Riešenia kategórie B

B-I-1 Horolezci

Na vstupe máme zadaný popis štítu, kde každá z jeho u častí má priradenú náročnosť a_i. Taktiež máme h horolezcov, pričom každý horolezec má úroveň schopností b_i a platí, že horolezec vie prejsť iba úseky s ostro menšou náročnosťou ako sú jeho schopnosti. A našou úlohou je zistiť, ako vysoko vylezie ktorý z horolezcov.

To najjednoduchšie riešenie, ktoré môžeme vyskúšať, je priamo simulovať lezenie každého z horolezcov. Zoberieme si nejakého horolezca, ktorý má úroveň schopností b_i a budeme postupne predchádzať pole, v ktorom máme poznačené náročnosti štítu až kým nenarazíme buď na koniec alebo na úsek, ktorý horolezec nevie preliezť (to znamená hodnotu väčšiu alebo rovnú hodnote b_i). Je samozrejme jasné, že toto lezenie skončí po najviac u pokusoch, keď horolezec dosiahne vrchol a ak to zopakujeme pre každého horolezca dostaneme časovú zložitosť $O(uh)$.

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>
using namespace std;

int u,h;
vector<int> U,H;

int main() {
    scanf("%d␣%d",&u,&h);
    U.resize(u); H.resize(h);
    for(int i=0; i<u; i++) scanf("␣%d",&U[i]);
    U.push_back(2000000047);
    for(int i=0; i<h; i++) scanf("␣%d",&H[i]);
    //prechadzam horolezcov
    for(int i=0; i<h; i++) {
        int zastane=0;
        while(U[zastane]<H[i]) zastane++;
        if(zastane==u) printf("vrchol\n");
        else printf("%d\n",zastane);
    }
}
```

Predchádzajúce riešenie nám získalo zopár bodov, nestačí však na vyriešenie väčších testovacích vstupov. Potrebujeme teda niečo vylepšiť. Skúsme sa pozrieť na to ako vyzerá kopec. Zo začiatku je strašne kostrbatý, raz stúpa, potom zase klesá. Nám by však vyhovovalo, ak by mal nejaký ustálený tvar, napríklad aby celý čas stúpal.

Pozrime sa teraz na nejakú časť kopca, ktorá má výšku x. Uvedomme si, že ak sa za týmto políčkom nachádza menšia hodnota y, môžeme túto hodnotu nahradiť číslom x. Ak sa totiž horolezec dostane za úsek s náročnosťou x, určite zvládne prekonať aj úsek s náročnosťou y. Toto nám zaručí, že vieme náš štít prerobiť na nový štít, ktorý na žiadnom úseku neklesá.

To spravíme tak, že budeme postupne prechádzať pole kde si pamätáme náročnosti úsekov štítu a budeme si pamätať najväčšiu náročnosť, ktorú sme zatiaľ videli v premennej m. Vždy keď nájdeme väčšie číslo, prepíšeme si hodnotu v premennej m, v opačnom prípade nahradíme hodnotu, na ktorej sme (teda menšiu ako m), hodnotou m. Výsledok bude vyzerať nejak takto:

```
pôvodná náročnosť:  2 1 4 5 1
upravená náročnosť: 2 2 4 5 5
```

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>
using namespace std;

int u,h;
vector<int> U,H;
```

```
void uprav() {
    for(int i=1; i<u; i++)
        if(U[i-1]>U[i]) U[i]=U[i-1];
}

int main() {
    scanf("%d␣%d",&u,&h);
    U.resize(u); H.resize(h);
    for(int i=0; i<u; i++) scanf("␣%d",&U[i]);
    U.push_back(2000000047);
    for(int i=0; i<h; i++) scanf("␣%d",&H[i]);
}
```

Nová hodnota b_i nám vlastne hovorí, akú úroveň schopností musí mať horolezec, aby prešiel všetko od začiatku až po tento úsek.

K tejto časti by som ešte rád dodal, že takisto je možné si zapamätať len tie miesta, kde štít stúpa, to znamená miesta, kde sme zmenili hodnotu m. V tom prípade si však musíme pamätať aj pozície, na ktorých sa to stalo. A v prípade, že štít už na začiatku stúpa si ani nezapamätáme menej hodnôt. Preto je lepšie použiť vyššie spomenuté riešenie, pretože v najhoršom prípade je rovnako rýchle a zároveň je jednoduchšie na implementáciu.

Binárne vyhľadávanie

V okamihu ako sme si našu postupnosť upravili tak aby neklesala, mali by nám napadnúť tieto dve slová. V prípade, že binárne vyhľadávanie nepoznáte, jeho myšlienka je veľmi jednoduchá. Predstavte si, že máte telefónny zoznam priezvisiek a snažíte sa nájsť meno „Mrkvička“. Čo spravíte? No predsa otvoríte zoznam v strede a pozriete sa, aké mená sa tam nachádzajú. Povedzme, že uvidíte meno „Novák“. Keďže písmeno N je až za písmenom M, znamená to, že priezvisko Mrkvička bude v prvej polovici zoznamu. Na druhú polovicu teda môžete zabudnúť a s prvou polovicou opakujete to isté. Otvoríte ju v strede a podľa mien určíte, v ktorej časti máte hľadať ďalej. V každom kroku sa tak zbavíte polovice možných strán, takže po logaritmickom počte pokusov by ste mali nájsť hľadané meno.

Toto isté teraz vieme spraviť aj s našimi horolezcami. Zoberieme si horolezca so schopnosťou b_i. Skúsme zistiť, či sa mu podarí vyštverať do stredu našeho štítu. Nech je na strednom úseku hodnota x. Ak je x väčšia alebo rovná ako b_i, horolezcovi sa sem nepodarí vyštverať a teda úsek kde skončí musíme hľadať v prvej polovici. V opačnom prípade horolezec určite vylezie na stredný úsek a teda musíme hľadať v druhej časti postupnosti. Na časti, ktorá nám zostala opakujeme ten istý postup, až kým nám nezostane jediný úsek – ten, na ktorom horolezec skončí svoj výstup.

Táto myšlienka je síce pomerne jednoduchá, napriek tomu jej správne implementovanie môže byť náročné. Skúsim ho tu čo najlepšie popísať a následne odporúčam si naštudovať program pre lepšie pochopenie.

Celý čas si budeme pamätať dve hodnoty z a k, ktoré budú označovať interval úsekov, na ktorých horolezec môže skončiť svoju túru. Na začiatku tieto hodnoty zahŕňajú celý interval, postupne sa však posúvajú. Naviac, aby v tom bol prehľad, chceme aby platilo, že z ukazuje na úsek, na ktorý horolezec **vie** vyliezť a k ukazuje na úsek, kam horolezec **nevie** vyliezť. Na začiatku sú tieto hodnoty dosť neintuitívne nastavené na $z = -1$ a $k = u$, čo sú oba prvky mimo poľa. Vieme to však vyriešiť, že na koniec pridáme úsek s náročnosťou $2 \cdot 10^9 + 47$, teda úsek kam nevie vyliezť nikto a k teda ukazuje do poľa. To, že z ukazuje mimo poľa nám potom nevadí.

Následne sa pozrieme na úsek v strede tohoto intervalu. Ten má index $s = (z + k)/2$. Ak je na pozícii s menšia hodnota ako b_i, horolezec na ňu vie vyliezť. A keďže naša postupnosť neklesá, musí vedieť vyjsť aj na všetky predošlé úseky. Chceme teda hľadať v druhej časti našeho intervalu. Zmeníme preto $z = s$. Uvedomme si, že týmto sme si nepokazili vlastnosť, ktorá má platiť pre z.

Ak je na pozícii s hodnota väčšia alebo rovná, znamená to, že horolezec sa cez ňu nevie dostať a preto správny úsek musíme hľadať v prvej polovici poľa, preto nastavíme $k = s$. Opäť sme zachovali podmienku, ktorú sme si stanovili. Keď nám zostane už len jeden prípustný prvok, kde horolezec môže skončiť ($k - z = 1$), úsek s indexom k je naša odpoveď.

Keď takéto vyhľadávanie spustíme pre každého horolezca dostaneme korektné riešenie, ktoré bude mať časovú zložitosť $O(u + h \log u)$. Spracovanie poľa nám totiž zaberie čas $O(u)$ a jedno binárne vyhľadávanie trvá $O(\log u)$.

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>
using namespace std;

int u,h;
vector<int> U,H;

void uprav() {
    for(int i=1; i<u; i++)
        if(U[i-1]>U[i]) U[i]=U[i-1];
}

int main() {
    scanf("%d␣%d",&u,&h);
    U.resize(u); H.resize(h);
    for(int i=0; i<u; i++) scanf("␣%d",&U[i]);
    U.push_back(2000000047);
    for(int i=0; i<h; i++) scanf("␣%d",&H[i]);
    uprav();
    //binarne vyhladavam pre kazdeho horolezca
    for(int i=0; i<h; i++) {
        int z=-1,k=u;
        while(k-z>1) {
            int s=(k+z)/2;
            if(U[s]<H[i]) z=s;
            else k=s;
        }
        if(k==u) printf("vrchol\n");
        else printf("%d\n",k);
    }
}
```

Alternatívne riešenie

Binárne vyhľadávanie je užitočná metóda, niekedy však môže byť ťažká na implementáciu, alebo človeku nemusí napadnúť. Pozrime sa, čo sa dalo s touto úlohou spraviť bez neho. Dokonca nebudeme potrebovať ani upravovať pole náročností štítu. Skôr sa pozrieme na horolezcov.

Zoberme si dvoch horolezcov i a j, pričom j-ty horolezec je zdatnejší, teda $b_i \leq b_j$. Uvedomme si, že všade, kam sa vie vyštverať i-ty horolezec sa vyštverá aj j-ty, lebo je aspoň tak schopný. Ak by sme teda vedeli, kde skončí svoj výstup horolezec i, keď budeme spracovávať horolezca j, nemusíme opäť prechádzať celé pole, ale môžeme začať rovno od tejto pozície, lebo tam určite zvládne vyliezť.

Zoberieme si teda všetkých horolezcov a usporiadame ich podľa ich schopností od najslabšieho po najlepšieho. Potom presne tak ako v jednoduchom riešení budeme postupne simulovať výstup každého z nich. Iba s tým rozdielom, že nebudeme vždy začínať na začiatku, ale tam, kde skončil posledný horolezec.

Všimnime si, že s takýmto zlepšením sa nikdy nepozrieme na úsek kopca dvakrát, lebo vždy len posúvame najvzdialenejší dosiahnutý úsek. Po jednom prejdení teda budeme mať odpoveď pre všetkých horolezcov. Celkovo sme teda potrebovali utriediť horolezcov v čase $O(h \log h)$ a potom prejsť poľom štítu v čase $O(u)$, takže máme algoritmus s časovou zložitosťou $O(u + h \log h)$. Dostávame tak rovnako dobrý algoritmus, ktorý sa navyšše aj jednoduchšie implementuje.

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>
using namespace std;

int u,h;
vector<int> U,Vys;
vector<pair<int,int> > H;

int main() {
    scanf("%d␣%d",&u,&h);
    U.resize(u); H.resize(h); Vys.resize(h);
    for(int i=0; i<u; i++) scanf("␣%d",&U[i]);
    U.push_back(2000000047);
    for(int i=0; i<h; i++) scanf("␣%d",&H[i].first);
    for(int i=0; i<h; i++) H[i].second=i;
    sort(H.begin(),H.end());
    int zacni=0;
```

```
    for(int i=0; i<h; i++) {
        while(U[zacni]<H[i].first) zacni++;
        Vys[H[i].second]=zacni;
    }
    for(int i=0; i<h; i++) {
        if(Vys[i]==u) printf("vrchol\n");
        else printf("%d\n",Vys[i]);
    }
}
```

B-I-2 Pexeso

Na vyriešenie tejto úlohy nebola potrebná zložitá myšlienka, zadanie pomerne priamočiaro ukazovalo, čo treba naprogramovať. Problém však mohol nastať pri samotnom programovaní. Ak ste sa totiž do toho vrhli bezhlavo a bez prípravy, mohli ste postupne zisťovať rôzne nepríjemnosti, ktoré táto úloha obsahuje. A to mohlo viesť k dlhému a komplikovanému programu, ktorý sa ťažko dokončuje a v ktorom sa ťažko hľadajú chyby. Z tejto úlohy si preto treba odniesť poučenie, že vždy je lepšie porozmýšľať nad programom pred tým, ako ho začneme písať. Skôr ako sa teda pustíme do vysvetľovania toho, ako to naprogramovať, ukážeme si pár pozorovaní, ktoré nám pomôžu navrhnúť jednoduchý a prehľadný program.

V každom okamihu existuje také číslo k, že sme videli všetky kartičky naľavo od k-tej kartičky a nevideli sme žiadnu z kartičiek napravo od k-tej kartičky a ani k-tu kartičku. V zadaní je totiž jasne napísané, že vždy, keď sa nevieme rozhodnúť čo máme odkryť, máme odkryť najľavejšiu ešte neodkrytú kartičku. To znamená, že sa nám nemôže stať, že odkryjeme k-te pexeso a ešte sme neodkryli l-té pexeso, ktoré je v rade skôr. Podľa zadania by sme totiž boli nútení otočiť práve toto l-té pexeso.

Keď sme si teda ukázali pravdivosť tohoto tvrdenia, môžeme si v našom programe vytvoriť premennú kde, ktorá bude ukazovať na najľavejšie neodkryté pexeso. Budeme teda vedieť rýchlo nájsť pozíciu pexesa, ktoré máme odkryť. Naviac vieme, že všetky pexesá napravo od pozície kde sme nevideli a preto nová hodnota kde, teda najľavejšie neodkryté pexeso po odkrytí pexesa kde, bude $kde + 1$.

Poďme si teraz rozobrať rôzne prípady toho, čo hráč, ktorý je na ťahu, vie o pexese. Sú len tri možnosti:

1. Vie, kde sa nachádza kompletný pár.
2. Nevie, kde sa nachádza celý pár. Otočí preto ešte neotočenú kartičku a zistí, že jej pár už videl.
3. Nevie, kde sa nachádza celý pár. Otočí preto ešte neotočenú kartičku, ale zistí, že jej pár ešte nevidel, takže musí otočiť aj druhú novú kartičku.

Pozrime sa na prvý prípad. Ak hráč na ťahu vie kde sa nachádza kompletný pár, hráč, ktorý išiel pred ním to vedieť nemohol, lebo by už on ten pár zobral. To ale mohlo nastať len jedným spôsobom – a to tak, že ako druhú v poradí otočil predchádzajúci hráč novú kartičku a ukázalo sa, že tá pasuje ku kartičke otočenej v nejakom skoršom ťahu. To nás vedie k ďalšiemu dôležitému pozorovaniu: **Ak existuje pár, z ktorého sme videli obe kartičky, jedna z nich sa musí nachádzať na pozícii $kde - 1$.**

Ďalšia vec, ktorú musíme vyriešiť, je spôsob, ako zistiť, či sme k danému pexesu videli už niekedy predtým pár. Jedným spôsobom je prejsť naše pole a skontrolovať to, keďže vieme, že pár sa musí nachádzať naľavo, keďže napravo sú ešte neotočené karty. To je však trochu pomalé. Oveľa lepšie riešenie je spraviť si pole $videl[]$, kde si pre každú hodnotu karty budem pamätať, či som ju už videl.

Na začiatku bude pole $videl[]$ naplnené hodnotami -1, čo znamená, že sme nič nevideli. Vždy keď otočíme nejakú kartu s číslom x, preznačíme hodnotu $videl[x]$ číslom 1. Jediný možný problém je, ak nastane predchádzajúci prípad, keď vidíme obe karty, to musíme rozlíšiť nejak zvlášť, vo vzorovom programe, ak idem preznačiť hodnotu 1, nastavím tam číslo 2, ktoré bude znamenať, že toto číslo som videl už dvakrát a v ďalšom ťahu chcem tento pár zobrať.

(Ak by sme chceli aj vypisovať pozície kariet, ktoré hráči otáčajú, stačilo by si v poli $videl[]$ namiesto čísla 1 zapísať ku každej hodnote, ktorú sme už videli, index do poľa, na ktorom sa nachádzala.)

Čo sa týka posledného prípadu, jediné čo si musíme uvedomiť je, že občas môžeme mať šťastie a obe karty, ktoré otočíme sú rovnaké. V tom prípade sa musíme samozrejme zachovať požadovaným spôsobom.

Implementácia

Zostáva nám už len zhrnúť naše pozorovania a povedať si ako to celé naimplementovať. Na označovanie toho, kto je na ťahu, budeme používať premennú *tah*, ktorá má buď hodnotu 0 (ťahá prvý hráč), alebo hodnotu 1 (ťahá druhý hráč). Ďalej budem mať už spomínané pole *videl*[], premennú *kde* a pole *karty*[], kde si uložíme karty na vstupe. Celý algoritmus bude bežať, kým premenná *kde* bude menšia ako $2n$.

Na spracovanie ťahu si vytvoríme funkciu *spravtah*(). Táto funkcia bude postupne rozoberať všetky vyššie spomenuté možnosti. Na konci vráti *true* alebo *false* podľa toho, či sa nám podarilo zobrať nejaký pár. Ak sa to podarilo, pokračuje na ťahu ten istý hráč a dostane bod. Ak nie, ide na ťah jeho protihráč.

Funkcia *spravtah*() najskôr zistí, či predchádzajúca karta ($kde - 1$) netvorí už známy pár. Ak áno, odoberie ho a vráti *true*. Ak nie, pozrie sa, či po otočení prvej novej karty nedostaneme pár. Ak áno, odoberie ho a vráti *true*. Ak nie, otočí aj druhú novú kartu a pozrie sa, či tá náhodou netvorí pár s prvou. Ak máme šťastie a je to tak, aj tento pár odoberieme a vrátime *true*. A ak ani posledný vyššie uvedený prípad nenastal, vrátime *false*. Zakaždým nezabudneme príslušne zmeniť hodnoty našich premenných a polí.

Celé toto riešenie bude mať zložitosť $O(n)$, keďže sa na každú kartu pozrieme len konštantne veľa ráz. Pamäť bude tiež $O(n)$.

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>
using namespace std;

int n,tah,kde;
vector<int> karty,videl;

bool spravtah() {
    //zarucene poznam dvojicu
    if(kde!=0 && videl[karty[kde-1]]==-2) {
        videl[karty[kde-1]]=-1;
        return true;
    }
    //otocim prvu kartu a uz som ju videl
    if(videl[karty[kde]]!=-1) {
        kde++;
        return true;
    }
    //obe otocene karty su rovnake
    if(kde!=2*n-1 && karty[kde] == karty[kde+1]) {
        kde+=2;
        return true;
    }
    //nic som neziskal
    videl[karty[kde]]=1;
    if(videl[karty[kde+1]]!=-1) videl[karty[kde+1]]=-2;
    else videl[karty[kde+1]]=1;
    kde+=2;
    return false;
}

void solve() {
    scanf("%d",&n);
    karty.clear(); karty.resize(2*n);
    videl.clear(); videl.resize(n,-1);
    tah = kde = 0;
    for(int i=0; i<2*n; i++) scanf(" %d",&karty[i]);
    int body[] = {0, 0};
    while(kde != 2*n) {
        if(spravtah()) body[tah]++;
        else tah=(tah+1)%2;
    }
    printf("%d %d\n",body[0],body[1]);
}

int main() {
    for(int i=0; i<5; i++) solve();
}
```

B-I-3 Výťah

Na prízemí veľapodlažnej budovy, v ktorej chodí jeden výťah, máme n ľudí. Výťah unesie naraz najviac k ľudí. Nastúpenie a vystúpenie jedného človeka trvá 1 sekundu a takisto posunutie výťahu o jedno poschodie trvá 1 sekundu. Každý človek má určené poschodie kam sa chce dostať a našou úlohou je zistiť najkratší čas, za ktorý sa všetci ľudia dostanú na chcené poschodia.

Keďže na začiatku nikto nie je tam, kde chce byť, každý z nich teda musí raz nastúpiť do výťahu. Ak sa však niekto ocitne vo výťahu, nemá už dôvod vystupovať inde ako na vlastnom poschodí. Ak by totiž vystupoval niekde inde, iba by strácal čas zbytočným nastupovaním a vystupovaním, a takisto sa neoplatí s niekým vymieňať, lebo všetci sú buď tam kde chcú byť alebo na prízemí. To znamená, že každý človek **práve raz** nastúpi a vystúpi z výťahu. Dokopy teda trvá $2n$ sekúnd nastupovanie a vystupovanie. Zvyšný čas je už len chodenie hore dole výťahom.

Úloha sa nám teda zjednodušila. Nastupovanie a vystupovanie nič nestojí, dôležitý je len pohyb výťahu. Teraz už len chceme zistiť, ako rozdeliť ľudí na také skupiny, že každá skupina nastúpi do výťahu spoločne a celkový čas rozvozu bude čo najmenší.

V každej skupine je jeden človek, ktorý chce ísť v rámci skupiny na najvyššie poschodie x. Výťah teda musí vyjsť na toto poschodie za čas x sekúnd. Vyššie chodiť nemusí, lebo tam nemá koho vysadiť. Zastavovanie a vystupovanie nás však už nič nestojí (čas toho sme zarátali zvlášť) a keďže x je najvyššie poschodie, kam chce ísť niekto z tejto skupiny, ostatní budú chcieť ísť na nižšie poschodia. Po ceste výťahu na poschodie x môžu teda všetci ostatní postupne vystúpiť tam kde chcú a na x-tom poschodí je už výťah prázdny. Môže sa teda za ďalších x sekúnd vrátiť na prízemie.

Vidíme teda, že rozvezenie ľubovoľnej skupiny ľudí bude trvať $2x$ sekúnd, kde x je najvyššie poschodie, kam sa chce niekto z nich dostať. Keď teda spravíme rozdelenie na skupiny, stačí nám sčítať ich maximá krát dva, prirátať $2n$ za vystupovanie a nastupovanie a máme odpoveď. Otázkou však zostáva to, ako budú vyzerať jednotlivé skupiny. (Nezabudnite, že každú skupinu môže tvoriť najviac k ľudí – viac sa ich do výťahu nezmestí!)

Maximálnosť skupín

Ako prvé si ukážeme, že nič nepokazíme, ak ich spravíme maximálne, teda na najviac jednu budú všetky obsahovať k ľudí. Keď chceme dokázať, že platí takýto greedy[1] predpoklad, najlepšie je odvolávať sa na nejaké (aj keď ho nepoznáme) optimálne riešenie. Dôkaz zoberie takéto optimálne riešenie a upraví ho na nami požadovaný tvar bez toho, aby ho zhoršil.

Zoberme si teda nejaké optimálne rozdelenie do skupín, kde existujú aspoň dve skupiny, ktoré neobsahujú k ľudí. Nech maximum prvej skupiny je x_1 a maximum druhej skupiny je x_2. Nech naviac $x_1 \leq x_2$. Čo sa stane, ak presunieme človeka x_1 do druhej skupiny? Určite si naše riešenie nezhoršíme. Druhá skupina musí stále ísť až na poschodie x_2 a vysadiť niekoho na x_1 nie je problém. Prvej skupine sme však odobrali maximálny prvok, takže sa jej maximum určite nezvýšilo – buď kleslo alebo ostalo rovnaké. No a túto úvahu môžeme opakovať až pokiaľ buď nenaplníme druhú skupinu alebo sa nám prvá skupina nevyprázdni.

Týmto sme dokázali, že naozaj chceme skupiny s čo najväčším počtom ľudí.

Zloženie skupiny

Zostáva nám zistiť, ako zaraďovať ľudí do skupín. Pozrime sa na človeka, ktorý chce ísť z úplne všetkých ľudí najvyššie, na poschodie x. Ten sa musí nachádzať v nejakej skupine a táto skupina sa určite odvedzie až na poschodie x. Máme teda $k-1$ voľných miest v tejto skupine kam chceme niekoho dať. Intuitívne by nám mohlo byť zjavné, že najlepší kandidáti sú tí, čo chcú ísť tiež vysoko. Takto sa ich totiž zbavím a nebudeme ich musieť riešiť neskôr. Intuícia však nestačí, treba si to aj dokázať.

Nech máme riešenie, kde v skupine s človekom x nie je k najvyššie idúcich ľudí. V tejto skupine je naviac človek y a má tam patriť človek z. Zjavne $y \leq z$, keďže z patrí medzi k najväčších a y nie. Ak ich však vymeníme naše riešenie sa určite nezhorší. Prvá skupina musí ísť stále na poschodie x a druhá skupina dostala niekoho, kto chce ísť nižšie ako ten predtým, takže určite nebude musieť ísť vyššie. Takýmto spôsobom môžeme upraviť

[1] Po slovensky pažravý. Popisuje to algoritmy, ktoré fungujú na princípe, že sa zakaždým rozhodujú podľa podmienky lokálne, teda vedia, že keď to spravia podľa nej, určite dostanú optimálne riešenie.

celú prvú skupinu aby obsahovala k najvyššie idúcich ľudí. Potom ju odstránime z našej množiny. Dostaneme množinu, kde je menej ľudí, a na ktorej vieme opäť použiť túto úvahu.

Pomocou týchto dvoch úvah by už nemal byť problém zostrojiť algoritmus. Na začiatku si usporiadame všetkých ľudí podľa poschodia kam chcú ísť od najvyššieho po najnižšie. Následne budeme postupne brať k-tice za sebou idúcich ľudí a radiť ich do jednej skupiny, pričom do výsledku zarátame vždy dvojnásobok maxima tejto skupiny. No a keď to mám usporiadané podľa výšky, asi si nie je ťažké uvedomiť, že najvyššie idúci človek prvej skupiny je na pozícii 0, druhej skupiny na pozícii k, tretej na $2k$... Na konci ešte netreba zabudnúť pričítať $2n$ za vystupovanie a nastupovanie.

Čas na usporiadanie n prvkov je $O(n \log n)$ a na sčítanie každého k-teho čísla $O(n/k)$, čo je dokopy stále $O(n \log n)$.

Listing programu (C++)

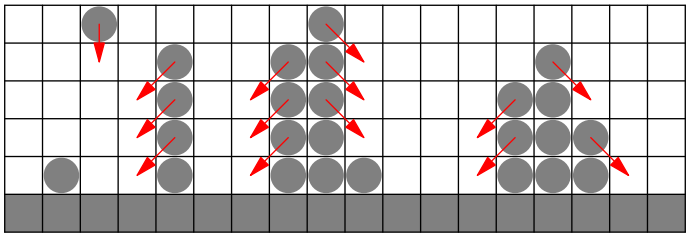
```
#include <cstdio>
#include <algorithm>
#include <vector>
using namespace std;

int main() {
    int n,k;
    scanf("%d␣%d",&n,&k);
    int vysledok = 2*n;
    vector<int> V; V.resize(n);
    for(int i=0; i<n; i++) scanf("␣%d",&V[i]);
    sort(V.begin(), V.end());
    reverse(V.begin(), V.end());
    for(int i=0; i<n; i+=k) vysledok += 2*V[i];
    printf("%d\n",vysledok);
}
```

B-I-4 Piesok

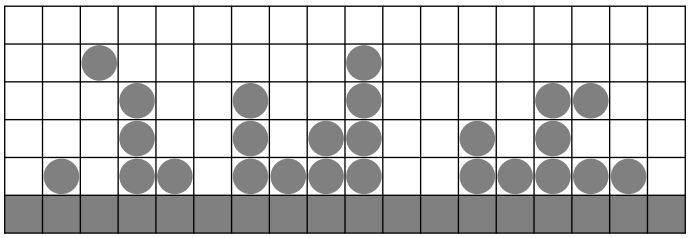
Postupne uvedieme riešenia všetkých troch podúloh v poradí, v akom boli uvedené v zadaní.

Podúloha A (3 body)

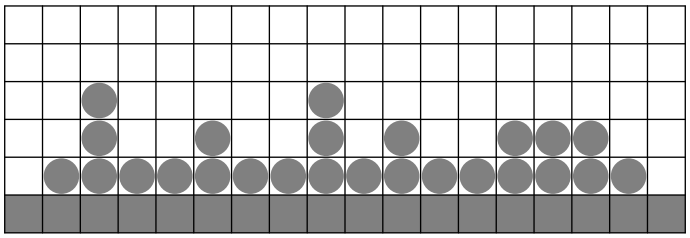
V tejto podúlohe stačilo len dôsledne aplikovať pravidlá pohybu zrniek. Pre rozloženie zo zadania sa zrnká chcú pohnúť nasledovne:



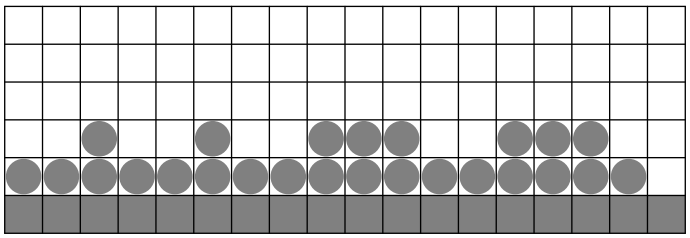
Po prvom takte teda dostaneme nasledujúce rozloženie:



Z neho po druhom takte vznikne toto (všimnite si hlavne, ako sa pohli zrnká z 10. stĺpca):

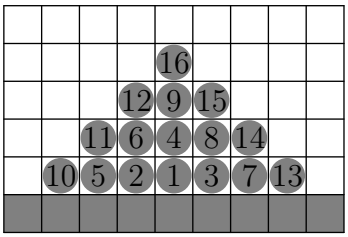


V treťom takte sa už pohnú len dve najvyššie položené zrná (na predošlom obrázku sú v treťom a v deviatom stĺpci, v treťom riadku). Obe spadnú šikmo doľava dodola. No a vo štvrtom takte už len ľavejšie z týchto dvoch zŕn padne ešte raz, opäť šikmo doľava dodola, čím dopadne na podložku v najľavejšom stĺpci. Tým všetok pohyb piesku ustane. Ustálenie piesku teda trvalo 4 takty. Tu je výsledné rozloženie zŕn:



Podúloha B (4 body)

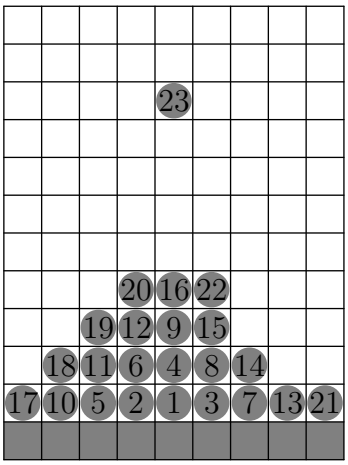
Zrnká piesku, ktoré padajú zhora na rovnú dosku, na nej začnú postupne vyrábať „pyramídu“ trojuholníkového tvaru. Na poslednom príklade v zadaní môžete vidieť, že po 31 taktoch je už prvých 5 zrniek tejto pyramídy na svojich záverečných miestach. Na nasledujúcom obrázku sú záverečné polohy prvých 16 zrniek. Všimnite si, že každá nová „vrstva“ pyramídy vzniká tak, že najskôr niekoľko zrniek odíde doľava, potom toľko isto odíde doprava, a na záver nasledujúce zrnko ostane sedieť v strede a vytvorí tak nový hrot pyramídy.



Pozorovanie, ktoré sme práve spravili, môžeme využiť na to, aby sme bez simulácie priamo zostrojili stav po 112 taktoch. Prvá vec, ktorú ľahko nahliadneme, je, že máme presne 23 zrniek: zrnko číslo 23 sa zjavilo po 110 taktoch. Toto posledné zrnko teda stihlo spadnúť len o dve políčka dodola, a teda je ešte vo vzduchu.

Zrnko číslo 21 malo na padanie 12 taktov, a to je dosť na to, aby dosiahlo podložku. V situácii po 112 taktoch sa teda prvých 21 zrniek už nehýbe a tvoria vyššie popísanú pyramídu.

Jediné, čo ešte potrebujeme zistiť, je aktuálna poloha zrnka číslo 22. Keď si nakreslíme pyramídu tvorenú prvými 21 zrnkami, jeho pohyb ľahko odsimulujeme. Výsledné rozmiestnenie zrniek teda vyzerá nasledovne:

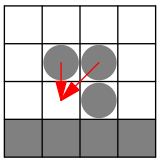


Podúloha C (3 body)

Ak máme len jedno zrnko, to bude padať rovno dodola, až kým nedopadne na podložku. Nikde žiaden problém. To isté platí pre dve zrnká v rôznych stĺpcoch.

Dve zrnká v tom istom stĺpci budú padať dodola, až kým sa jedno z nich neocitne tesne nad druhým. Potom to horné padne naľavo od toho dolného. Opäť, žiaden problém nenastáva.

Najmenší počet zrniek, pri ktorých nastane problém, sú teda tri. A skutočne, existuje rozmiestnenie troch zrniek, pre ktoré v zadaní popísaný problém nastane:



Ako sa takémuto problému vyhnúť? Existuje veľa rôznych spôsobov, vedú však k vizuálne odlišným správaniam piesku. Jednou možnosťou je očíslovať si zrnká a v každom takte nimi hýbať postupne po jednom, v poradí podľa čísel. Druhou možnosťou je robiť to isté, ale v každom takte napríklad po riadkoch zdola hore a v rámci riadku zľava doprava. Treťou možnosťou je si najskôr našim pôvodným spôsobom pre všetky zrnká spočítať, kam ktoré zrnko chce ísť, a potom ich v niektorom systematickom poradí skúšať pohnúť. (Rozdiel oproti predchádzajúcim dvom možnostiam je v tom, že to, kam sa zrnko pohne, neurčujeme podľa jeho okolia v okamihu, keď ním ideme hýbať, ale čisto podľa jeho polohy po predošlom takte.) Ktorá z týchto možností je najlepšia? To už ponecháme na vaše subjektívne posúdenie.

TRIDSIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek, Vladimír Macko
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2014