x : odkial) { vzdialenost[x] = 0; Q.push(x); }

    while (!Q.empty()) {
        int kde = Q.front(); Q.pop();
        if (vzdialenost[kde] == maxdist) continue;
        for (int h : G[kde]) {
            int kam = druhy_koniec(h,kde);
            hrany[h].ofarbena = true;
            if (vzdialenost[kam] == maxdist+1) {
                vzdialenost[kam] = vzdialenost[kde]+1;
                Q.push(kam);
            }
        }
    }
}

// začínajúc vo vrchole [odkial], nájdi všetky stanice dosiahnuteľné po ofarbených hranách
vector<int> najdi_dosiahnutelne(int odkial) {
    vector<bool> navstivil( M+N, false );
    queue<int> Q;
    navstivil[odkial] = true;
    Q.push(odkial);

    while (!Q.empty()) {
        int kde = Q.front(); Q.pop();
        for (int h : G[kde]) {
            if (!hrany[h].ofarbena) continue;
            int kam = druhy_koniec(h,kde);
            if (navstivil[kam]) continue;
            navstivil[kam] = true;
            Q.push(kam);
        }
    }

    vector<int> odpoved;
    for (int s : stanice) if (navstivil[s]) odpoved.push_back(s);
    return odpoved;
}

int main() {
    // načítame vstup
    cin >> N >> M >> D >> K >> A >> B;

    stanice.resize(D);
    for (int &s : stanice) cin >> s;
    stanice.push_back(A); // aj keby v A nebola stanica, na začiatku tam sme a máme plnú batériu

    G.resize( N+M );
    for (int m=0; m<M; ++m) {
        int x, y;
        cin >> x >> y;
        // namiesto pôvodnej hrany x-y pridáme dve nové cez pomocný vrchol
        hrany.push_back( {x,N+m,false} ); // hrana číslo 2*m
        hrany.push_back( {y,N+m,false} ); // hrana číslo 2*m+1
        G[x].push_back(2*m);
        G[N+m].push_back(2*m);
        G[y].push_back(2*m+1);
        G[N+m].push_back(2*m+1);
```

```
    }

    ofarbi(stanice,K); // ofarbi všetko do vzdialenosti K v novom grafe (čiže K/2 v pôvodnom)
    vector<int> dobre_stanice = najdi_dosiahnutelne(A); // nájdi stanice, kam sa dá dostať po ofarbených hranách
    for (auto &h : hrany) h.ofarbena = false;
    ofarbi(dobre_stanice,2*K); // ofarbi všetko do vzdialenosti 2*K od dobrých staníc

    // zisti, či vieme dosiahnuť B
    bool vieme = false;
    for (int h : G[B]) if (hrany[h].ofarbena) vieme = true;
    cout << (vieme ? "ano\n" : "nie\n");
}
```

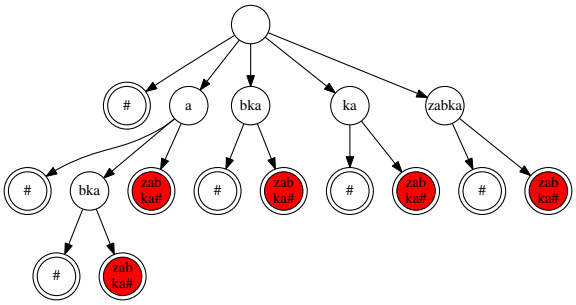
A-III-3 Sufixové stromy

Podúloha A: zakázané podreťazce

V prvej podúlohe si stačilo uvedomiť, že nepotrebujeme porovnávať samostatne každý reťazec s každým iným. Šikovnejšie riešenie bude vyzerať nasledovne:
Z našich reťazcov si v lineárnom čase (presnejšie, v čase lineárnom od súčtu ich dĺžok) vyrobíme jeden dlhý reťazec $T = S_1\#S_2\#\ldots\#S_n$. Znak # je „zarážka“ – symbol, ktorý sa v našich reťazcoch nenachádza. Pre tento reťazec si potom postavíme sufixový strom a ten následne predspracujeme algoritmom, ktorý je uvedený v študijnom texte v príklade 2.
A tým sme už vlastne vyhrali. Jediné, čo ostáva, je postupne pre každé i overiť, že má reťazec S_i v reťazci T práve jeden výskyt. Pre konkrétne i na túto kontrolu spravíme počet krokov priamo úmerný dĺžke S_i. (Všimnite si, že tento počet krokov nezávisí od dĺžky celého T.) Dokopy všetky kontroly teda zbehnú v čase lineárnom od veľkosti vstupu.

Podúloha B: cyklické posuny

Predstavme si, že z koreňa sufixového stromu nejakého reťazca $S\#$ spustíme prehľadávanie do hĺbky. Toto prehľadávanie navyše implementujeme tak, že vždy, keď z vrcholu vedie dodola viacero hrán, spustíme sa na ne v abecednom poradí. (Hrana začínajúca znakom # je skôr v abecede ako všetky písmená.) Ľahko nahliadneme, že toto prehľadávanie navštívi konce všetkých sufixov v ich lexikografickom poradí.
Toto by mohlo nejak súvisieť so súťažnou úlohou. Každý cyklický posun pôvodného reťazca totiž začína nejakým jeho sufixom. Ako však vidíme napríklad aj na príklade v zadaní, poradie sufixov nemusí zodpovedať poradiu cyklických posunov. Napr. u reťazca `zabka` je najmenším sufixom sufix `a`, ale jemu zodpovedajúci cyklický posun `azabk` je v lexikografickom poradí až druhý, za reťazcom `abkaz`.
Ako si s týmto problémom poradiť? Najjednoduchším riešením je drobný trik: namiesto reťazca S sa pozrieme na reťazec SS, teda na dve kópie S za sebou. Nech n je dĺžka reťazca S. Potom cyklickým posunom reťazca S zodpovedá n najdlhších sufixov reťazca SS: každý z týchto sufixov začína jedným z cyklických posunov S.



Označeným vrcholom v poradí zľava doprava zodpovedajú cyklické posuny `abkaz`, `azabk`, `bkaza`, `kazab` a `zabka`.

Kompletné riešenie teda bude vyzerať nasledovne: postavíme si sufixový strom pre reťazec $SS\#$ a v ňom

nasledovne prehľadávaním do hĺbky nájdeme k-ty „najľavejší“ spomedzi koncov dostatočne dlhých sufixov. Časová zložitosť tohto riešenia je zjavne lineárna od dĺžky S.

Listing programu (Python)

```
from modelA_tree import vyrob_strom

S = input()
strom = vyrob_strom(S+S+'#')
kolko_este = int( input() )

aktualna_cesta = []

def dfs(kde, aktualna_hlbka):
    global S, strom, kolko_este

    # ak na tomto mieste konci dostatocne dlhy sufix, zaratame ho
    if kde.koniec and aktualna_hlbka > len(S)+1:
        kolko_este -= 1
        if kolko_este == 0:
            riesenie = ''
            for hrana in aktualna_cesta:
                riesenie += strom.retazec[ hrana.od : hrana.po ]
            print( riesenie[ : len(S) ] )
            return

    # v lexikografickom poradi prezerame deti,
    # skoncime ked najdeme dost vela spravnych sufixov
    for x in sorted( kde.deti.keys() ):
        hrana = kde.deti[x]
        aktualna_cesta.append( hrana )
        dfs( hrana.kam, aktualna_hlbka + hrana.po - hrana.od )
        aktualna_cesta.pop()
        if kolko_este == 0: break

dfs( strom.koren, 0 )
```

TRIDSIATY PRVÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Eduard Batmendijn, Michal Forišek, Marián Horňák, Jaroslav Petrucha
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2016