A-III-1 Vodovody

Úsek miest, ktoré sú spojené dokopy vodovodom a majú v súčte nezápornú produkciu budeme v tomto vzorovom riešení volať *okres*. Cena okresu bude rovná súčtu cien potrubí, ktoré ho spájajú dokopy. Naším cieľom je teda rozdeliť kráľovstvo na niekoľko okresov tak, aby súčet ich cien bol najmenší možný.

Určenie produkcie úseku miest a ceny ich spojenia

Počas hľadania optimálneho riešenia sa nám určite zíde vedieť efektívne určiť pre daný úsek miest, či majú v súčte nezápornú produkciu (t.j. či môžu tvoriť okres) a ak áno, koľko nás bude stáť ich spojenie dokopy.

Na toto sú výborným nástrojom *prefixové súčty*. Označme $\hat{p}(i)$ celkovú produkciu prvých i miest a $\hat{c}(i)$ cenu za postavenie prvých i ciest. Špeciálne teda $\hat{p}(0) = \hat{c}(0) = 0$. Prefixové súčty vieme ľahko vypočítať v lineárnom čase: napr. konkrétnu hodnotu $\hat{p}(i)$ vieme určiť ako $\hat{p}(i-1) + p_i$.

Pomocou týchto prefixových súčtov vieme potrebné údaje pre ľubovoľný úsek vypočítať v konštantnom čase: Majme úsek tvorený mestami od a po b vrátane. Celkovú produkciu tohoto úseku môžeme vyjadriť ako $\hat{p}(b) - \hat{p}(a-1)$. Ak je táto hodnota nezáporná, má zmysel sa pozrieť na to, koľko by nás stálo z daného úseku vyrobiť okres. Túto cenu (t.j. celkovú cenu za postavenie správnych $b-a$ kusov potrubia) vieme vypočítať ako $\hat{c}(b-1) - \hat{c}(a-1)$.

Riešenie v kvadratickom čase

Všetkých možných rozdelení kráľovstva na okresy môže byť až exponenciálne veľa v závislosti od počtu miest. Dobré riešenie teda nemôžeme založiť na skúšaní všetkých možností. Presnejšie, budeme to skúšanie musieť robiť efektívnejšie.

Predstavme si, že sme sa už rozhodli, koľko miest bude tvoriť posledný okres – teda ten obsahujúci mesto n. Takéto rozhodnutie nám rozdelí kráľovstvo na dva úseky: prvý tvorený mestami 1 až x, druhý mestami $x+1$ až n, pre nejaké x. Celý druhý úsek bude jeden okres, zatiaľ čo prvý úsek môžeme ešte skúsiť ďalej deliť na viacero menších okresov.

Samozrejme, niektoré voľby x nemusia viesť k platným riešeniam. To nahliadneme ľahko – rozdeliť kráľovstvo medzi mestami x a $x+1$ môžeme len vtedy, ak má aj úsek od 1 po x, aj úsek od $x+1$ po n nezápornú celkovú produkciu.

A akonáhle sme zvolili x, dostávame ten istý problém ako na začiatku, len už s kratšou postupnosťou: „Ako najlacnejšie vieme rozdeliť do okresov postupnosť tvorenú prvými x mestami?“

Túto otázku môžeme samozrejme zodpovedať vhodným rekurzívnym volaním nášho programu. Takto ľahko dostaneme rekurzívne riešenie s exponenciálnou časovou zložitosťou, ktoré skúša všetky možné spôsoby rozdelenia kráľovstva na okresy. Schéma takéhoto riešenia:

```
najlepsie(z):
    odpoved = cena pospajania 1..z (alebo nekonecno ak mesta 1..z netvoria okres)
    pre kazde x od 1 po z-1:
        ak mozeme 1..z rozdelit na 1..x a (x+1)..z:
            toto = najlepsie(x) + cena pospajania (x+1)..z
            odpoved = min(odpoved, toto)
    return odpoved

riesenim ulohy je najlepsie(n)
```

No a od takéhoto pomalého rekurzívneho riešenia je už len krok ku riešeniu efektívnemu. Stačí nepočítať tú istú vec zbytočne veľakrát. Presnejšie, stačí si všimnúť, že celý výpočet nášho programu spočíva v tom, že dokola voláme funkciu `najlepsie` s rôznymi parametrami – ale tento parameter nadobúda len hodnoty 1 až n. Dokola teda znova a znova zbytočne počítame tie isté hodnoty. Omnoho efektívnejšie bude použiť *memoizáciu*: akonáhle raz zistíme, že napr. `najlepsie(30)` vráti hodnotu 4700, zapamätáme si to a všetky ďalšie volania `najlepsie(30)` už prebehnú v konštantnom čase – len vrátime zapamätanú hodnotu.

Takéto riešenie má zjavne časovú zložitosť $\Theta(n^2)$: pre každé z od 1 po n sa len raz vykoná telo funkcie `najlepsie(z)`, ktorého časová zložitosť je lineárna od z.

Listing programu (C++)

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

const int NEVIEM = -1;
const long long NEKONECNO = 1LL<<60;
int N;                              // počet miest
vector<long long> P, C, sP, sC; // produkcie miest, ceny potrubí a ich prefixové súčty
vector<long long> memo;         // zapamätané hodnoty funkcie najlepsie()

vector<long long> prefixove_sucty(const vector<long long> &V) {
    vector<long long> sV(1,0);
    for (long long x : V) sV.push_back( sV.back()+x );
    return sV;
}

bool moze_byt_okres(int a, int b) { return ( sP[b] - sP[a-1] ) >= 0; } // môže úsek miest a..b tvoriť okres?
long long cena_okresu(int a, int b) { return sC[b-1] - sC[a-1]; } // koľko ma stojí pospájať mestá a..b?

long long najlepsie(int z) {
    if (memo[z] == NEVIEM) {
        memo[z] = NEKONECNO;
        if (moze_byt_okres(1,z)) memo[z] = cena_okresu(1,z);
        for (int x=1; x<z; ++x)
            if (moze_byt_okres(1,x) && moze_byt_okres(x+1,z))
                memo[z] = min( memo[z], najlepsie(x) + cena_okresu(x+1,z) );
    }
    return memo[z];
}

int main() {
    // načítame vstup
    cin >> N;
    P.resize(N);   for (long long &p : P) cin >> p;
    C.resize(N-1); for (long long &c : C) cin >> c;
    sP = prefixove_sucty(P);
    sC = prefixove_sucty(C);
    memo.resize(N+1,NEVIEM);
    cout << najlepsie(N) << endl;
}
```

Ekvivalentné riešenie vieme zapísať aj bez rekurzie využitím *dynamického programovania*: hodnoty, ktoré si predchádzajúce riešenie počas výpočtu zapamätalo, budeme počítať v cykle s rastúcou hodnotou z.

Listing programu (C++)

```
// začiatok programu je rovnaký ako v predchádzajúcom listingu
int main() {
    cin >> N;
    P.resize(N);   for (long long &p : P) cin >> p;
    C.resize(N-1); for (long long &c : C) cin >> c;
    sP = prefixove_sucty(P);
    sC = prefixove_sucty(C);
    memo.resize(N+1,NEVIEM);
    for (int z=1; z<=N; ++z) {
        memo[z] = NEKONECNO;
        if (moze_byt_okres(1,z)) memo[z] = cena_okresu(1,z);
        for (int x=1; x<z; ++x)
            if (moze_byt_okres(1,x) && moze_byt_okres(x+1,z))
                memo[z] = min( memo[z], memo[x] + cena_okresu(x+1,z) );
    }
    cout << memo[N] << endl;
}
```

Vzorové riešenie

Ukážeme si teraz, ako zlepšiť časovú zložitosť predchádzajúceho riešenia. Základná myšlienka zlepšenia: pri počítaní najlepšieho riešenia pre mestá 1 až z nebudeme skúšať všetky x od 1 po $z-1$, ale použijeme vhodnú dátovú štruktúru, ktorá nám efektívne povie najlepšiu spomedzi všetkých použiteľných hodnôt x.

V prvom rade potrebujeme vedieť spomedzi všetkých x vybrať tie, ktoré zodpovedajú prípustnému rozdeleniu miest 1 až z na posledný úsek a zvyšok. Inými slovami, tie x, pre ktoré majú aj úsek 1 až x, aj úsek $x+1$ až

z nezápornú celkovú produkciu. Pomocou prefixových súm môžeme tieto podmienky prepísať nasledovne: musí platiť $\hat{p}(x) \geq 0$ a zároveň $\hat{p}(z) \geq \hat{p}(x)$.

Spomedzi týchto x hľadáme to, ktoré nám dá najlepší výsledok – teda najmenší súčet už spočítaného riešenia pre úsek 1 až x a ceny za pospájanie úseku od $x+1$ po z.

Tu sa ale črtá možný problém. Nám by sa hodilo mať dátovú štruktúru, do ktorej len vkladáme nové záznamy a pýtame sa jej. Lenže hodnota, ktorú sa snažíme minimalizovať, nie je konštantná, ale závisí od aktuálnej hodnoty z. Čo s tým?

Našťastie sa obávame zbytočne, žiaden problém tu nie je. Všimnime si totiž, čo sa stane, keď prejdeme z nejakej hodnoty z na nasledujúcu, o jedno väčšiu. Pre *úplne všetky* možné výbery x cena za pospájanie posledného úseku narastie o tú istú hodnotu c_z (t.j. cenu za spojenie miest z a $z+1$). Relatívne poradie jednotlivých možností sa teda nezmení.

V našej implementácii to vyriešime nasledovne: Predstavme si, že sme práve spočítali, že najlacnejší spôsob, ako pospájať prvých z miest, má cenu φ. Do našej dátovej štruktúry si teraz uložíme pre index z cenu $\varphi + \hat{c}(n-1) - \hat{c}(z)$. Slovne: k cene za optimálne riešenie pre mestá 1 až z pripočítame cenu za spojenie všetkých ostatných miest ($z+1$ až n) dokopy. Z uloženej ceny vieme ľahko v konštantnom čase určiť cenu pre ľubovoľný iný, kratší úsek pospájaných miest na konci.

V pseudokóde si teda naše nové riešenie môžeme zapísať nasledovne:

```
zober prazdnu datovu strukturu D

pre kazde z od 1 po n:
    odpoved = cena pospajania 1..z (alebo nekonecno ak mesta 1..z netvoria okres)
    v D najdi pripustny zaznam s najmensou cenou
    ak si taky zaznam nasiel:
        vypocitaj z jeho ceny cenu len pre mesta 1..z
        uloz ju do odpoved[z]
    vloz do D zaznam o odpoved[z]
```

Záznamy v našej dátovej štruktúre budú mať dve zložky: prefixovú sumu $\hat{p}(z)$, aby sme vedeli, kedy danú hodnotu môžeme neskôr použiť, a hodnotu $odpoved[z] + \hat{c}(n-1) - \hat{c}(z)$, ktorú sa snažime minimalizovať.

Existuje viacero možných implementácií, napr. pomocou intervalového stromu: usporiadame si všetky hodnoty prefixových súm $\hat{p}$ a následne nad nimi postavíme intervalový strom, do ktorého budeme vkladať upravené vypočítané odpovede.

My sme si zvolili implementáciu pomocou množiny – vyvažovaného stromu, v ktorom si pamätáme vyššie popísané záznamy ako usporiadané dvojice – ale pamätáme si len tie, ktoré majú teoretickú šancu niekedy byť použité. Akonáhle teda vkladáme záznam (a, b), zmažeme všetky záznamy (c, d) pre ktoré súčasne platí $c \geq a$ a $d \geq b$ – totiž kedykoľvek, kedy by sme mohli použiť záznam (c, d), môžeme použiť aj záznam (a, b) a ten je lepší. Takto dosiahneme, že postupnosť dvojíc uložených v našej množine je *súčasne* usporiadaná vzostupne podľa prefixovej sumy aj zostupne podľa odpovede.

Ak teda napr. máme v nejakom okamihu v našej dátovej štruktúre uložené dvojice $(2, 10)$ a $(4, 7)$, môžeme to chápať nasledovne: kedykoľvek, keď v budúcnosti budeme spracúvať nejaké z, pre ktoré $\hat{p}(z) \geq 2$, existuje k nemu x, pomocou ktorého optimálne riešenie pre aktuálne z vypočítame z čísla 10 odpočítaním vhodnej konštanty. A ak dokonca platí $\hat{p}(z) \geq 4$, tak vieme mať ešte o 3 lepšie riešenie pre aktuálne z.

Aká je časová zložitosť tohto riešenia? Do usporiadanej množiny vložíme nanajvýš n záznamov a každý z nich nanajvýš raz zmažeme. Navyše nanajvýš n-krát budeme v našej množine vyhľadávať najlepšiu použiteľnú dvojicu. Každú z týchto operácií vieme spraviť v čase $O(\log n)$, celková časová zložitosť je teda $O(n \log n)$.

Listing programu (C++)

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

const long long NEKONECNO = 1LL<<60;
int N;                          // počet miest
```

```
vector<long long> P, C, sP, sC; // produkcie miest, ceny potrubí a ich prefixové súčty
vector<long long> odpoved;      // odpoved[z] je cena optimálneho riešenia pre mestá 1..z

vector<long long> prefixove_sucty(const vector<long long> &V) {
    vector<long long> sV(1,0);
    for (long long x : V) sV.push_back( sV.back()+x );
    return sV;
}

bool moze_byt_okres(int a, int b) { return ( sP[b] - sP[a-1] ) >= 0; } // môže úsek miest a..b tvoriť okres?
long long cena_okresu(int a, int b) { return sC[b-1] - sC[a-1]; } // koľko ma stojí pospájať mestá a..b?

typedef pair<long long, long long> zaznam;
set<zaznam> D;                  // dátová štruktúra, v ktorej máme naše záznamy

void vloz(long long a, long long b) {
    // overíme, či nemáme lepšiu alebo rovnú možnosť ako (a,b)
    auto it = D.lower_bound(zaznam(a+1,0));
    --it;
    if (it->second <= b) return;
    // vyhádžeme možnosti od ktorých je (a,b) lepšie
    ++it;
    while (it != D.end() && it->second >= b) { auto jt=it; ++it; D.erase(jt); }
    // pridáme (a,b)
    D.insert(zaznam(a,b));
}

int main() {
    cin >> N;
    P.resize(N);   for (long long &p : P) cin >> p;
    C.resize(N-1); for (long long &c : C) cin >> c;
    sP = prefixove_sucty(P);
    sC = prefixove_sucty(C);
    odpoved.resize(N+1,NEKONECNO);
    D.insert(zaznam(0,NEKONECNO)); // zarážka
    for (int z=1; z<=N; ++z) if (moze_byt_okres(1,z)) {
        odpoved[z] = cena_okresu(1,z);

        // nájdi v D lepšiu možnosť
        auto it = D.lower_bound(zaznam(sP[z]+1,0));
        --it;
        if (it->second != NEKONECNO) odpoved[z] = it->second - cena_okresu(z,N);

        // vlož do D práve spočítanú odpoveď
        vloz( sP[z], odpoved[z] + cena_okresu(z+1,N) );
    }
    cout << odpoved[N] << endl;
}
```

A-III-2 Dievka na vydaj

Malé hodnoty k

Pozrime sa najskôr na riešenia, ktoré využívajú fakt, že k je relatívne malé (do 10^5). Pre $n = 2$ vieme použiť prístup ako v mergesorte. Na začiatku mám dva ukazovatele, oba nastavené na začiatok k-tic. Následne zistím, ktorý z prvkov, na ktoré ukazujú, je väčší a to bude najbohatší pytač. Daný ukazovateľ posuniem o jedna dozadu. Tento postup zopakujem dokopy k-krát a posledné vybraté číslo bude pytač, ktorého hľadáme. Takto dosiahneme časovú zložitosť $O(k)$.

Ako to zovšeobecníme pre $n > 2$? Vtedy si musíme udržiavať n ukazovateľov a v každej iterácii zistiť maximum zo všetkých prvkov na ktoré ukazujú. Dôležité je uvedomiť si, že aj keď nepoznáme presné hodnoty prvkov, na ktoré ukazujeme, vieme ich porovnávať. To znamená, že vieme upraviť všetky algoritmy, ktoré fungujú na princípe porovnávania tak, aby fungovali aj pre náš problém. V tomto prípade sa nám oplatí použiť haldu. Nad našimi n ukazovateľmi si teda postavíme maximovú haldu, z ktorej v každej iterácii vyberieme najväčší prvok, daný ukazovateľ posunieme o jedno dozadu a tento nový prvok vložíme spať do haldy. To nám zabezpečí časovú zložitosť $O(k \log n)$.

Veľké hodnoty k

Zo zadania však vyplýva, že by bolo dobré vedieť riešiť túto úlohu aj pre veľké hodnoty k, preto nechceme, aby časová zložitosť nášho programu bola lineárna od k. Opäť začnime prípadom, keď $n = 2$. Povedzme si, že

k-ty najväčší prvok sa nachádza v prvej skupine. A to, ktorý konkrétny prvok to je, budeme binárne vyhľadávať. Povieme si, že x-tý najväčší prvok v prvej skupine bude výsledok. Ako to overíme? Máme dve možnosti, buď v druhej skupine binárne dohľadáme, ktorý prvok je najmenší väčší ako prvok x, a overíme, či je to prvok $k - x$, alebo sa rovno spýtame, či $(k - x)$-tý prvok je najmenší väčší ako x a v konštantnom čase overíme. To nám dá zložitosť $O(\log^2 k)$, respektíve $O(\log k)$.

Skúsme teraz zovšeobecniť toto riešenie aj pre n skupín. Postupne vyskúšame n možností pre to, v ktorej skupine leží výsledok. Následne budeme výsledok binárne vyhľadávať a overovať to tak, že vo zvyšných skupinách binárne vyhľadám prvok, ktorý je najmenší väčší ako ten, čo sme si zvolili. Z toho spočítam celkový počet prvkov väčších alebo rovných práve skúšanému a to porovnám so želanou hodnotou k. Toto nám zaručí časovú zložitosť $O(n^2 \log^2 k)$.

Ako si pamätať, čo sme už zistili

Predchádzajúce riešenie má zjavnú slabinu: pre každú skupinu začíname hľadanie odznova a nevyužívame pri tom, čo sme sa už skôr dozvedeli. Ako to ale využiť vieme?

Budeme si udržiavať množinu *kandidátov* – princov, ktorí ešte môžu byť k-tym najbohatším zo všetkých. Na začiatku množinu kandidátov zjavne tvorí prvých k princov z každej skupiny. Ako sa budeme dozvedať nové informácie, bude sa množina kandidátov zmenšovať, až v nej nakoniec ostane len jediný princ. Navyše, ak o nejakom princovi zistíme, že medzi kandidátov nepatrí, lebo je príliš chudobný, nebudú medzi kandidátov patriť ani princovia z jeho skupiny, ktorí sú za ním v poradí – tí sú ešte chudobnejší. A podobne, ak je nejaký princ príliš bohatý, môžeme spolu s ním vylúčiť všetkých z jeho skupiny, ktorí sú od neho bohatší. Množina kandidátov sa nám teda bude pamätať ľahko: v každej zo skupín budú kandidáti vždy tvoriť súvislý úsek.

Zmenšovať množinu kandidátov vieme napríklad postupom, ktorý sme si už načrtli vyššie, ale pre poriadok ho popíšeme znova a poriadne: Vyberieme si nejakého princa p. Zadarmo vieme, koľko princov v jeho skupine je od neho bohatších. Pre každú zo zvyšných $n - 1$ skupín môžeme použiť binárne vyhľadávanie a v čase $O(\log k)$ určiť, koľko princov v danej skupine je od princa p bohatších. Takže v celkovom čase $O(n \log k)$ vieme zistiť presné poradie princa p.

No a v tomto okamihu nastávajú tri možné prípady. Ak máme šťastie a princ p je presne k-ty, tak sme vyriešili úlohu a končíme. Ak je princ p chudobnejší ako k-ty, môžeme princa p a všetkých chudobnejších od neho vylúčiť spomedzi kandidátov. (Vďaka binárnym vyhľadávaniam, ktoré sme práve dokončili, vieme v každom riadku, ktorých princov ideme vylúčiť.) No a ak je princ p príliš bohatý, vylúčime spomedzi kandidátov okrem p aj všetkých princov bohatších od neho.

Teraz by sa zdalo, že ľahko vylepšíme predchádzajúce riešenie. V okamihu, keď ideme spracovať i-tu skupinu, jej aktuálny úsek kandidátov môže byť zmenšený o to, čo sme sa už dozvedeli pri spracúvaní predchádzajúcich skupín. Môže sa teda stať, že budeme potrebovať menej otázok pri binárnom vyhľadávaní správneho princa v i-tej skupine. (Uvedomte si, že za princa p má vždy zmysel voliť len jedného z aktuálnych kandidátov, inak sa nedozvieme nič nové.)

V mnohých situáciách by toto zlepšenie skutočne pomohlo. Problémom je však najhorší možný prípad. Ten aj po našom vylepšení zostáva stále rovnaký. Predstavte si napríklad vstup, kde pre každé i platí, že všetci pytači v i-tej skupine sú chudobnejší od pytačov v nasledujúcich skupinách. Stále teda máme riešenie s časovou zložitosťou v najhoršom prípade až $\Theta(n^2 \log^2 k)$.

Využitie náhody

Ocitli sme sa práve vo veľmi podobnej situácii ako napr. pri triedení QuickSort: hoci skoro každá voľba pivota (v našom prípade princa p) je dobrá a teda výrazne nám zmenší množinu kandidátov, existujú akési špecifické zlé vstupy, kedy ako na potvoru robíme samé zlé voľby a teda výpočet beží dlho.

No a podobne ako u QuickSortu, aj tu nám pomôže náhoda. Dostaneme tak riešenie, ktoré *nebude mať žiadne zlé vstupy*. Pre *úplne hocijaký* vstup bude platiť, že čas behu programu závisí len od toho, ako dobre sa nám bude dariť pri náhodnom výbere pivotov. No a bude platiť, že *v priemernom (očakávanom) prípade* sa nám bude dariť dostatočne dobre. Pozrime sa teda na to poriadnejšie.

Nový, vylepšený algoritmus: Na začiatku inicializujeme množinu kandidátov na prvých k princov z každej skupiny. No a kým máme viac ako jedného kandidáta, tak dokola opakujeme: za princa p zvolíme *náhodného*

spomedzi všetkých kandidátov, v čase $O(n \log k)$ zistíme poradie princa p, a podľa neho upravíme množinu kandidátov.

Odhad časovej zložitosti len načrtneme. Uvažujme ľubovoľnú situáciu počas behu algoritmu. Nech má aktuálna množina kandidátov veľkosť c. Potom ľahko spočítame, že v nasledujúcom kole zahodíme v priemere aspoň $c/4$ kandidátov. No a keďže $(3/4)^3 < 1/2$, môžeme očakávať, že za každé tri kolá sa nám počet kandidátov v priemere zmenší aspoň na polovicu. V priemernom prípade bude teda kôl nanajvýš $3 \log_2(nk)$, čo vďaka nerovnosti $n < k$ môžeme odhadnúť ako $O(\log k)$ kôl.

V rámci každého kola potrebujeme najskôr vybrať náhodného kandidáta (rozmyslite si, že to vieme ľahko spraviť napr. v čase $\Theta(n)$) a potom ho použiť ako pivota (na čo treba čas $O(n \log k)$). Dokopy teda dostávame riešenie, ktorého očakávaná časová zložitosť (na ľubovoľnom vstupe) je $O(n(\log k)^2)$.

(Poznamenáme ešte, že sa dá dokázať aj silnejšie tvrdenie, ktoré hovorí, že pre dostatočne veľké n a k je pravdepodobnosť toho, že by náš algoritmus bežal rádovo dlhšie, zanedbateľne malá.)

Vzorové riešenie

Vráťme sa teraz späť k prvému riešeniu, kde sme používali haldu. Problém bol, že sme zo skupín vždy vyberali len po jednom prvku, takže to trvalo príliš dlho. Skúsme teda odhadzovať viac prvkov naraz. Rozdeľme si každú skupinu princov na bloky veľkosti s. (To, aké veľké s bude, sa rozhodneme neskôr.) Bloky budeme porovnávať tak, že porovnáme princov, ktorými začínajú. Zoberieme maximovú haldu do ktorej vložíme n záznamov: prvý blok z každej skupiny princov. Následne z haldy postupne vyberieme a zahodíme $\lfloor k/s \rfloor$ blokov (pričom zakaždým, keď nejaký blok vyhodíme, vložíme namiesto neho nasledujúci blok z tej istej skupiny).

Nech p je prvý (teda najväčší) prvok v poslednom vybranom bloku. Všetky prvky, ktoré sme nevybrali, sú nutne menšie ako p – lebo všetky ostatné bloky v halde začínajú prvkami menšími ako p a všetky ostatné prvky sú od týchto začiatočných prvkov menšie. No a vybratých prvkov bolo nanajvýš k, preto platí, že princ, ktorého hľadáme, je buď p, alebo je od neho chudobnejší.

Čo ešte vieme o hľadanom princovi povedať? Predstavme si, že sme pri práve popísanom procese z jednej skupiny postupne vyhodili bloky b_1 až b_t. Posledný blok b_t začína princom aspoň tak bohatým ako p. (Je to presne p, ak je b_t úplne posledný vyhodený blok, inak je to iný, bohatší princ.) To ale znamená, že všetky bloky, ktoré sme z danej skupiny vyhodili skôr, obsahujú samých princov bohatších ako p – a teda bohatších ako ten, ktorého hľadáme. Všetkých týchto princov môžeme teda poslať domov a príslušne zmenšiť poradové číslo k princa, ktorého hľadáme.

Aby to bolo rozumne rýchle, s si na začiatku zvolíme ako najbližšiu mocninu dvoch menšiu ako k. Následne budeme celý vyššie popísaný proces viackrát opakovať, pričom zakaždým zmenšíme s na polovicu. V poslednej iterácii bude $s = 1$, čím dostaneme algoritmus popísaný v časti o malej hodnote k.

Odhadnime teraz časovú zložitosť tohto algoritmu. Zjavne prebehne $\log k$ iterácií vyššie popísaného procesu. V prvej iterácii (vďaka začiatočnej voľbe s) budú výbery z haldy nanajvýš dva.

Všimnime si teraz ľubovoľnú iteráciu i. Máme nejakú aktuálnu hodnotu k (poradové číslo princa, ktorého práve hľadáme) a s. Postupne vyberieme $\lfloor k/s \rfloor$ blokov z haldy a následne z každej skupiny všetky vybraté bloky okrem posledného pošleme domov. Domov sme teda poslali aspoň $\lfloor k/s \rfloor - n$ blokov princov. Inými slovami, spomedzi k najbohatších princov nám ich ostalo určite menej ako $s(n+1)$, a teda pre nové poradové číslo k' princa, ktorého hľadáme v nasledujúcej iterácii, bude platiť $k' < s(n+1)$.

To ale znamená, že v iterácii $i+1$, keď budeme mať bloky veľkosti $s/2$, vyberieme z haldy dokopy $\lfloor k'/(s/2) \rfloor \leq 2(n+1)$ blokov.

No a vyššie popísaná úvaha platí pre každé i. V každej iterácii bude teda najviac $2(n+1)$ výberov z haldy. No a každý z nich trvá $O(\log n)$, čím dostávame celkovú časovú zložitosť $O(n \log n \log k)$.

Poznámka na záver: Existuje aj komplikovanejšie riešenie s časovou zložitosťou $O(n \log k)$.

Listing programu (C++)

```
#include <vector>
#include <queue>
#include <cstdio>
```

```
using namespace std;

struct princ {
    int s; //číslo skupiny
    int i; //pozícia v skupine
};

bool operator<(const princ &p, const princ &d) {
    return bohatsi(d.s+1,d.i+1,p.s+1,p.i+1)<0;
}

int main() {
    int n,k;
    scanf("%d %d",&n,&k);
    vector<int> pocty(n);
    for(int i=0; i<n; i++) {
        scanf("%d ",&pocty[i]);
    }
    //najbližšia mocnina 2
    int s=1;
    while(2*s <= k) s*=2;
    //halda na skupiny princov
    priority_queue<princ> halda;
    vector<int> pozicie(n,0);
    while(s>=1) {
        halda=priority_queue<princ>();
        for(int i=0; i<pozicie.size(); i++) {
            //vráťme poslednú skupinu
            if(pozicie[i] > 0){
                pozicie[i]-=2*s;
                k+=2*s;
            }
            //naplňme haldu
            princ p={i,pozicie[i]};
            halda.push(p);
        }
        //skúšame kroky veľkosti s
        for(; k-s>0; k-=s) {
            princ p = halda.top();
            halda.pop();
            p.i=pozicie[p.s]+=s;
            if(p.i < pocty[p.s]) halda.push(p);
        }
        s/=2;
    }
    //hľadaný princ je teraz navrchu haldy
    princ hladany = halda.top();
    printf("%d %d\n",hladany.s+1,hladany.i+1);
    return 0;
}
```

A-III-3 Log-space výpočty

Podúloha A

Na bežnom počítači by sme túto úlohu vyriešili prehľadávaním. O každom prvku by sme si pamätali, či už bol navštívený. Pri počítaní cyklov postupne prechádzame prvkami permutácie a vždy, keď nájdeme nenavštívený prvok, zvýšime si počet cyklov, prejdeme celý cyklus obsahujúci daný prvok a označíme všetky jeho prvky ako navštívené.

Na takéto riešenie by sme však potrebovali pomocnú pamäť lineárnu od dĺžky permutácie, aby sme pre každý prvok vedeli, či sme ho navštívili alebo nie. Log-space program si však nemôže pamätať niečo pre každý prvok, dokonca ani pre každý cyklus nie – môže ich byť až $O(n)$. (Napr. pre $n = 8$ má permutácia 12345678 presne 8 cyklov, permutácia 21436587 ich má $n/2 = 4$.)

Ako na to teda s logaritmickou pamäťou?

Jedna sľubne vyzerajúca cesta je priradiť prvkom rôzne váhy: keď spracúvam prvok, zistím si dĺžku d cyklu, na ktorom leží, a k výsledku pripočítam hodnotu $1/d$. Keď teda mám napríklad konkrétny cyklus dĺžky 3, pre každý jeho prvok pripočítam k výsledku hodnotu 1/3, čím dostanem v súčte 1. Toto riešenie síce funguje, ale samo o sebe nie je log-space, keďže naše programy nevedia pracovať s reálnymi číslami.

Mohli by sme sa pokúsiť reprezentovať priebežný súčet ako zlomok: v jednej premennej si budeme pamätať jeho čitateľ, v druhej menovateľ. Takéto riešenie však opäť nie je log-space. Prečo? Pretože čitateľ aj menovateľ

môžu byť veľmi veľké v porovnaní s n. Uvedomte si totiž, že vo všeobecnosti môže byť niekedy uprostred výpočtu menovateľ pamätaného zlomku najmenším spoločným násobkom dĺžok všetkých cyklov danej permutácie.

(Pomocou vysokoškolskej matematiky sa dá dokázať, že na uloženie menovateľa treba $\Theta(\sqrt{n \log n})$ bitov, čo je priveľa. Jednoduchší príklad toho, že $O(\log n)$ bitov nestačí: predstavte si permutáciu, ktorá má zhruba $\sqrt{n}$ cyklov, každý inej dĺžky, pričom tie dĺžky sú všetky približne rovné $\sqrt{n}$. Aký veľký môže byť približne menovateľ zlomku počas spracúvania takejto permutácie? Koľko bitov treba na jeho uloženie?)

Je možné, že sa toto riešenie dá upraviť do funkčnej podoby – a to využitím pozorovania, že súčet je vždy celé číslo, a teda nám neprekážajú rozumne malé zaokrúhľovacie chyby. Takáto úprava by zrejme bola veľmi komplikovaná, nehovoriac o dôkaze jej správnosti. Existuje však aj omnoho jednoduchšie riešenie.

Namiesto toho, aby sme pre každý z d prvkov na cykle zarátali k výsledku $1/d$, jednoducho pri jednom z nich zarátame 1 a pri ostatných 0. Ako toto ale zabezpečiť? Potrebujeme si pre každý cyklus určiť jeho jedného *reprezentanta*. Dobrou voľbou je napríklad najmenší prvok ležiaci na danom cykle.

Celé riešenie si teda ľahko zhrnieme jednou vetou: Pre každý prvok permutácie prejdeme celý jeho cyklus, a ak na ňom nestretneme žiaden menší prvok, tak zväčšíme o 1 počet nájdených cyklov.

Na toto nám s prehľadom stačí konštantne veľa premenných, ako demonštruje aj náš program.

Listing programu (Pascal)

```
var n: integer;                      { vstup: dĺžka permutácie }
    P : array [1..n] of integer;     { vstup: permutácia }
    c : integer;                { výstup: počet cyklov}
    poc, min, v, i : integer; { poc. cyklov, min. videny prvok, v ktorom vrchole som, z ktoreho prave prehladavam}

begin
    poc := 0
    for i := 1 to n do begin
      min := i
      v := i
      repeat
        v := P[v];
        if min > v then min := v;
      until v = i;
      if min = i then inc(poc);
    end;
    c := poc;
end.
```

Iné riešenie. Nápad „pre každý prvok ležiaci na cykle dĺžky d k výsledku pripočítam hodnotu $1/d$“ sa dá prerobiť na funkčné riešenie nasledovne: Postupne skúšame všetky d od 1 po n. Pre konkrétne d postupne prejdeme všetky prvky a spočítame, koľko z nich leží na cykle dĺžky presne d. Takto získaný počet vydelíme d (čo vieme spraviť v celých číslach) a pripočítame k výsledku.

Ešte iné riešenie. Postupne pre každé x si položíme otázku „leží x na niektorom z cyklov obsahujúcich prvky 1 až $x-1$?“ Otázku zodpovieme tak, že pre každé y od 1 po $x-1$ prejdeme cyklus obsahujúci y. Ak nikdy x nestretneme, zväčšíme si počet nájdených cyklov.

Podúloha B

S luxusnou lineárnou pamäťou by nebol problém naprogramovať prehľadávanie stromu napríklad do hĺbky. Pamätáte si krajské kolo? Tam sme spolu prišli na to, ako aj v log-space vieme spraviť prehľadávanie stromu. Okrem prvku, na ktorom práve sme, si pamätáme aj prvok, z ktorého sme sem prišli. Keď chceme ísť ďalej, ideme do suseda s najbližším väčším číslom. Dokázali sme si, že takéto prehľadávanie naozaj dokola chodí po celom strome.

Do tohoto prehľadávania však nestačí len priamočiaro pridať počítanie spravených krokov. Totiž kým sa zo začiatočného vrcholu u dostaneme do v, môže sa stať, že nepôjdeme priamo, ale niektoré vrcholy navštívime viackrát.

Predstavme si, že začneme náš strom prehľadávať z vrcholu u a časom prídeme do v. Čo vieme teraz povedať o hľadanej ceste? Nevieme síce, ako vyzerá celá, ale vieme, že jej posledná hrana je tá, ktorou sme práve prišli do v.

Prečo? Zakoreňme si strom vo vrchole v. Vrchol u leží v niektorom z podstromov. Nech e je hrana, ktorá vedie z v do daného podstromu. Do iných podstromov sa nemáme ako dostať bez toho, aby sme šli cez hranu e. Ale akonáhle ňou prejdeme, sme vo v a prehľadávanie končí.

Zistili sme teda takto poslednú hranu želanej cesty. Nech je to (x, v) pre nejaký vrchol x. Zvýšime si počet hrán na ceste, posunieme si cieľ v do vrcholu x a ak ešte neplatí $u = v$, začneme celý postup odznova.

Ku prehľadávaniu z krajského kola sme pridali len konštantný počet premenných, stále teda máme log-space program. Za povšimnutie ešte stojí, že najkratšia cesta má dĺžku menej ako n, celý postup teda budeme opakovať menej ako n-krát.

Listing programu (Pascal)

```
var n, u, v : integer;              { vstup: počet vrcholov, štart a cieľ }
    A, B : array [1..m] of integer; { vstup: hrany grafu }
    d : integer;                    { výstup: dĺžka najkratšej cesty }
    start, ciel, v, z, tmp, hran : integer;

function dalsia_hrana (w, z : integer) : integer;
var i, minn, nextn : integer;
begin
    minn := z; { sused s najmenším číslom }
    nextn := m+1; { sused s najmenším číslom väčším ako z }
    for i := 1 to m do begin
        if A[i] = w then begin
            if B[i] < minn then minn := B[i];
            if (B[i] > z) and (B[i] < nextn) then nextn := B[i];
        end;
        if B[i] = w then begin
            if A[i] < minn then minn := A[i];
            if (A[i] > z) and (A[i] < nextn) then nextn := A[i];
        end;
    end;
    if nextn <= m then dalsia_hrana := nextn
    else dalsia_hrana := minn; { neexistuje sused s väčším číslom, vrátime najmenšieho suseda }
end;

begin
    start := u;
    ciel := v;
    hran := 0;
    repeat
      v := start
      z := -1;
      repeat
          tmp := dalsia_hrana (v, z);
          z := v;
          v := tmp;
      until v = ciel;
      inc(hran);
      ciel = z;
    until start = ciel;
    d := hran;
end.
```

Iné funkčné riešenie zistí dĺžku cestu tak že určí počet vrcholov na nej – pre každý vrchol x overíme, či sa prehľadávanie z u dostane do vrcholu v, ak ho nepustíme cez x. Podobnou možnosťou je overiť to isté postupne pre každú hranu.

DVADSIATY ÔSMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Vladimír Boža, Michal Forišek, Ján Hozza, Marek Špano
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2013