A-II-1 Tréning

Úloha má viacero rôznych riešení „hrubou silou“. Na tri body stačilo samostatne vyskúšať každý dostatočne dlhý úsek. Štyri body ste mohli získať, ak ste predchádzajúce riešenie trochu vylepšili: ak už vieme, kde má maximum úsek $a_i, \ldots, a_j$, ľahko v konštantnom čase zistíme, kde má maximum úsek $a_i, \ldots, a_{j+1}$.

Piaty bod sa dal získať za jednoduché, ale veľmi užitočné pozorovanie: Namiesto úsekov dĺžky **aspoň** k stačí uvažovať úseky dĺžky **presne** k. Ak totiž vieme, že nejaká hodnota x je maximom nejakého dlhého úseku, tak zjavne platí, že toto isté x ostane maximom aj ak ten úsek z ľubovoľného konca skrátime (samozrejme tak, aby x stále v úseku ostalo).

Úsekov dĺžky k je len $n - k + 1$. Ak budeme každý z nich kontrolovať zvlášť, dostaneme riešenie s časovou zložitosťou $O(k(n - k + 1))$, čo môžeme zhora odhadnúť jednoduchšie ako $O(nk)$.

Listing programu (Python)

```
N, K = [ int(_) for _ in input().split() ]
A    = [ int(_) for _ in input().split() ]

mozne_maxima = set()
for zaciatok in range(N-K+1): mozne_maxima.add( max( A[zaciatok:zaciatok+K] ) )
print( len(mozne_maxima) )
```

Skoro-vzorové riešenia

Existuje veľa spôsobov, ako súťažnú úlohu vyriešiť v časovej zložitosti $O(n \log n)$ alebo $O(n \log k)$. Za ľubovoľnú z týchto dvoch časových zložitostí sa dalo získať 8 bodov.

Jedno možné riešenie je založené na pozorovaní, že keď sme práve spracovali úsek $a_i, \ldots, a_{i+k-1}$ a chceme spracovať úsek $a_{i+1}, \ldots, a_{i+k}$ tak potrebujeme spraviť len dve zmeny: zo spracúvaného úseku odstrániť hodnotu a_i a naopak doň pridať novú hodnotu a_{i+k}.

Ak si prvky aktuálneho úseku budeme udržiavať v usporiadanej množine, budeme vedieť v čase $O(\log k)$ pridať novú hodnotu, odstrániť odoberanú hodnotu, aj zistiť, akú hodnotu má aktuálne maximum.

Listing programu (C++)

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

int main() {
    int N, K; cin >> N >> K;
    vector<int> A(N); for (int &a:A) cin >> a;

    set<int> aktualny_usek( A.begin(), A.begin()+K );
    int aktualne_max = *aktualny_usek.rbegin();
    int pocet_maxim = 1;

    for (int zaciatok=1; zaciatok+K<=N; ++zaciatok) {
        aktualny_usek.erase( A[zaciatok-1] );
        aktualny_usek.insert( A[zaciatok+K-1] );
        int nove_max = *aktualny_usek.rbegin();
        if (nove_max != aktualne_max) {
            aktualne_max = nove_max;
            ++pocet_maxim;
        }
    }
    cout << pocet_maxim << endl;
}
```

Iné šikovné riešenie je napríklad toto: Spravíme si prázdne pole veľkosti n a nad týmto poľom si postavíme intervalový strom. V každom vrchole stromu si budeme pamätať, či už je niekde pod ním nejaké neprázdne políčko, a ak áno, tak aký je najmenší a najväčší spomedzi indexov takýchto políčok.

Do poľa teraz postupne budeme zapisovať na správne miesta čísla našej postupnosti, ale nie zľava doprava, ale **od najväčšieho po najmenšie**.

Vždy, keď ideme pridať nejaké číslo x na nejaký index i, tak sa najskôr nášho intervalového stromu opýtame dve otázky: kde je najbližšie už zaplnené políčko naľavo a napravo od indexu i? Obe otázky vieme pomocou údajov uložených vo vrcholoch intervalového stromu zodpovedať v logaritmickom čase. A načo nám budú odpovede na tieto otázky? No predsa nám udávajú ľavý a pravý okraj najdlhšieho úseku, ktorého maximom je

práve pridávaná hodnota x. A už len stačí skontrolovať, či je tento úsek dostatočne dlhý, a vieme, či x mohlo alebo nemohlo skončiť na Samkovom svetri.

Vzorové riešenie

Vzorové riešenie bude mať optimálnu časovú zložitosť: $O(n)$, teda lineárnu od veľkosti vstupu. Bude založené na rovnakom pozorovaní ako predchádzajúce riešenie: na to, aby sme zistili, či je konkrétna hodnota x maximom dostatočne dlhého úseku, stačí nájsť najbližšiu väčšiu hodnotu naľavo aj napravo od x.

Nižšie si ukážeme, ako vieme na jeden prechod poľom zľava doprava v lineárnom čase nájsť ku každému prvku poľa najbližší väčší naľavo od neho. Rovnakú funkciu potom ešte raz použijeme na reverz zadanej postupnosti aby sme našli aj najbližšie väčšie prvky napravo od každého prvku.

Budeme postupne zľava doprava spracúvať prvky našej postupnosti. V každom okamihu si budeme pamätať množinu tých prvkov, ktoré ešte môžu byť „najbližším väčším prvkom naľavo“ pre nejaký prvok, ktorý príde v budúcnosti.

Kľúčové pozorovanie je nasledovné: Akonáhle spracujeme prvok s hodnotou x, môžeme zabudnúť na všetky prvky, ktoré sú naľavo od neho a majú menšiu hodnotu. Tieto prvky už v budúcnosti nikdy nebudú pre nikoho najbližším väčším prvkom naľavo, lebo x ich zatieni.

Príklad: predstavme si, že sme už spracovali postupnosť 100, 14, 74, 39, 40, 27 a 12. V tomto okamihu si stačí pamätať len nasledovné hodnoty (a ich indexy): 100, 74, 40, 27 a 12. Ak by nasledujúcim prvkom postupnosti bolo číslo 47, mohli by sme po jeho spracovaní zabudnúť ďalšie tri prvky a pamätali by sme si už len hodnoty 100, 74 a 47.

Spracovanie jedného prvku x teda bude vyzerať nasledovne:

- Zabudneme na všetky prvky, ktoré sú menšie ako x.
- Najmenší z prvkov, ktoré si ešte stále pamätáme, je najbližším väčším prvkom naľavo od x.
- Pridáme x medzi prvky, ktoré si pamätáme.

Teraz si už stačí len všimnúť, že prvky, ktoré si pamätáme, budú automaticky vždy usporiadané od najväčšieho po najmenší – totiž vždy zabúdame od najmenšieho, a v okamihu, keď pridávame nový prvok, je tento najmenším spomedzi práve pamätaných. Preto nepotrebujeme žiadne zložité dátové štruktúry – na uloženie si aktuálne pamätaných prvkov nám stačí obyčajný zásobník.

Odhad časovej zložitosti potom spravíme nasledovne: Každý prvok raz spracujeme a vložíme do zásobníka, a každý prvok najviac raz zo zásobníka vyhodíme. Preto dokopy spravíme pri celom prechode postupnosťou $O(n)$ operácií.

Listing programu (Python)

```
N, K = [ int(_) for _ in input().split() ]
A    = [ int(_) for _ in input().split() ]

def odzadu(A): return list(A)[::-1]

def indexy_vacsich_nalavo(A):
    odpoved = []
    kandidati = []
    for i in range(len(A)):
        # vyhodime primalych kandidatov
        while kandidati != [] and A[kandidati[-1]] < A[i]:
            kandidati.pop()
        # zapiseme si index najblizsieho vacsieho od A[i]:
        odpoved.append( -1 if kandidati == [] else kandidati[-1] )
        # pridame noveho kandidata
        kandidati.append(i)
    return odpoved

nalavo  = indexy_vacsich_nalavo(A)
napravo = odzadu( N-1-x for x in indexy_vacsich_nalavo( odzadu(A) ) )

moze_byt_maximum = [ (napravo[i] - nalavo[i] - 1 >= K) for i in range(N) ]
print( sum(moze_byt_maximum) )
```

Alternatívne vzorové riešenia

Existujú aj iné lineárne riešenia tejto súťažnej úlohy.

Jedno dostaneme tak, že upravíme riešenie, ktoré postupne spracúvalo úseky a pamätalo si ich obsah v usporiadanej množine (sete). Namiesto setu totiž vieme šikovne použiť tzv. obojstrannú frontu (deque). Detaily tohto riešenia nechávame iniciatívnym čitateľom ako domácu úlohu. Napovieme vám ešte, že kľúčové je opäť všimnúť si, že ak práve do úseku pribudla nová hodnota x, tak staré hodnoty menšie ako x už nikdy maximom skúmaného úseku nebudú – a teda ich môžeme spokojne zabudnúť.

Iné lineárne riešenie dostaneme tak, že si vstupnú postupnosť nakrájame na kúsky dĺžky k a v každom kúsku si predpočítame prefixové a sufixové maximá. Z takto získaných údajov už vieme vypočítať maximum ľubovoľného úseku dĺžky k v konštantnom čase.

A-II-2 Telenovela II

Riešenie začneme tým, že rovnako ako v domácom kole ošetríme to, že treba vidieť prvý aj posledný diel: nájdeme prvý výskyt prvého dielu, posledný výskyt posledného, tie označíme ako pozreté a odteraz sa už budeme pozerať len na úsek medzi nimi, vrátane nich (alebo zistíme, že úloha nemá riešenie).

Po tejto úprave už nemusíme nijak špeciálne ošetrovať prvú a poslednú epizódu, zjavne totiž existuje optimálne riešenie, ktoré ich obe použije, takže keď nájdeme dĺžku celkovo najlepšieho riešenia, bude to zároveň dĺžka najlepšieho riešenia, pri ktorom Ondro začne prvou epizódou a skončí poslednou.

Kľúčovou myšlienkou nášho vzorového riešenia teraz bude nasledujúce pozorovanie: Bez ohľadu na to, či je optimálnym riešením rastúca postupnosť alebo postupnosť s jednou výnimkou, optimálne riešenie určite vieme získať tak, že spracúvanú postupnosť vhodne „rozstrihneme“ na dve časti a zvlášť v každej z nich nájdeme čo najdlhšiu rastúcu podpostupnosť. Budeme teda postupovať nasledovne:

1. Postupne pre každé k zistíme, aká najdlhšia rastúca podpostupnosť končí k-tym prvkom našej postupnosti.
2. Cez hodnoty získané v kroku 1 prejdeme zľava doprava a spočítame ich prefixové maximá – teda pre každé k získame dĺžku s_k najdlhšej rastúcej podpostupnosti, ktorá sa dá vybrať z prvých k prvkov.
3. Postupne pre každé k zistíme, aká najdlhšia rastúca podpostupnosť začína k-tym prvkom našej postupnosti. (V tomto kroku ideme cez zadanú postupnosť sprava doľava a robíme to isté ako v kroku 1, len hľadáme klesajúcu postupnosť namiesto rastúcej.)
4. Cez hodnoty získané v kroku 2 prejdeme sprava doľava a spočítame ich sufixové maximá – teda pre každé ℓ získame dĺžku t_ℓ najdlhšej rastúcej podpostupnosti, ktorá sa dá vybrať z posledných ℓ prvkov vstupu.
5. Optimálnym riešením je najväčšia z hodnôt $s_k + t_{n-k}$. V lineárnom čase vyskúšame všetky možné k a vyberieme to najlepšie.

V podstate jedinou netriviálnou časťou tohto riešenia je krok 1. (A tiež krok 3, ktorý je jeho kópiou. V našej implementácii na oba šikovne použijeme tú istú funkciu.) Algoritmus na riešenie tohto kroku sme si už ale ukázali vo vzorových riešeniach domáceho kola.

Pripomeňme si, ako sme v domácom kole hľadali najdlhšiu rastúcu podpostupnosť. Udržiavali sme si hodnoty c_j s nasledovným významom: keď sa pozrieme na všetky možné rastúce j-prvkové podpostupnosti v už spracovaných dátach a zoberieme posledný prvok každej z nich, najmenšia z takto získaných hodnôt bude práve c_j. Špeciálne budeme mať $c_0 = -\infty$ (za postupnosť dĺžky 0 sa dá pridať čokoľvek) a $c_j = +\infty$ ak ešte v spracovaných dátach žiadna rastúca j-prvková podpostupnosť neexistuje.

Ukázali sme si, že vždy, keď prečítame a spracujeme nasledujúci prvok postupnosti, zmení sa nám nanajvýš jedna z hodnôt c_i. Teraz si už len stačí uvedomiť navyše to, že práve index tejto hodnoty nám povie dĺžku najdlhšej rastúcej podpostupnosti, ktorá končí práve spracovaným prvkom.

Príklad: ak sme už spracovali $(10, 3, 8, 6, 9, 4, 6, 22, 8, 5)$ tak budeme mať $c_1 = 3$, $c_2 = 4$, $c_3 = 5$, $c_4 = 8$, a $c_5 = +\infty$. Všimnite si, že rôzne c_i môžu zodpovedať rôznym podpostupnostiam. Napríklad v tejto chvíli najlepšou podpostupnosťí dĺžky 3 je $(3, 4, 5)$, zatiaľ čo pre dĺžku 4 to je $(3, 4, 6, 8)$.

Ak by nasledujúci prvok postupnosti bol $x = 7$, prvá hodnota c_i, ktorá je väčšia alebo rovná ako x je c_4. Túto hodnotu teda zmenšíme z 8 na 7. Zároveň sme ale zistili, že najdlhšia rastúca podpostupnosť končiaca týmto x má dĺžku práve 4. Ak by sme namiesto toho mali $x = 100$, menili by sme hodnotu c_5, čiže sme práve zistili, že najdlhšia postupnosť končiaca práve spracúvanou hodnotou 100 má dĺžku 5. Ak by bolo $x = 5$, nemenilo by sa nič, lebo práve máme $c_3 = 5$. Našli sme teda len nový spôsob ako spraviť rastúcu postupnosť dĺžky 3 končiacu číslom 5.

Kroky 1 a 3 vieme teda takto implementovať v čase $O(n \log n)$, ostatné kroky v lineárnom čase. Celková časová zložitosť tohto riešenia je teda $O(n \log n)$.

Listing programu (C++)

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

// v čase n log n vypočítame dĺžku najdlhšej rastúcej podpostupnosti pre každý prefix A
vector<int> dlzky_lis(const vector<int> &A) {
    // inicializujeme si C tak, aby C[0]=-inf; v každom okamihu si budeme pamätať len tú časť C ktorá je < inf.
    vector<int> C(1, -1);
    vector<int> odpoved;
    unsigned najviac = 0;
    for (int a : A) {
        // nájdeme najmenšie i také, že C[i] >= a
        unsigned i = upper_bound( C.begin(), C.end(), a-1 ) - C.begin();
        // upravíme hodnotu C[i] na a
        if (i == C.size()) C.push_back(a); else C[i] = a;
        najviac = max( najviac, i );
        odpoved.push_back(najviac);
    }
    return odpoved;
}

int main() {
    int N, E;                cin >> N >> E;
    vector<int> epizody(N); for (int n=0; n<N; ++n) cin >> epizody[n];

    // nájdeme prvé vysielanie prvej a posledné vysielanie poslednej epizódy
    int prva = 0, posledna = N-1;
    while (prva < N && epizody[prva] != 1) ++prva;
    while (posledna >= 0 && epizody[posledna] != E) --posledna;

    // ak sa nestíhajú oba alebo niektorú vôbec nevysielajú, riešenie neexistuje
    if (posledna < prva) { cout << -1 << endl; return 0; }

    // zostrojíme si postupnosť, v ktorej ideme naozaj hľadať
    vector<int> ostalo( epizody.begin()+prva, epizody.begin()+posledna+1 );

    // zistíme optimálnu dĺžku rastúcej podpostupnosti pre každý prefix
    vector<int> S = dlzky_lis(ostalo);

    // obrátime postupnosť, prečíslujeme epizódy naopak a použijeme tú istú funkciu
    // na nájdenie optimálnej dĺžký rastúcej podpostupnosti pre každý sufix
    reverse( ostalo.begin(), ostalo.end() );
    for (int &e : ostalo) e = E+1-e;
    vector<int> T = dlzky_lis(ostalo);

    // nájdeme najlepší spôsob ako našu postupnosť rozdeliť na prefix a sufix
    int odpoved = 0;
    for (int i=1; i<int(ostalo.size()); ++i) odpoved = max( odpoved, S[i-1]+T[ostalo.size()-i] );
    cout << odpoved << endl;
}
```

Alternatívne pomalšie riešenie

Ukážeme si ešte riešenie, ktoré bude ľahšie na implementáciu, ale pobeží v čase $O(n^2)$. Upravíme pomalšie riešenie z domáceho kola.

Rovnako ako v domácom kole označme b_i dĺžku najdlhšej rastúcej podpostupnosti, ktorej posledný prvok je na indexe i. Navyše teraz označme c_i dĺžku najdlhšej rastúcej podpostupnosti s hľadanou vlastnosťou (teda rastúcej až na možno jednu výnimku), ktorej posledný prvok je na indexe i.

Hodnoty b_i už vieme počítať. Hodnoty c_i vypočítame nasledovne: ak je posledný prvok na indexe i a predposledný na indexe $j < i$ tak sú dve možnosti: ak $a_j < a_i$, môžeme zobrať najdlhšiu postupnosť s najviac jednou

výnimkou ktorá končí na indexe j a za ňu pridať prvok na indexe i. Ak $a_j \geq a_i$, tak tým, že za prvok na indexe j pridáme prvok na indexe i vyrobíme výnimku. Od začiatku po prvok na indexe j preto môžeme zobrať len postupnosť, ktorá je čisto rastúca.

Dokopy teda dostávame nasledovné riešenie:

Listing programu (C++)

```
// vstup máme uložený v premennej N a v poli hodnôt A[0..N-1]
vector<int> B(N), C(N);

for (int i=0; i<N; ++i) {
    B[i] = C[i] = 1;
    for (int j=0; j<i; ++j) if (A[j] < A[i]) B[i] = max( B[i], 1+B[j] );
    for (int j=0; j<i; ++j) {
        if (A[j] < A[i]) C[i] = max( C[i], 1+C[j] );
        else             C[i] = max( C[i], 1+B[j] );
    }
}
```

A-II-3 Varenie

Táto súťažná úloha úzko súvisí s úlohami o najkratších cestách v grafoch. Všetky algoritmy, ktoré si budeme ukazovať, budú priamymi analógiami grafových algoritmov.

Optimálne varenie je acyklické

Predstavme si, že už poznáme pre každé jedlo najlacnejšiu cenu jednej jeho porcie. Ak si jedlá podľa tejto ceny usporiadame, tak zjavne bude platiť, že pri optimálnej príprave konkrétneho jedla sa ako ingrediencie oplatí použiť len jedlá „naľavo“ od neho, teda tie s menšou cenou.

Ak by nám niekto správne poradie jedál (to podľa optimálnej ceny) prezradil, vedeli by sme už súťažnú úlohu vyriešiť ľahko: išli by sme po správnom poradí zľava doprava a počítali by sme optimálne ceny.

Zistenie optimálnej ceny pre konkrétne jedlo by bolo jednoduché: buď ho priamo kúpime, alebo ho uvaríme podľa niektorého receptu. A ak je optimálne ho uvariť, v tejto chvíli už poznáme optimálne ceny oboch ingrediencií a z tých hravo vypočítame optimálnu cenu práve spracúvaného jedla.

V tomto okamihu už vieme implementovať prvé korektné, aj keď dosť zúfalo pomalé riešenie: vyskúšame všetkých $n!$ možných poradí jedál a pre každé z nich použijeme vyššie popísaný postup. Najlepší spôsob, ako navariť jedlo číslo 1, je potom minimom cez všetkých $n!$ možností.

Bellmanov-Fordov algoritmus

V tejto časti si ukážeme jednoduchý polynomiálny algoritmus riešiaci našu súťažnú úlohu.

Pre každé jedlo i bude b_i označovať najlepšiu cenu, za ktorú ho aktuálne vieme navariť. Začneme tým, že každé b_i nastavíme na cenu c_i, za ktorú vieme príslušné jedlo kúpiť v obchode.

Výpočet teraz bude až prekvapivo jednoduchý: postupne $(n-1)$-krát prejdeme cez celý zoznam receptov a pre každý z nich skontrolujeme, či pomocou neho nevieme zlepšiť hodnotu b pre jedlo, ktoré vyrába.

Ak teda máme recept, ktorý z jedál s_i a t_i navarí jedlo u_i, pozrieme sa, či $b_{s_i} + b_{t_i}$ nie je menej ako doterajšia hodnota b_{u_i}. A ak áno, hodnotu b_{u_i} zmenšíme na práve nájdenú lepšiu cenu.

Je zjavné, že pre n jedál a r receptov má tento algoritmus časovú zložitosť $O(nr)$. Naozaj ale funguje?

Dokážeme si, že keď algoritmus skončí, všetky hodnoty b_i zodpovedajú skutočne najlacnejším cenám, za ktoré sa dajú jednotlivé pokrmy navariť.

Ako v minulej úvahe, aj teraz si predstavme, že nám už niekto zoradil pokrmy do poradia $p_0, \ldots, p_{n-1}$ podľa optimálnej ceny, za ktorú ich vieme vyrobiť.

Zjavne platí $b_{p_0} = c_{p_0}$, teda najlacnejší pokrm je najlepšie rovno kúpiť. Teda už na začiatku výpočtu (po inicializácii hodnôt b_i na c_i) poznáme optimálnu cenu pokrmu p_0.

No a teraz si už len stačí všimnúť, že každým prechodom cez všetky recepty už správne určíme cenu aspoň jedného ďalšieho pokrmu v našom optimálnom poradí: po prvom prechode cez recepty už budeme poznať optimálnu cenu pre p_1, po druhom aj pre p_2, po treťom aj pre p_3, a tak ďalej.

Dôvod je presne rovnaký ako v úvahe v predchádzajúcej časti riešenia: Nech platí, že po i prechodoch cez zoznam receptov už poznáme optimálne ceny pre pokrmy $p_0, \ldots, p_i$. Keď budeme ešte raz prechádzať všetky recepty, vyskúšame (okrem iného) všetky spôsoby ako z už optimálne vyriešených pokrmov navariť pokrm p_{i+1}, a teda určite nájdeme aj optimálne riešenie pre tento pokrm.

Môže sa samozrejme stať, že v jednom prechode cez recepty nájdeme optimálne riešenia pre viaceré (možno dokonca rovno pre všetky) jedlá. Namiesto toho, že budeme fixne robiť $n - 1$ prechodov cez zoznam receptov, by sme náš algoritmus mohli sformulovať aj nasledovne: dokola prechádzaj cez zoznam receptov dovtedy, kým sa pri nejakom prechode už žiadna z hodnôt b_i nezmení. Takto implementovaný Bellmanov-Fordov algoritmus bude mať stále v najhoršom prípade časovú zložitosť $\Theta(nr)$, ale v praxi bude často o čosi rýchlejší.

Dijkstrov algoritmus

Je asi dosť zjavné, čo je pomalé na predchádzajúcom algoritme: v každej „fáze“ znova a znova prechádzame úplne všetky recepty, pričom mnohé nám nijak nepomôžu – niektoré preto, že už sme ich pre aktuálne vstupy použili, niektoré zas preto, že pre ich vstupy ešte nepoznáme optimálne ceny.

Omnoho lepšie by bolo, keby sme poznali správne poradie jedál podľa optimálnej ceny, potom by stačilo každý recept vyskúšať použiť len raz.

Na prvý pohľad to znie ako začarovaný kruh – to poradie sa predsa snažíme zistiť, nie? Trik bude v tom, že to poradie nepotrebujeme poznať hneď celé. Počas behu programu ho budeme postupne zostrojovať.

Počas behu tohto nového algoritmu budeme jedlá postupne označovať ako hotové. V okamihu, keď nejaké jedlo označíme za hotové, budeme si istí, že jeho hodnota b_i už zodpovedá optimálnej cene jeho výroby.

Jedlá, ktoré ešte nie sú hotové, budeme nazývať kandidátmi. Pre každého kandidáta i bude platiť, že b_i je minimum z dvoch možností: jeho nákupnej ceny c_i a najlacnejšieho spôsobu, ako jedlo i vyrobiť len pomocou samých hotových jedál.

Na začiatku behu programu budú všetky jedlá kandidátmi. Keďže ešte nemáme hotové jedlá, jediným povoleným spôsobom výroby je priama kúpa, a teda pre každé jedlo platí $b_i = c_i$.

Hlavnou myšlienkou algoritmu bude nasledovné pozorovanie: Kandidáta, ktorého b_i je najmenšie spomedzi všetkých kandidátov, môžeme prehlásiť za hotového.

Prečo? No lebo už jeho cenu nevieme nijak zlepšiť. Za cenu menšiu ako súčasné b_i nevieme ani navariť nič iné z už hotových jedál, ani nijaké iné jedlo kúpiť.

Teraz teda vieme o ďalšom jedle, ktoré môžeme prehlásiť za hotové. Čo ale presne treba spraviť, keď sa tak stane? Potrebujeme zabezpečiť, aby opäť platilo, že pre ostatných kandidátov poznáme najlacnejší spôsob, ako ich vyrobiť pomocou hotových jedál. Tým, že nám pribudlo hotové jedlo, pribudli nám možno nejaké nové, lacnejšie spôsoby ako navariť iných kandidátov. Tieto musíme prezrieť. Našťastie ich nie je až tak veľa – stačí nám prezrieť tie recepty, ktoré priamo zahŕňajú ako surovinu to konkrétne jedlo, ktoré sme práve prehlásili za hotové.

Počas tohto nového algoritmu teda každé jedlo práve raz niekedy vyhlásime za hotové a každý recept práve dvakrát spracujeme (vždy, keď vyhlásime za hotovú jednu z dvoch ingrediencií, ktoré sa v ňom používajú).

Potrebujeme ešte vedieť efektívne robiť dve veci: Za prvé, potrebujeme k danému jedlu vedieť prejsť všetky recepty, v ktorých sa používa. Toto je ľahké, už pri načítaní vstupu si vieme recepty roztriediť do samostatných zoznamov pre každé jedlo.

Za druhé, v každom „kole“ potrebujeme vedieť nájsť, ktorý recept spomedzi kandidátov môžeme prehlásiť za hotový.

Toto môžeme robiť hrubou silou: v každom kole prejdeme všetkých kandidátov a takto nájdeme, ktorý z nich má najmenšie b_i. Keďže kôl, rovnako ako jedál, bude n a v každom strávime hľadaním $O(n)$ času, bude mať toto riešenie časovú zložitosť $O(n^2 + r) = O(n^2)$.

Šikovnejšie je ale použiť lepšiu dátovú štruktúru. Kandidátov si napríklad môžeme udržiavať usporiadaných podľa aktuálnej hodnoty b_i. Presnejšie, budeme mať usporiadanú množinu, v ktorej budú záznamy tvaru (b_i, i). Pri takejto implementácii vieme v čase $O(\log n)$ nájsť najlacnejšieho spomedzi kandidátov – je ním aktuálne minimum. Naopak sa nám trochu spomalí spracovanie konkrétneho receptu. Totiž ak nejakému jedlu chceme zmenšiť jeho hodnotu b_i, musíme to spraviť aj s jeho záznamom v našej usporiadanej množine. To sa najľahšie robí tak, že zmažeme starý záznam a pridáme nový. Takáto implementácia má časovú zložitosť $O((n+r)\log n)$.

(Existuje aj o chlp lepšia implementácia s časovou zložitosťou $O(n \log n + r)$, tá však používa pokročilé dátové štruktúry a získané zlepšenie už nie je príliš zaujímavé. Rovnako dobrá časová zložitosť ako má naše vzorové riešenie sa dá dosiahnuť aj implementáciou pomocou prioritnej fronty. Tá sa od našej líši tým, že záznamy nemažeme. Ak nejakú hodnotu b_i zlepšíme, jednoducho do prioritnej fronty vložíme aj nový, lepší záznam. Pri tomto riešení môže prioritná fronta narásť až na $O(r)$ záznamov, toto nám však časovú zložitosť nepokazí.)

Listing programu (C++)

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

typedef struct { int par, vystup; } recept;
typedef struct { long long cena; int jedlo; } zaznam;

bool operator< (const zaznam &A, const zaznam &B) {
    return A.cena < B.cena || (A.cena == B.cena && A.jedlo < B.jedlo);
}

int main() {
    int N, M;
    cin >> N >> M;
    vector<long long> nakup(N);
    for (int n=0; n<N; ++n) cin >> nakup[n];
    vector< vector<recept> > kucharka(N);
    for (int m=0; m<M; ++m) {
        int vysledok, surovina1, surovina2;
        cin >> vysledok >> surovina1 >> surovina2;
        --vysledok; --surovina1; --surovina2;
        kucharka[surovina1].push_back( {surovina2,vysledok} );
        kucharka[surovina2].push_back( {surovina1,vysledok} );
    }

    set<zaznam> Q;
    for (int n=0; n<N; ++n) Q.insert( {nakup[n],n} );
    vector<long long> best = nakup;

    while (!Q.empty()) {
        int mam = Q.begin()->jedlo;
        Q.erase( Q.begin() );
        for (const recept &r : kucharka[mam]) {
            if (best[mam] + best[r.par] < best[r.vystup]) {
                Q.erase( { best[r.vystup], r.vystup } );
                best[r.vystup] = best[mam] + best[r.par];
                Q.insert( { best[r.vystup], r.vystup } );
            }
        }
    }

    cout << best[0] << endl;
}
```

A-II-4 Stavebnica funkcií

Postupne si ukážeme riešenia jednotlivých súťažných podúloh.

Podúloha A: umocňovanie

Podobne ako sčítanie bolo opakovaným použitím nasledovníka a násobenie opakovaným použitím sčítania, umocňovanie je opakovaným použitím násobenia. Výpočet x^y si môžeme predstaviť tak, že začneme z hodnoty 1 a tú postupne y-krát vynásobíme hodnotou x.

Tu ale vidíme jeden drobný rozdiel. Sčítanie aj násobenie boli komutatívne, takže napríklad u násobenia bolo jedno, či x-krát sčítame y alebo y-krát sčítame x. U umocňovania už na poradí parametrov záleží.

A ako na potvoru, my by sme chceli nejaký postup opakovať y-krát, Cyklovač ale vie ako počet opakovaní použiť len prvý parameter, nie druhý. Postupovať budeme podobne ako pri výrobe odčítania: najskôr si vyrobíme pomocnú funkciu *wop*, ktorá robí umocňovanie, ale má opačné poradie parametrov – teda $\forall x, y : wop(x, y) = y^x$.

Hľadanú funkciu *wop* chceme vyrobiť pomocou Cyklovača. Poďme si teda zistiť, aké dve funkcie f a g doň musíme vložiť, aby nám von vypadlo práve takéto umocňovanie. Keď do Cyklovača vložíme unárnu funkciu f a ternárnu funkciu g, dostaneme nasledujúcu binárnu funkciu:

```
def wop(x,y):
    tmp = f(y)
    for i = 0 to x-1:
        tmp = g(i,y,tmp)
    return tmp
```

Pre $x = 0$ táto funkcia vráti hodnotu $f(y)$. No a keďže hocičo umocnené na nultú je jedna, potrebujeme ako f použiť unárnu konštantnú funkciu ktorá vždy vráti hodnotu 1, čiže funkciu k_1^1. Program sa nám teda upravil nasledovne:

```
def mul(x,y):
    tmp = 1
    for i = 0 to x-1:
        tmp = g(i,y,tmp)
    return tmp
```

My by sme teraz chceli, aby každá iterácia cyklu vynásobila pomocnú premennú *tmp* premennou y. Preto ako g chceme použiť funkciu **troch** premenných, ktorá na výstup vráti súčin druhého a tretieho vstupu. To je **skoro**, ale nie úplne, funkcia *mul*. Ale my už predsa vieme, ako pomocou Kompozítora upraviť počet a poradie vstupov funkcie. Hľadanú funkciu g vyrobíme napríklad nasledovne: $g \equiv K[v_2^3, v_3^3, mul]$.

Dokopy teda dostávame, že $wop \equiv C[\, k_1^1,\ K[v_2^3, v_3^3, mul]\,]$.

A už len treba vymeniť poradie parametrov: $pow \equiv K[v_2^2, v_1^2, wop]$.

Podúloha B: signum

Signum je najľahšie zostrojiť drobným trikom pomocou Cyklovača. Premennú *tmp* na začiatku nastavíme na 0 a v každom cykle ju potom nastavíme na 1. Ak sa teda nevykonal ani jeden cyklus, vrátime nulu, ak sa vykonala aspoň jedna iterácia cyklu, vrátime jednotku.

Na korektné použitie Cyklovača teda potrebujeme funkciu f **s aritou 0** ktorá vráti nulu a funkciu g **s aritou 2** ktorá vždy vráti jednotku. Platí teda: $sgn \equiv C[z, k_1^2]$.

Podúloha C: predikát „väčší alebo rovný“

Ako vieme zistiť, či je x ostro väčšie ako y? Od x odčítame y a pozrieme sa, či nám ešte niečo ostalo.

Ako vieme zistiť, či je x väčšie alebo rovné ako y? Keďže aj x aj y sú nezáporné celé čísla, stačí ku x pridať jednotku a potom sa pozrieť, či je nové x väčšie ako y.

Postupujme teda nasledovne:

- Pomocná funkcia $f_1 \equiv K[v_1^2, s]$ má dva vstupy, zoberie prvý z nich, pripočíta k nemu jednotku a získanú hodnotu dá na výstup.
- Pomocná funkcia $f_2 \equiv K[f_1, v_2^2, sub]$ má dva vstupy. K prvému z nich pripočíta jednotku a potom od neho odčíta druhý z nich.
- My už vieme, že f_2 vráti kladnú hodnotu práve vtedy, ak jej prvý vstup bol väčší alebo rovný ako druhý. Jediná úprava, ktorú ešte potrebujeme spraviť, je, že vo všetkých tých prípadoch chceme na výstup vracať hodnotu 1. To je ale triviálne: stačí na jej výstup použiť funkciu signum zostrojenú v podúlohe B.

  Predikát *geq* teda zostrojíme nasledovne: $geq \equiv K[f_2, sgn]$.

Podúloha D: vetvenie

V tejto podúlohe sme mali zobrať neznáme, ale už zostrojené funkcie f_0, f_1 a predikát g a mali sme ukázať, ako z nich zostrojiť funkciu h definovanú nasledovným predpisom:

$$h(x_1, \ldots, x_n) = \begin{cases} f_0(x_1, \ldots, x_n) & \text{ak } g(x_1, \ldots, x_n) = 0 \\ f_1(x_1, \ldots, x_n) & \text{ak } g(x_1, \ldots, x_n) = 1 \end{cases}$$

Začneme tým, že si definujeme funkciu *not*, ktorá bude robiť logický not – teda pre nulu vráti jednotku a pre jednotku (a tiež pre akýkoľvek iný kladný vstup) vráti nulu. Táto funkcia je veľmi podobná funkcii signum a vieme ju definovať presne rovnakým spôsobom: $not \equiv C[k_1^0, k_0^2]$.

Teraz už môžeme priamo zostrojiť h. Pomôžeme si jednoduchým trikom. Výber z dvoch hodnôt spravíme tak, že obe sčítame, ale tú z nich, ktorú práve nechceme, vynásobíme nulou. Inými slovami, uvedomíme si, že platí nasledujúci vzťah:

$$h(x_1, \ldots, x_n) \;=\; not(g(x_1, \ldots, x_n)) \cdot f_0(x_1, \ldots, x_n) \;+\; g(x_1, \ldots, x_n) \cdot f_1(x_1, \ldots, x_n)$$

Prečo? Ak pre nejaké $x_1, \ldots, x_n$ nám predikát g vráti nulu, zjednoduší sa náš výraz na

$$h(x_1, \ldots, x_n) \;=\; 1 \cdot f_0(x_1, \ldots, x_n) \;+\; 0 \cdot f_1(x_1, \ldots, x_n) \;=\; f_0(x_1, \ldots, x_n)$$

a to je presne to, čo sme chceli. No a ak g vráti jednotku, vďaka násobeniu nulou zmizne zase druhý člen:

$$h(x_1, \ldots, x_n) \;=\; 0 \cdot f_0(x_1, \ldots, x_n) \;+\; 1 \cdot f_1(x_1, \ldots, x_n) \;=\; f_1(x_1, \ldots, x_n)$$

Hľadaná funkcia h bude teda súčtom dvoch funkcií a každá z nich bude súčinom dvoch funkcií s aritou n. Formálne môžeme písať:

$$h \equiv K[\; \underbrace{K[\,K[g, not], f_0, mul\,]}_{\text{toto je } not(g)\cdot f_0},\; K[\,g, f_1, mul\,],\; add\,]$$

TRIDSIATY TRETÍ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2018