A-I-1 Trojice

Získať *nejaké* body v tejto úlohe je ľahké: stačí vygenerovať všetky trojice a usporiadať ich podľa súčtu:

```
vector<long long> sucty;
for (int p=0; p<N; ++p)
    for (int q=0; q<p; ++q)
        for (int r=0; r<q; ++r)
            sucty.push_back( A[p] + A[q] + A[r] );
sort( sucty.begin(), sucty.end() );
cout << sucty[K-1] << endl;
```

Takéto riešenie má časovú zložitosť $O(n^3 \log n)$ kvôli triedeniu. Táto časová zložitosť sa dá o malý chlp zlepšiť na $O(n^3)$, a to buď tak, že použijeme efektívnejšie triedenie (napr. radixsort) alebo tak, že namiesto triedenia použijeme lineárny algoritmus na nájdenie k-teho najmenšieho prvku. Keďže však existujú omnoho lepšie riešenia, detaily týchto drobných zlepšení si odpustíme.

Zhruba-kvadratické riešenie

Ako by sa túto úlohu dalo riešiť lepšie? Jednou inou technikou, ktorá vedie k lepšiemu riešeniu, je napríklad *binárne vyhľadávanie odpovede*.
Použime nové značenie $\varphi(x)$ pre počet trojíc, ktoré majú súčet najviac x. Súťažnú úlohu si teraz môžeme sformulovať nasledovne: ku danému k treba nájsť najmenšie x, pre ktoré je $\varphi(x) \geq k$. (Slovne: existuje aspoň k trojíc z ktorých každá má súčet x alebo menej.)
Vieme, že hľadaná hodnota x leží niekde medzi 0 a $3 \cdot 10^{12}$. Ak by sme vedeli hodnotu $\varphi(x)$ efektívne počítať, môžeme to správne x nájsť binárnym vyhľadávaním v tomto intervale.

Ako vieme hodnotu $\varphi(x)$ počítať v lepšom ako kubickom čase?
Uvažujme vyššie popísaný algoritmus generujúci všetky trojice indexov (p, q, r). Keď sme si zvolili p a q, dostávame nasledovnú otázku: koľko rôznych možností pre r nám dá dostatočne malý súčet? Keďže čísla na vstupe máme usporiadané od najmenšieho po najväčšie, namiesto skúšania všetkých možných r stačí nájsť najväčšie vyhovujúce r. Toto vieme spraviť v logaritmickom čase (opäť) binárnym vyhľadávaním. Takto teda vieme konkrétnu hodnotu $\varphi(x)$ vypočítať v čase $O(n^2 \log n)$. A keďže správne x hľadáme v intervale konštantnej dĺžky, budeme potrebovať vypočítať len konštantne veľa hodnôt φ, a teda celková asymptotická časová zložitosť nášho riešenia bude tiež $O(n^2 \log n)$.
Ešte trochu lepšie riešenie sa zaobíde bez vnútorného binárneho vyhľadávania. Namiesto toho si stačí všimnúť, že keď pre pevne zvolené p postupne zväčšujeme q, tak sa postupne zmenšuje maximálne vyhovujúce r. Presnejšie, sú len dve možnosti: buď je maximálne vyhovujúce r ešte stále rovné $q - 1$, alebo je už $A[p] + A[q]$ také veľké, že trojica indexov $(p, q, q - 1)$ už nevyhovuje. Akonáhle nastane táto druhá situácia, tak už bude platiť, že keď zväčšujeme q, tak sa nám nikdy nezväčší maximálne vyhovujúce r. A preto si stačí udržiavať premennú s maximálnym vyhovujúcim r a vždy, keď zväčšíme q, tak túto premennú podľa potreby zmenšovať. Takéto riešenie má časovú zložitosť $O(n^2)$.

Listing programu (C++)

```
long long lo = A[0]+A[1]+A[2]-1, hi = A[N-1]+A[N-2]+A[N-3];
// invariant: je primálo trojíc so súčtom <= lo, je dosť trojíc so súčtom <= hi
while (hi - lo > 1) {
    long long med = (lo+hi) / 2;
    // ideme vypočítať, koľko trojíc má súčet <= med
    long long cnt = 0;
    for (int p=2; p<N; ++p) {
        int max_r = 0;
        for (int q=1; q<p; ++q) {
            // upravíme hodnotu max_r: najväčšie r také, že A[p]+A[q]+A[r] <= med
            if (A[p] + A[q] + A[q-1] <= med) {
                max_r = q-1;
            } else {
                while (max_r >= 0 && A[p] + A[q] + A[max_r] > med) --max_r;
            }
            cnt += (max_r + 1);
        }
    }
    // podľa výsledku zmenšíme na polovicu interval v ktorom hľadáme odpoveď
    if (cnt >= K) hi = med; else lo = med;
}
cout << hi << endl;
```

Vzorové riešenie

Doterajšie riešenia ešte nijak nevyužili jeden z predpokladov uvedených v zadaní: skutočnosť, že k je rozumne malé. Vzorové riešenie túto skutočnosť využíva, a to nasledovne: k-ty najmenší súčet nájdeme tak, že postupne vygenerujeme všetkých k najmenších súčtov. Presnejšie, vygenerujeme im zodpovedajúce trojice indexov (p, q, r), pričom tentokrát bude vždy platiť $p < q < r$.
Začať je ľahké: najmenší súčet zjavne zodpovedá trojici indexov $(0, 1, 2)$. Aj druhý krok je dokonca ešte vždy rovnaký: druhý najmenší súčet zodpovedá trojici $(0, 1, 3)$. Ako ale pokračovať?
Naše vzorové riešenie bude fungovať nasledovne: v každom okamihu budeme mať množinu *kandidátov*: trojíc, ktoré môžu byť nasledujúcou trojicou s najmenším súčtom spomedzi všetkých ešte nespracovaných trojíc.
Na začiatku sme ešte nevygenerovali žiadnu trojicu a máme jediného kandidáta: trojicu $(0, 1, 2)$.
Teraz k-krát zopakujeme nasledujúci postup:

1. Spomedzi všetkých kandidátov vyberieme tú trojicu indexov, ktorej zodpovedá najmenší súčet. (Ak je takých viac, tak ľubovoľnú z nich.) Vybranú trojicu odstránime spomedzi kandidátov. Ak je toto už k-ta spracovaná trojica, vypíšeme na výstup jej zodpovedajúci súčet a skončíme.

2. Ak sme práve spracovali trojicu (a, b, c), pridáme medzi kandidátov nasledujúce trojice: $(a + 1, b, c)$, $(a, b + 1, c)$ a $(a, b, c + 1)$. Presnejšie, pridáme len tie, ktoré sú platné usporiadané trojice indexov.

Dokážme si teraz, že toto riešenie skutočne funguje, teda že skutočne generuje trojice indexov v správnom poradí. Na to stačí dokázať nasledujúce tvrdenie: v ľubovoľnom okamihu platí, že ak ešte nespracovaná trojica indexov nie je medzi kandidátmi, tak určite nemá najmenší súčet spomedzi všetkých nespracovaných trojíc.
Toto tvrdenie vyplýva z nasledovného pozorovania: trojica indexov (a, b, c) má ostro väčší súčet ako každá z platných trojíc spomedzi $(a - 1, b, c)$, $(a, b - 1, c)$ a $(a, b, c - 1)$. Trojicu (a, b, c) teda chceme spracovať až po všetkých týchto trojiciach. A teda rozhodne platí, že kým žiadna z nich nebola spracovaná, nemá zmysel zaraďovať (a, b, c) medzi kandidátov, keďže vieme o nespracovanej trojici s menším súčtom.
(Technicky by stačilo (a, b, c) zaradiť medzi kandidátov až keď spracujeme *poslednú* spomedzi vyššie spomenutých „o jedno menších“ trojíc, zatiaľ čo my to spravíme už keď spracujeme prvú z nich. To však zjavne nepokazí korektnosť nášho algoritmu – len máme občas množinu kandidátov o čosi väčšiu ako by bolo nutné. A ako uvidíme nižšie, nepokazí to ani časovú zložitosť.)

Pri udržiavaní množiny kandidátov potrebujeme vedieť efektívne pridávať nových kandidátov a vyberať najlepšieho. Môžeme teda na ich uloženie použiť prioritnú frontu, a tú implementovať buď pomocou haldy alebo pomocou vyvažovaného stromu. V oboch prípadoch bude každá operácia s množinou kandidátov bežať v čase logaritmickom od ich počtu. No a keďže za každého spracovaného kandidáta pridáme nanajvýš troch nových, bude počet kandidátov lineárne rásť s počtom už spracovaných trojíc, a teda bude kandidátov $O(k)$. Následne teda vieme, že každého kandidáta spracujeme v čase $O(\log k)$, čiže toto riešenie má celkovú časovú zložitosť $O(k \log k)$.

Listing programu (C++)

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

int N;
long long K;
vector<long long> A;

struct triple {
    int a,b,c;
    long long sum() const { return A[a]+A[b]+A[c]; }
};

set<triple> candidates;

bool operator<(const triple &X, const triple &Y) {
    long long xs = X.sum(), ys = Y.sum();
    if (xs != ys) return xs < ys;
    return make_tuple(X.a,X.b,X.c) < make_tuple(Y.a,Y.b,Y.c);
}

int main() {
```

```
    cin >> N >> K;
    A.resize(N);
    for (int n=0; n<N; ++n) cin >> A[n];

    candidates.insert( {0,1,2} );
    while (true) {
        triple curr = *candidates.begin();
        candidates.erase( candidates.begin() );
        --K;
        if (K == 0) { cout << curr.sum() << endl; break; }
        if (curr.a+1 < curr.b) candidates.insert( {curr.a+1,curr.b,curr.c} );
        if (curr.b+1 < curr.c) candidates.insert( {curr.a,curr.b+1,curr.c} );
        if (curr.c+1 < N     ) candidates.insert( {curr.a,curr.b,curr.c+1} );
    }
}
```

A-I-2 Telenovela

Riešenie začneme tým, že zabezpečíme, aby sme videli prvý aj posledný diel telenovely. Je zjavné, že prvý diel chceme vidieť čo najskôr a posledný čo najneskôr, aby nám medzi nimi ostalo čo najviac iných dielov. Nájdeme preto prvé vysielanie prvého dielu a posledné vysielanie posledného dielu. Ak niektorý z nich vôbec nevysielajú, alebo ak prvý diel vysielajú len neskôr ako posledný, úloha nemá riešenie. V opačnom prípade nám ostala jednoduchšia verzia súťažnej úlohy: v časti televízneho programu ktorá zodpovedá úseku medzi prvou a poslednou epizódou potrebujeme vybrať čo najviac ostatných epizód v správnom poradí.
Inými slovami, z danej postupnosti čísel chceme vybrať čo najdlhšiu *rastúcu podpostupnosť* tvorenú číslami 2 až $e-1$. Vo zvyšku tohto riešenia si ukážeme dva rôzne efektívne algoritmy ktoré riešia tento problém.

Kvadratické riešenie

Uvažujme nasledujúcu verziu našej úlohy: na vstupe je postupnosť čísel $a_1, \ldots, a_n$ a nás zaujíma dĺžka najdlhšej ostro rastúcej podpostupnosti tejto postupnosti.
Na vyriešenie úlohy použijeme dynamické programovanie. Označme b_i dĺžku najdlhšej rastúcej podpostupnosti, ktorej posledný prvok je na indexe i. Riešením našej úlohy je zjavne $\max_i b_i$, takže na vyriešenie úlohy nám stačí postupne vypočítať všetky hodnoty od b_1 po b_n.
No a ako na to? Ak je na indexe i posledný prvok rastúcej podpostupnosti, sú len dve možnosti: buď má dĺžku 1 (čiže už žiaden iný prvok neobsahuje), alebo má predposledný prvok na nejakom indexe $j < i$. Keďže celá postupnosť má byť rastúca, pripadajú do úvahy len tie j, pre ktoré platí $a_j < a_i$. No a pre každé takéto j platí, že najdlhšia rastúca podpostupnosť, ktorá končí na indexe j, má dĺžku b_j. Ak teda chceme vyrobiť najdlhšiu rastúcu podpostupnosť končiacu na indexe i, musíme spomedzi vhodných j vybrať to s najväčším b_j. Dotyčná podpostupnosť končiaca na indexe j spolu s prvkom na indexe i potom vytvorí rastúcu podpostupnosť dĺžky $b_j + 1$ končiacu na indexe i.
Všetky hodnoty b_i teda vieme vypočítať v čase $O(n^2)$ nasledovným postupom:

Listing programu (C++)

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

    for (int i=0; i<N; ++i) {
        // hodnotu B[i] inicializujeme na 1, čo zodpovedá jednoprvkovej postupnosti
        B[i] = 1;
        // ak je optimálna postupnosť dlhšia, vyskúšame všetky možnosti pre to,
        // kde má predposledný prvok a vyberieme najlepšiu z nich
        for (int j=0; j<i; ++j) {
            if (A[j] < A[i]) {
                B[i] = max( B[i], 1+B[j] );
            }
        }
    }
```

Vzorové riešenie

Existuje viacero rôznych algoritmov, ktoré vedia najdlhšiu rastúcu podpostupnosť nájsť v čase $O(n \log n)$. V tejto časti si ukážeme jeden z nich. Bude zaujímavý aj tým, že je to online algoritmus, ktorý vie fungovať aj

bez toho, aby vopred vedel, akú dlhú postupnosť budeme spracúvať. Jednoducho budeme zo vstupu postupne čítať prvky postupnosti a po každom z nich budeme vedieť, akú najlepšiu podpostupnosť sme mali v doteraz spracovaných dátach.) Základné pozorovanie, na ktorom algoritmus založíme, je nasledovné: Predstavme si, že sme už spracovali nejakú časť vstupnej postupnosti, napríklad $(10, 3, 8, 6, 9, 4, 6, 22, 7, 5)$. V tejto postupnosti mohlo byť veľa rôznych dvojprvkových rastúcich podpostupností, napríklad $(10, 22)$, $(6, 7)$, alebo $(3, 4)$. Teraz nám na vstupe príde nový prvok x. Predstavme si, že chceme vyrobiť trojprvkovú rastúcu podpostupnosť, ktorá končí týmto x. Toto musíme spraviť tak, že zoberieme niektorú dvojprvkovú rastúcu podpostupnosť, ktorú sme mali v už spracovaných dátach, a na jej koniec pridáme x. No a ktorá zo všetkých možných dvojprvkových podpostupností je na to najvhodnejšia? Je zjavné, že ak je napríklad postupnosť $(5, 7, x)$ rastúca, tak aj postupnosť $(3, 4, x)$ bude rastúca, lebo ak $x > 7$, tak tým skôr platí aj $x > 4$. Inými slovami, najlepšia je tá postupnosť, ktorá končí najmenším možným číslom. Každé x, ktoré by „pasovalo“ za inú podpostupnosť, bude pasovať aj za túto.
Budeme si teda udržiavať 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.
Príklad: ak sme už spracovali $(10, 3, 8, 6, 9, 4, 6, 22, 7, 5)$ tak budeme mať $c_1 = 3$, $c_2 = 4$, $c_3 = 5$, $c_4 = 7$, a $c_5 = +\infty$. Všimnite si, že rôzne c_i môžu zodpovedať rôznym podpostupnostiam. Napríklad v tejto chvíli najlepšou podpostupnosťou dĺžky 3 je $(3, 4, 5)$, zatiaľ čo pre dĺžku 4 to je $(3, 4, 6, 7)$.

Nasledujúce pozorovanie, ktoré použijeme: konečné hodnoty c_i sú vždy usporiadané podľa veľkosti v ostro rastúcom poradí. Napríklad vždy platí $c_3 < c_4$, lebo keď zoberiem *najlepšiu* rastúcu podpostupnosť dĺžky 4 a odstránim z nej posledný prvok, tak dostanem *nejakú* (dokonca nie nutne najlepšiu) rastúcu podpostupnosť dĺžky 3, ktorá končí hodnotou ostro menšou ako c_4.
Predstavme si teraz, že prečítame zo vstupu nasledujúci prvok x a zaujíma nás, aká najdlhšia rastúca podpostupnosť končí týmto prvkom. Aby sme to zistili, chceme nájsť najväčšie i také, že $c_i < x$. Potom je zjavné, že najdlhšia rastúca podpostupnosť končiaca práve prečítaným x má dĺžku $i + 1$. No a keďže vieme, že hodnoty c sú usporiadané podľa veľkosti, vieme hľadané i nájsť v logaritmickom čase binárnym vyhľadávaním.

Ostáva nám posledný krok: zistiť, aké nové rastúce podpostupnosti vznikli tým, že sme do postupnosti pridali práve prečítaný prvok x. Presnejšie, chceme zistiť, ktoré z hodnôt c sa nám zmenili a ako.
Ukáže sa, že zmena je vždy len minimálna. Spomeňme si, že v predchádzajúcom kroku sme našli najväčšie i také, že $c_i < x$. Teraz platí:

- Pre každé $j \leq i$ existuje j-prvková rastúca podpostupnosť končiaca prvkom menším ako x. Hodnoty c_1 až c_i sa teda nezmenia.
- Nemáme žiaden spôsob, ako vyrobiť rastúcu podpostupnosť dĺžky $i + 2$ a viac, ktorá by končila práve prečítaným x. Ani hodnoty od c_{i+2} ďalej sa teda nezmenia.
- Tým pádom jediné, čo sa môže zmeniť, je hodnota c_{i+1}. Doteraz sme mohli mať $c_{i+1} > x$, teraz zjavne máme $c_{i+1} = x$.

Napríklad ak by sme pokračovali vo vyššie uvedenom príklade tým, že prečítame zo vstupu ako ďalší prvok postupnosti hodnotu $x = 6$, zmenila by sa hodnota c_4 zo 7 na 6. Ak by sme namiesto toho prečítali $x = 100$, ostala by $c_4 = 7$ a zmenila by sa c_5 z ∞ na 100.

Po prečítaní a spracovaní n prvkov budeme mať nanajvýš n-prvkovú rastúcu podpostupnosť, preto je v ľubovoľnom okamihu len $O(n)$ prvkov postupnosti c ktoré majú konečnú veľkosť. A teda binárne vyhľadávanie v nich beží v čase $O(\log n)$. Každý prvok zo vstupu teda prečítame a spracujeme v logaritmickom čase, a tým pádom je celková časová zložitosť tohto riešenia $O(n \log n)$.

Listing programu (C++)

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

```cpp
// v čase n log n vypočítame dĺžku najdlhšej rastúcej podpostupnosti v A
int lis_nlogn(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);
    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;
    }
    return C.size()-1;
}

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 postupnosť vysielaní zvyšných epizód ktoré stíhame vidieť
    vector<int> ostalo;
    for (int i=prva+1; i<posledna; ++i) {
        if (1 < epizody[i] && epizody[i] < E) {
            ostalo.push_back( epizody[i] );
        }
    }

    // nájdeme v nich najdlhšiu rastúcu podpostupnosť a pripočítame 2 za prvú+poslednú
    cout << 2 + lis_nlogn(ostalo) << endl;
}
```

A-I-3 Reverzy prefixov

Riešenia podúloh si ukážeme v trochu inom poradí: podúlohu C si necháme na koniec.

Podúloha A: usporiadaj pole

Usporiadané pole budeme vyrábať od konca. Pomocou nanajvýš dvoch operácií $flip$ vieme dostať najväčší prvok na koniec poľa. Stačí ho nájsť v poli (nech je na indexe k), spraviť $flip(k)$, nech ho dostaneme na začiatok poľa, a následne spraviť $flip(n)$, nech ho dostaneme na koniec poľa. Od tejto chvíle už budeme robiť $flip$ len s parametrom menším ako n, čím dosiahneme, že najväčší prvok už bude až do konca nepohnute sedieť na konci poľa. Analogicky postupujeme aj ďalej. Takto celé pole usporiadame pomocou nanajvýš $2n - 2$ operácií $flip$. A keďže pred každou z nich potrebujeme nanajvýš $O(n)$ času, má tento algoritmus polynomiálnu (presnejšie, kvadratickú) časovú zložitosť.

Podúloha B: testovanie pre 10 prvkov

V tejto podúlohe máme postupne prejsť všetkých 10! permutácií desiatich prvkov a pre každú z nich odsimulovať deterministický postup zo zadania.
Implementovať postup zo zadania je ľahké. Jediné, čo potrebujeme navyše, je vedieť prejsť cez všetky permutácie. Pri praktickej implementácii sa na to dá použiť knižničnú funkciu (napr. `next_permutation` v C++, alebo `itertools.permutations` v Pythone).
Ak by sme si takúto funkciu potrebovali implementovať sami, dá sa to spraviť tak, že si vyrobíme pomocnú funkciu ktorá z danej permutácie vyrobí nasledujúcu v lexikografickom poradí. Podrobnejší popis implementácie takejto funkcie nájdete napr. v riešení úlohy A-2-3 z 26. ročníka OI.
Príklad implementácie:

Listing programu (Python)

```python
def simuluj(pole):
    krokov = 0
```

```
    while pole[0] != 1:
        k = pole[0]
        pole = pole[:k][::-1] + pole[k:]
        krokov += 1
    return krokov

from itertools import permutations
pocty_krokov = [ ( simuluj(perm), perm ) for perm in permutations(range(1,11)) ]
pocty_krokov.sort()
for row in pocty_krokov[-10:]: print(row)
```

Spustením programu zistíme, že najväčší počet krokov je 38, a to pre jedinú permutáciu: $(5, 9, 1, 8, 6, 2, 10, 4, 7, 3)$.

Podúloha D: vždy skončíme?

Ako už naznačujú naše experimenty, deterministický proces popísaný v zadaní vždy skonverguje k tomu, že sa na začiatok postupnosti dostane číslo 1. Ako to ale dokázať?
Sporom. Predpokladajme, že to neplatí, teda že sa pre nejakú permutáciu číslo 1 nikdy na začiatok nedostane. Predstavme si, že náš proces budeme opakovať do nekonečna. Keďže rôznych permutácií je len konečne veľa, časom sa nejaká celá permutácia zopakuje. A keďže náš proces je deterministický, od tohto okamihu sa celý proces zacyklí a bude sa dookola opakovať tá istá postupnosť permutácií.
Označme k najväčšie číslo, ktoré sa počas tohto cyklu zjaví na začiatku permutácie. Naše tvrdenie teraz dokážeme sporom. Predpokladajme, že $k > 1$. Čo sa stane? Hneď v nasledujúcom kroku spravíme $flip(k)$, čím sa k dostane na index k. Z definície k vieme, že všetky ostatné čísla, ktoré sa počas cyklu zjavia na začiatku permutácie (vrátane toho, ktoré je tam teraz) sú menšie ako k. To ale znamená, že teraz spravíme $flip(\ell)$ pre nejaké $\ell < k$. Takýto flip ale nechá prvok k na mieste, preto aj on musí na začiatok poľa presunúť nejakú hodnotu menšiu ako k. Lenže tu dostávame hľadaný spor – na jednej strane sme predpokladali, že k sa na začiatku zjavuje periodicky, na druhej strane ale vidíme, že k sa už na začiatku nezjaví nikdy, pretože tam už stále budeme mať hodnotu ostro menšiu ako k.
Preto ostáva jediná možnosť: $k = 1$, a teda sa náš proces nutne vždy zacyklí v situácii, v ktorej je na začiatku poľa číslo 1.

(Iná možná formulácia dôkazu vyzerá nasledovne: Nech x je najväčšie číslo ktoré sa aspoň raz zjaví na začiatku permutácie. Ak $x > 1$, tak podobnou úvahou ako vyššie vieme ukázať, že sa x na začiatku zjaví práve raz a odvtedy už bude navždy sedieť nedotknuté na indexe x. No a teraz môžeme zobrať ten okamih t_1, kedy toto nastane, a od neho ďalej zopakovať túto istú úvahu: aké je najväčšie číslo, ktoré sa na začiatku zjaví po čase t_1? Každou iteráciou tejto úvahy sa dostaneme k menšiemu číslu, a keďže tých máme len konečne veľa, po konečnom počte opakovaní tejto úvahy sa dostaneme k tomu, že najväčšie číslo, ktoré sa odteraz vyskytne na začiatku, je už len číslo 1.)

Podúloha C: hľadanie dobrej permutácie

Na koniec tohto vzorového riešenia sme si nechali podúlohu, ktorá naschvál bola formulovaná otvorene: bolo treba nájsť ľubovoľnú dostatočne dobrú permutáciu.
Už z podúlohy B máme implementovanú funkciu, ktorá nám pre konkrétnu permutáciu spočíta počet krokov. Najľahší spôsob, ako získať nejaké body, je jednoducho skúšať túto funkciu spustiť na rôznych náhodných vstupoch a zapamätať si najlepší, ktorý stretneme.
Podľa našich experimentov by malo pár miliónov pokusov skoro určite stačiť na prekročenie 500-krokovej hranice. No a pri pár miliardách pokusov (čiže keď ten istý program necháte bežať cez noc) by ste už mali naraziť na nejakú permutáciu, ktorá potrebuje viac ako 750 krokov.
Existujú aj efektívnejšie spôsoby hľadania. Vieme si napríklad všimnúť, že pre permutácie, ktoré sa líšia len na pár pozíciách, bude často značná časť krokov nášho algoritmu prebiehať rovnako. Preto keď stretneme nejakú permutáciu, ktorá vyžaduje veľa krokov, oplatí sa pozrieť sa aj na iné, ktoré sa na ňu podobajú. Na takomto pozorovaní sa dá založiť napríklad algoritmus, ktorý pri hľadaní dobrých permutácií používa optimalizačnú techniku *hill climbing*. Najlepšia permutácia, ktorú náš šikovnejší program za pár hodín našiel, potrebuje až 899 krokov. Vyzerá nasledovne: (35, 45, 23, 38, 11, 6, 15, 39, 43, 32, 20, 18, 30, 16, 5, 29, 7, 12, 14, 37, 31, 3, 17, 41, 36, 13, 19, 2, 10, 28, 44, 42, 47, 1, 25, 46, 21, 8, 27, 34, 4, 26, 22, 33, 24, 9, 40).

Na záver dodáme, že nám nie je známy žiadny exaktný postup, ktorý by pre dané n zaručene vyrobil permutáciu, ktorá bude potrebovať veľa krokov. Ak nejaký vymyslíte, dajte nám vedieť :)

A-I-4 Stavebnica funkcií

Postupne si ukážeme riešenia všetkých podúloh.

Podúloha A: ternárna nula

Máme zostrojiť funkciu zzz^3, ktorá má tri vstupy, ale bez ohľadu na to, čo na nich dostane, vždy vráti nulu.
V študijnom texte sme si v Príklade 5 zostrojili unárnu konštantnú nulu: funkciu zz^1, ktorá má jeden vstup a vždy vráti nulu.
Najľahší spôsob, ako teraz vyrobiť zzz^3, je pomocou Kompozítora. Zložíme dokopy napríklad vyberaciu funkciu v_1^3 s funkciou zz^1. Vznikne nám tým funkcia s tromi vstupmi. Tie vložíme do funkcie v_1^3, ktorá z nich vypočíta nejakú pomocnú hodnotu. Je jedno akú, lebo tú následne vložíme do funkcie zz^1 a tá na výstup vráti nulu.
(Namiesto v_1^3 sme mohli použiť aj ľubovoľnú inú ternárnu funkciu, konštrukcia by stále fungovala – práve preto, že na výstupe použitej ternárnej funkcie nezáleží.)
Formálne môžeme vyššie popísanú konštrukciu zapísať nasledovne: $zzz^3 \equiv K[v_1^3, zz^1]$.

Iné riešenie tejto podúlohy využíva Cyklovač. V Príklade 5 sme mali postup, ktorým sme z konštantnej funkcie s aritou 0 spravili konštantnú funkciu s aritou 1. Ak analogický postup zopakujeme ešte dvakrát, dostaneme postupne konštantnú funkciu s aritou 2 a následne s aritou 3.
Formálne: najskôr si zostrojíme pomocnú funkciu $zzzz^2 \equiv C[zz^1, v_3^3]$ a z nej potom vyrobíme hľadanú funkciu $zzz^3 \equiv C[zzzz^2, v_4^4]$.

Podúloha B: preusporiadanie parametrov

V tejto podúlohe máme z neznámej funkcie φ^4 vyrobiť novú funkciu ψ^3 definovanú nasledovne: $\forall a, b, c : \psi^3(a, b, c) = \varphi^4(b, a, c, a)$.
Práve na takéto „technické úpravy“ slúžia vyberacie funkcie. Funkciu ψ vyrobíme pomocou Kompozítora. Tomu povieme, že v druhom kroku chceme použiť funkciu φ^4 a že v prvom kroku jej chceme vybrať jednotlivé vstupy vhodnými vyberacími funkciami. Presnejšie, do φ^4 chceme postupne zadať druhý, prvý, tretí, a znova prvý vstup.
Výsledná konštrukcia teda vyzerá nasledovne: $\psi \equiv K[\ v_2^3, v_1^3, v_3^3, v_1^3,\ \varphi^4\ ]$.

Podúloha C: násobenie

Intuitívne, tak ako sčítanie bolo opakovaným použitím nasledovníka, tak násobenie je opakovaným použitím sčítania. Na vyriešenie tejto podúlohy bude teda treba spraviť niečo podobné tomu, čo robíme v študijnom texte v Príklade 4 pri výrobe sčítacej funkcie *add*.
Postupujme teda podobne. Chceme hľadanú funkciu *mul* vyrobiť pomocou Cyklovača. Poďme si teda zistiť, aké dve funkcie f a g doň musíme vložiť, aby nám von vypadlo práve násobenie. Keď do Cyklovača vložíme unárnu funkciu f a ternárnu funkciu g, dostaneme nasledujúcu binárnu funkciu:

```
def mul(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 nula krát čokoľvek je nula, potrebujeme, aby $f(y)$ vždy vrátila nulu. Inými slovami, ako f chceme použiť unárnu konštantnú nulu, teda funkciu zz^1 zo študijného textu.
Program sa nám teda upravil nasledovne:

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

My by sme teraz chceli, aby každé z x použití funkcie g zväčšilo premennú tmp o y. Funkcia g teda musí na výstup vrátiť hodnotu $y+tmp$. Inými slovami, ako g chceme použiť funkciu **troch** premenných, ktorá na výstup vráti súčet druhého a tretieho vstupu. To je **skoro**, ale nie úplne, funkcia add. Našťastie sme už v podúlohe B vymysleli, ako funkciu add upraviť do požadovanej podoby.
Najskôr si teda z funkcie add vyrobíme Kompozítorom vhodnú funkciu $g \equiv K[v_2^3, v_3^3, add]$, a potom už Cyklovačom vyrobíme násobenie: $mul \equiv C[zz^1, g]$.

Podúloha D: predchodca

Na zostrojenie funkcie predchodcu šikovne (dalo by sa povedať, že až trikovo), použijeme Cyklovač. Začiatok je ľahký: predchodcom nuly je nula. Čo ale spraviť pre $x > 0$? Pozrime sa opäť na pseudokód funkcie, ktorú nám Cyklovač vyrobí, ak doň vložíme nulárnu funkciu z a neznámu binárnu funkciu g:

```
def p(x):
    tmp = z()               # t.j. tmp = 0
    for i = 0 to x-1:
        tmp = g(i,tmp)
    return tmp
```

Naše doterajšie použitia Cyklovača doteraz spokojne ignorovali premennú i, teda riadiacu premennú cyklu. Tentokrát ju však použijeme. Všimnime si, že táto premenná postupne nadobúda hodnoty od 0 po $x - 1$. Posledná hodnota i je teda presne hodnota, ktorú chceme vrátiť na výstup. Už je zjavné, ako na to?
Presne tak. Úplne stačí, keď g na výstup vráti vždy svoj prvý vstup. Tým dostaneme nasledujúci pseudokód:

```
def p(x):
    tmp = 0
    for i = 0 to x-1:
        tmp = i
    return tmp
```

Je to síce neefektívne (kto to kedy videl, počítať predchodcu v lineárnom čase, že?), ale robí to to, čo má.
Vidíme teda, že unárnu funkciu predchodcu vieme vyrobiť Cyklovačom z funkcie z a vyberacej funkcie v_1^2. Formálne, $p \equiv C[z, v_1^2]$.

Podúloha E: odčítanie

Na záver domáceho kola si chceme zostrojiť odčítanie, respektíve teda funkciu, ktorá sa naň čo najviac podobá: binárnu funkciu sub takú, že pre $x > y$ je $sub(x, y) = x - y$ a pre $x \leq y$ je $sub(x, y) = 0$.

Základná myšlienka je zjavná: tak, ako je sčítanie opakovaným použitím nasledovníka, tak je odčítanie opakovaným použitím predchodcu. Takto automaticky zabezpečíme aj to, že pre $x < y$ dostaneme ako výsledok nulu. Ak by sme chceli napríklad robiť 3 mínus 7, tak na číslo 3 použijeme 7-krát funkciu p, čím dostaneme nulu, lebo $p(0) = 0$.
Trocha problém je v tom, že Cyklovač vždy použije prvý parameter funkcie ako počet opakovaní. Nevieme teda odčítanie zostrojiť priamo. Vyrobíme si preto najskôr pomocnú funkciu bus, ktorá bude mať parametre v opačnom poradí: $bus(x, y)$ bude y mínus x.

Funkciu bus už vieme vyrobiť Cyklovačom. Pre $x = 0$ platí, že y mínus 0 je y, ako prvú funkciu do Cyklovača teda vložíme identitu. No a druhá funkcia bude, analogicky k Príkladu 4, funkcia $K[v_3^3, p]$ – teda funkcia „zober predchádzajúcu hodnotu tmp a použi na ňu funkciu predchodcu“.
Formálne teda $bus \equiv C[\, v_1^1,\, K[v_3^3, p]\,]$.
No a vymeniť poradie parametrov už vieme z podúlohy B: $sub \equiv K[v_2^2, v_1^2, bus]$.

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 2017