A-III-1 Suši bufet

Existuje veľa rôznych efektívnych riešení s podobnou alebo dokonca rovnakou asymptotickou časovou zložitosťou. Stručne popíšeme niekoľko rôznych so zložitosťou okolo $\Theta(n \log n)$ a potom sa pozrieme na jedno možné lineárne riešenie.

Mnohé efektívne riešenia tejto úlohy sú založené na tom, že vedia rýchlo robiť dve operácie: pre ľubovoľný súvislý úsek tanierikov povedať *súčet* jeho váh a *minimum* jeho kvalít.

Súčty úsekov sú ľahké: stačí si predpočítať prefixové súčty postupnosti $v_1, \ldots, v_n$ a z tých už vieme v konštantnom čase povedať súčet ľubovoľného úseku.

Minimá úsekov vieme rozumne efektívne (v logaritmickom čase) zisťovať napr. pomocou intervalového stromu.

Binárne vyhľadávanie

Pre každé suši i si označme m_i najmenšie také j, že ak Kika zje všetky suši od i-teho po j-te vrátane, zje aspoň z gramov suši. (Ak sa od i-teho suši ďalej už nedá zjesť dostatočne veľa gramov suši, budeme mať $m_i = \infty$.)

Ak by sme chceli len nájsť obed dostatočnej veľkosti a maximálnej kvality, určite by stačilo uvažovať úseky od i-teho suši po m_i-te (pre tie i, pre ktoré ešte existuje konečné m_i) – teda pre každý možný začiatok ten obed, pri ktorom prestaneme jesť akonáhle dosiahneme z zjedených gramov. Totiž tým, že budeme jesť viac, kvalitu obeda môžeme len znížiť, nikdy nie zvýšiť.

Pre konkrétne i vieme hodnotu m_i nájsť binárnym vyhľadávaním v čase $O(\log n)$, pričom používame predpočítané prefixové súčty postupnosti v na to, aby sme o každom úseku suši vedeli v konštantnom čase povedať, aký má súčet váh.

Akonáhle poznáme všetky hodnoty m_i, vieme pre každý z týchto $O(n)$ úsekov v čase $O(\log n)$ zistiť jeho minimum. Najväčšia z týchto hodnôt je zjavne hodnotou q_{max}, ktorú hľadáme.

No a akonáhle poznáme q_{max}, hodnotu v_{max} vieme ľahko zistiť v lineárnom čase. Stačí si uvedomiť, že všetky suši s kvalitou menšou ako q_{max} sú zakázané, čím sa nám vstupná postupnosť rozpadne na niekoľko súvislých úsekov tvorených dostatočne kvalitnými suši. Každému úseku spočítame celkovú váhu. Hodnota v_{max} je zjavne rovná najväčšej z týchto hodnôt.

Dva pointre

Hodnoty m_i vieme určiť aj v lineárnom čase pomocou techniky dvoch pointrov. Keď vieme pre konkrétne i jeho hodnotu m_i a následne posunieme začiatok úseku doprava (z i na $i+1$), koniec úseku sa určite neposunie doľava: vždy bude platiť $m_{i+1} \geq m_i$. Hodnotu m_{i+1} vieme teda vypočítať tak, že ju inicializujeme na m_i a následne ju o 1 zvyšujeme, kým treba.

Keďže aj začiatok aj koniec úseku posúvame len doprava, urobí toto riešenie dokopy vždy menej ako $2n$ posunov, a teda beží v lineárnom čase.

Naďalej však potrebujeme vedieť určiť aj minimum kvality na každom z týchto úsekov. Namiesto minimového intervalového stromu to však vieme robiť priebežne: k aktuálnemu úseku si budeme pamätať usporiadanú množinu jeho prvkov (napr. multiset v C++). Keď posúvame začiatok a koniec úseku, zároveň vyberáme z množiny prvky, ktoré v úseku prestali byť, a vkladáme prvky, ktoré do neho pribudli. A vždy, keď určíme novú hodnotu m_i, sa pozrieme na aktuálne minimum množiny, teda kvalitu práve skúmaného obeda.

S množinou dokopy spravíme $O(n)$ operácií, každú v čase $O(\log n)$.

Veľké kladivo

Range minimum query (RMQ) je všeobecná verzia problému, ktorý sa práve snažíme riešiť: dané je pole dĺžky n, môžeme si ho predspracovať a následne budeme potrebovať odpovedať na otázky tvaru „aké je minimum z hodnôt na indexoch od i po j?“.

Poznáme riešenia tohto problému, ktoré predspracujú vstup v čase $O(n)$ a následne vedia každú otázku zodpovedať v čase $O(1)$. Kombináciou techniky dvoch pointrov a tohto kladiva by sme teda vedeli celú súťažnú úlohu vyriešiť v lineárnom čase. Všeobecné optimálne algoritmy na RMQ sú však vcelku komplikované. Odpustíme si ich vysvetľovanie a namiesto toho si popíšeme jednoduchšie lineárne riešenie našej úlohy.

Vzorové riešenie

Pre každé i si označme ℓ_i a r_i index najbližšieho suši naľavo a napravo od i ktoré má kvalitu ostro menšiu ako q_i. Ak také suši neexistuje, tak máme $\ell_i = 0$, resp. $r_i = n + 1$.
Rozmyslite si, že keď poznáme všetky ℓ_i a r_i, vieme už celú úlohu doriešiť v lineárnom čase. Totiž niektoré suši muselo byť najhorším v Kikinom obede. My sa teraz pre každé suši opýtame otázku: ak si ty bolo to najhoršie suši, koľko najviac toho mohla Kika zjesť? No a odpoveď je zjavná: najväčší obed, v ktorom je toto suši najhoršie zo všetkých, tvoria práve všetky suši na indexoch od $\ell_i + 1$ po $r_i - 1$ vrátane. Z indexov vieme v konštantnom čase zistiť, koľko toho Kika zjedla. Minimum zisťovať ani nemusíme: priamo z otázky vieme, že ním je q_i.
Spomedzi týchto n maximálnych obedov uvažujeme len tie, pri ktorých Kika zjedla aspoň z gramov. Spomedzi ich kvality vyberieme tú najväčšiu, a následne spomedzi maximálnych obedov tej kvality vyberieme ten, pri ktorom zjedla najviac.
Hodnoty ℓ_i vieme vypočítať jedným prechodom poľom zľava doprava, pričom použijeme dátovú štruktúru *stack* (zásobník). V tom si budeme pamätať všetky indexy, ktoré ešte niekedy môžu byť ℓ_i. Na začiatku doň vložíme index 0 (pričom sa tvárime, že na tomto indexe máme suši kvality $-\infty$).
Každý ďalší index i teraz spracujeme nasledovne: Na vrchu stacku máme nejaké suši $j < i$. Sú dve možnosti.
Prvá je, že $q_j \geq q_i$. V takomto prípade platí, že suši j už odteraz nebude relevantné: je aspoň tak kvalitné ako q_i a zároveň platí, že ak sa v budúcnosti zjaví suši k vyššej kvality, určite nebude $\ell_k = j$, keďže medzi nimi je suši i, ktoré je aspoň tak zlé ako suši j. Ak teda nastane táto možnosť, suši j jednoducho zo stacku vyhodíme (a pokračujeme v spracúvaní suši i).
Druhá možnosť je, že $q_j < q_i$. V takomto prípade si zaznačíme, že $\ell_i = j$, a následne vložíme suši i na vrch stacku a skončíme jeho spracovanie.
Keďže každé suši raz vložíme na stack a nanajvýš raz ho zo stacku vyhodíme, má tento algoritmus celkovú časovú zložitosť $O(n)$. Pre dôkaz správnosti môžeme matematickou indukciou dokázať, že po spracovaní každého suši platí, že na stacku máme práve tie suši, ktoré (v doteraz spracovanom úseku) majú napravo od seba len suši ostro väčšej kvality.
Vďaka symetrii vieme hodnoty r_i vypočítať tak, že použijeme tento istý algoritmus na reverz pôvodného poľa. Tým sme dostali riešenie s lineárnou časovou zložitosťou.

Listing programu (C++)

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

vector<int> najdi_mensie_vlavo(const vector<int> &Q) {
    int N = Q.size();
    vector<int> odpoved(N), index(1,-1), hodnota(1,-1);
    for (int n=0; n<N; ++n) {
        while (hodnota.back() >= Q[n]) { index.pop_back(); hodnota.pop_back(); }
        odpoved[n] = index.back();
        index.push_back(n); hodnota.push_back(Q[n]);
    }
    return odpoved;
}

int main() {
    int N, Z;
    cin >> N >> Z;
    vector<int> Q(N);
    for (int n=0; n<N; ++n) cin >> Q[n];
    vector<int> V(N);
    for (int n=0; n<N; ++n) cin >> V[n];

    // spocitame prefixove sucty vah
    vector<int> SV(N+1,0);
    for (int n=0; n<N; ++n) SV[n+1] = SV[n] + V[n];

    // pre kazde susi najdeme najblizsie horsie vlavo...
    vector<int> L = najdi_mensie_vlavo(Q);
    // ... a vpravo
    reverse( Q.begin(), Q.end() );
    vector<int> R = najdi_mensie_vlavo(Q);
    reverse( Q.begin(), Q.end() );
    reverse( R.begin(), R.end() );
    for (int &r : R) r = N-1-r;

    // pre kazde susi zistime, kolko najviac zjeme, ak ono je najhorsie
    int Qmax = -1, Vmax = 0;
    for (int n=0; n<N; ++n) {
```

```
        int lo = L[n] + 1, hi = R[n]; // zjem polo-otvoreny interval [lo,hi)
        int Vcur = SV[hi] - SV[lo];   // toto je vaha toho co zjem
        if (Vcur < Z) continue;       // neviem zjest dost
        if (Q[n] > Qmax) { Qmax = Q[n]; Vmax = Vcur; }
        if (Q[n] == Qmax) Vmax = max( Vmax, Vcur );
    }
    cout << Qmax << endl << Vmax << endl;
}
```

A-III-2 Pexeso

V kvadratickom čase vieme úlohu riešiť ľahko. Na začiatku spravíme *kompresiu súradníc*: obrázkom priradíme nové čísla z rozsahu od 0 po $r-1$, kde $r \leq n$ je počet rôznych obrázkov.
Potom postupne pre každý začiatok začneme s prázdnym úsekom a postupne posúvame koniec doprava. Vždy, keď do úseku pribudne nový obrázok, zväčšíme si počet jeho výskytov (vďaka kompresii súradníc na to môžeme použiť obyčajné pole) a príslušne si prepočítame počet obrázkov, ktoré majú práve dva výskyty.

Prvé vzorové riešenie: šikovný intervalový strom

Predstavme si, že prechádzame vstupné pole zľava doprava. Pre každú hodnotu v poli platí, že keď stretneme jej druhý výskyt, narastie počet dvojíc v úseku, ktorý sme už prešli, a keď stretneme jej tretí výskyt, počet dvojíc zase klesne.
Zaznačme si tieto informácie do poľa. Zoberme pole B inicializované na nuly. Druhému výskytu každej hodnoty (ak existuje) nastavme $B[i] = +1$ a tretiemu (opäť, ak existuje) nastavme $B[i] = -1$. Teraz zjavne pre každé x platí, že počet práve-dvojíc na prvých x pozíciách poľa A je rovný súčtu prvých x hodnôt poľa B.
Optimálne riešenie *začínajúce na začiatku celého poľa* teda zodpovedá najväčšiemu z prefixových súčtov poľa B. Pozrime sa teraz, čo sa stane, keď zahodíme prvok $A[0] = x$. V poli B sa zmení len konštantne veľa pozícií: nový druhý a tretí výskyt hodnoty x sú teraz inde.
Tým, že budeme postupne zahadzovať prvky zo začiatku poľa A a prepočítavať pole B, postupne vyskúšame všetky možné začiatky úseku.
Ostáva už len jediný krok: pre každý začiatok potrebujeme vedieť efektívne zistiť, akú hodnotu má momentálne najväčší z prefixových súčtov poľa B. Aby sme toto vedeli robiť efektívne, postavíme si nad poľom B intervalový strom. V každom vrchole si budeme pamätať dva údaje: súčet celého úseku, ktorý mu zodpovedá, a najväčší z prefixových súčtov pre jeho úsek.
Toto riešenie má časovú zložitosť $O(n \log n)$ a pamäťovú $O(n)$.

Listing programu (C++)

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

struct zaznam {
    int cely_sucet, max_prefixovy_sucet;
    zaznam() { cely_sucet = 0; max_prefixovy_sucet = 0; }
};

struct intervalac {
    vector< vector<zaznam> > data;
    intervalac(int N);
    int max_prefixovy_sucet();
    void nastav(int kde, int co);
};

intervalac::intervalac(int N) {
    for (int d=0; ; ++d) {
        data.push_back( vector<zaznam>(1<<d) );
        if (int(data.back().size()) >= N) break;
    }
}

int intervalac::max_prefixovy_sucet() {
    return data[0][0].max_prefixovy_sucet;
}

void intervalac::nastav(int kde, int co) {
    int x = data.size() - 1;
```

```
    data[x][kde].cely_sucet = co;
    data[x][kde].max_prefixovy_sucet = max(0, co);
    while (true) {
        --x;
        if (x < 0) break;
        kde /= 2;
        data[x][kde].cely_sucet =
            data[x+1][2*kde].cely_sucet +
            data[x+1][2*kde+1].cely_sucet;
        data[x][kde].max_prefixovy_sucet = max(
            data[x+1][2*kde].max_prefixovy_sucet,
            data[x+1][2*kde].cely_sucet + data[x+1][2*kde+1].max_prefixovy_sucet
        );
    }
}

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

    map<int, deque<int> > vyskyty;
    for (int n=0; n<N; ++n) vyskyty[ A[n] ].push_back(n);

    intervalac T(N);
    for (auto rec : vyskyty) {
        int a = rec.first;
        if (vyskyty[a].size() >= 2u) T.nastav( vyskyty[a][1], +1 );
        if (vyskyty[a].size() >= 3u) T.nastav( vyskyty[a][2], -1 );
    }

    int odpoved = T.max_prefixovy_sucet();

    for (int n=0; n<N; ++n) {
        int a = A[n];
        vyskyty[a].pop_front();
        if (vyskyty[a].size() >= 1u) T.nastav( vyskyty[a][0],  0 );
        if (vyskyty[a].size() >= 2u) T.nastav( vyskyty[a][1], +1 );
        if (vyskyty[a].size() >= 3u) T.nastav( vyskyty[a][2], -1 );
        odpoved = max( odpoved, T.max_prefixovy_sucet() );
    }

    cout << odpoved << endl;
}
```

Druhé vzorové riešenie: zametanie obdĺžnikov

Pre každý obrázok x, ktorý sa v poli nachádza, sa postupne pozrime na každé dva po sebe idúce jeho výskyty. Ak je tento obrázok súčasťou pexesa, bude v ňom zjavne niektorá z týchto dvojíc. Pre každú konkrétnu dvojicu po sebe idúcich výskytov x sa teraz pozrime na to, ktoré kúsky obsahujú práve ju a žiadne ďalšie x.
Najkratší možný kúsok samozrejme začína prvým a končí druhým z týchto výskytov. Začiatok môžeme posunúť doľava až kým neprídeme buď na začiatok celého poľa, alebo tesne pred skorší výskyt x. Nezávisle od toho môžeme koniec posunúť doprava až po koniec poľa alebo najbližší ďalší výskyt x. Pre každú konkrétnu dvojicu si takto zostrojíme a zapíšeme dva uzavreté intervaly: jeden pre index začiatku a jeden pre index konca jej zodpovedajúceho kúsku.

Predstavme si teraz dvojrozmerné pole, ktorého riadky zodpovedajú všetkým možným začiatkom a stĺpce všetkým možným koncom kúsku vstupu. Každé políčko (r, s) nad hlavnou uhlopriečkou teda zodpovedá jednému možnému kúsku vstupu.
Vyššie sme si zdôvodnili, že pre každú dvojicu po sebe idúcich výskytov toho istého obrázku existuje nejaký interval $[r_1, r_2]$ a nejaký interval $[s_1, s_2]$, že práve kúsky, ktoré začínajú v prvom a končia v druhom intervale budú obsahovať túto dvojicu vo svojom pexese. Keď si v našom poli vyfarbíme všetky políčka, ktoré zodpovedajú týmto kúskom, zjavne dostaneme obdĺžnik.
Pre každú po sebe idúcu dvojicu takto dostávame jeden obdĺžnik. Týchto obdĺžnikov je dokopy menej ako n, keďže každý konkrétny výskyt obrázku je druhým výskytom v nanajvýš jednej dvojici.
Všimnime si teraz ešte, že keď sa pozeráme na rôzne po sebe idúce dvojice výskytov toho istého obrázku, dostávame pre ne vždy obdĺžniky, ktoré sú navzájom disjunktné. (Ak by sa nejaké dva na nejakom políčku prekrývali, znamenalo by to, že tomu políčku zodpovedajúci kúsok obsahuje obe tie dvojice, to je ale v spore s tým, že naše kúsky obsahujú každý len dva výskyty nášho konkrétneho obrázku.)

Máme teda $x < n$ obdĺžnikov. Pre každý kúsok pásika vieme určiť jeho veľkosť pexesa tak, že spočítame, v koľkých obdĺžnikoch leží. V našej tabuľke teda chceme nájsť políčko, ktoré leží v najviac obdĺžnikoch.
Túto otázku vieme zodpovedať v lepšom ako kvadratickom čase vhodným použitím zametania.
Predstavme si, že nad našou tabuľkou nakreslíme vodorovnú čiaru. Teraz budeme túto čiaru postupne posúvať dodola. Pre každý obdĺžnik niekedy nastane moment, kedy čiara trafí jeho hornú stranu. Od tohto okamihu naša čiara pretína obdĺžnik. Ich prienikom je stále vodorovná úsečka zodpovedajúca stĺpcom, v ktorých tento obdĺžnik leží. Toto platí až do druhého okamihu, kedy naša čiara príde na spodok obdĺžnika.
Počas celého práve popísaného procesu teda nastane dokopy presne $2x$ udalostí: každý obdĺžnik raz začne a raz prestane pretínať našu zametaciu priamku. Tieto udalosti si môžeme všetky vygenerovať, usporiadať zhora dole a v tomto poradí ich spracovať. Keď spracujeme začiatok obdĺžnika, pribudne nám nová úsečka, a keď spracujeme jeho koniec, táto úsečka zase prestane existovať.
Náš dvojrozmerný problém sme tým zredukovali na postupnosť jednorozmerných problémov. Medzi každými dvoma udalosťami (ktoré nastanú v rôznych časoch) máme nejakú sadu úsečiek, ktorá zodpovedá obdĺžnikom, ktoré obsahujú daný riadok tabuľky. Pre každú takúto sadu hľadáme políčko, ktoré leží na najväčšom počte z týchto úsečiek.
Na tieto otázky vieme efektívne odpovedať tak, že si úsečky budeme ukladať do vhodnej dátovej štruktúry. Jedna možnosť je postaviť si „lenivý“ intervalový strom nad stĺpcami našej tabuľky. Do tohto stromu budeme potom vkladať naše úsečky, keď pribudnú, a zase ich odoberať, keď prestanú existovať. V každom podstrome si pri tom budeme priebežne udržiavať informáciu o tom, aký najväčší počet úsečiek v ňom má spoločný prienik. Takto vieme dosiahnuť rovnakú časovú zložitosť $O(n \log n)$ ako v predchádzajúcom riešení.

A-III-3 O Vekslákbotovi a Pokladničke

Podúloha A (2 body): majorita

Kľúčové je uvedomiť si, že ak z Pokladničky odstránime ľubovoľné dva žetóny **rôznych farieb**, majorita vždy zostane zachovaná.
(Nech má napr. červená farba majoritu. Predstavme si väčšiu kôpku červených žetónov a menšiu kôpku zelených a modrých dokopy. Ak odoberieme červený a iný žetón, odobrali sme jeden z každej kôpky, a teda červená kôpka ostala väčšia. A ak odoberieme zelený a modrý, je to ešte zjavnejšie.)
Každý program tvaru „kým existujú dva žetóny rôznych farieb, nejakú takú dvojicu odober“ teda eventuálne skončí s tým, že už existujú len žetóny jednej farby – tej, ktorá mala na začiatku majoritu.
Teraz už len potrebujeme zredukovať počet týchto žetónov na jeden. To je našťastie triviálne: stačí pridať druhú časť programu s inštrukciami tvaru „ak vidíš dva rovnaké žetóny, nahraď ich jedným tej istej farby“.

Listing programu

```
cervena, zelena ->
cervena, modra ->
zelena, modra ->

cervena, cervena -> cervena
zelena, zelena -> zelena
modra, modra -> modra
```

Podúloha B (4 body): logaritmus

Začneme ošetrením špeciálneho prípadu: $g(1) = 0$, preto ak je v Pokladničke len jeden červený žetón, rovno skončíme. Vo všetkých ostatných prípadoch začneme tým, že si vyrobíme jeden svetlomodrý žetón.
Hľadáme najmenšie m také, že $2^m \geq c$. Nájdeme ho tak, že budeme dookola opakovať nasledovné kroky:

- Z m svetlomodrých žetónov vyrobíme m modrých a 2^m fialových.
- Zistíme, či je fialových aspoň toľko ako červených.
- Ak áno, aktuálny počet modrých je hľadanou odpoveďou.

- Ak nie, vyrobíme $m + 1$ svetlomodrých a začneme odznova.

Listing programu

```
OBMEDZENIE: start <= 1
OBMEDZENIE: umocnuj1 + umocnuj2 + kontroluj + resetuj + koniec <= 1

2 cervena -> start, umocnuj1, 2 cervena, svetlomodra, fialova

umocnuj1, svetlomodra, fialova -> umocnuj1, svetlomodra, 2 tmavofialova
umocnuj1, svetlomodra -> umocnuj2, modra
umocnuj2, tmavofialova -> umocnuj2, fialova
umocnuj2, svetlomodra -> umocnuj1, svetlomodra
umocnuj2 -> kontroluj

kontroluj, cervena, fialova -> kontroluj, tmavocervena
kontroluj, cervena -> resetuj, cervena
kontroluj -> koniec

resetuj, tmavocervena -> resetuj, cervena
resetuj, modra -> resetuj, svetlomodra
resetuj -> umocnuj1, svetlomodra, fialova
```

Všimnite si, že vďaka žetónu `start` sa prvá inštrukcia vykoná len raz. Navyše, vďaka „2 červená“ na ľavej strane sa táto inštrukcia nevykoná pre $c = 1$. (Vtedy sa nevykoná vôbec nič.)
Štartová inštrukcia nám tiež vyrobí prvý svetlomodrý a prvý fialový žetón. V stavoch `umocnuj1` a `umocnuj2` postupne m-krát vynásobíme počet fialových dvoma, čím dosiahneme, že fialových je 2^m. Zároveň svetlomodré prefarbíme na modré.
Následne v stave `kontroluj` súčasne prefarbujeme červené a mažeme fialové. Ak sa fialové minú skôr, ešte ich bolo málo – vrátime teda červené do pôvodného stavu, súčasné modré prefarbíme naspäť na svetlomodré a pridáme im ešte jednu svetlomodrú navyše. Ak sa červené a fialové minú naraz alebo dokonca sa minú červené skôr, už nemusíme robiť vôbec nič: aktuálny počet modrých je správny.

Podúloha C (4 body): Fibonacci

Podobne ako v krajskom kole aj vyššie uvedenej podúlohe B, aj tu použijeme špeciálne farby žetónov na reprezentáciu stavu, v ktorom sa práve výpočet nachádza. Keďže ale nesmieme používať obmedzenia, budeme si musieť postrážiť, aby sa nám stavy nijak nepomiešali.
Na úvod si všimnime, že vieme rozoznať začiatočný stav (existujú červené žetóny a nič iné) a vtedy raz spraviť niečo špeciálne. Taktiež vieme aj bez použitia obmedzení skončiť v želanom koncovom stave (stačí nemať žiadnu inštrukciu, ktorá má na ľavej strane len modré žetóny).
Ako by vyzeral „normálny“ výpočet n-tého Fibonacciho čísla? Napr. nasledovne:

```
p := F[0]
q := F[1]
zopakuj n-krát:
    r := p+q
    p := q
    q := r
v premennej p je hodnota F[n]
```

Na reprezentáciu samotných Fibonacciho čísel počas výpočtu budeme používať púpavové, kvietočkové a ružové žetóny (skrátene teda „premenné“ p, q, r). Výpočet nášho programu pre Pokladničku bude prebiehať v kolách a po každom celom kole budú počty týchto žetónov zodpovedať vyššie uvedenému programu.
Keďže $F_0 = 0$ a $F_1 = 1$, na začiatku potrebujeme mať nula žetónov typu p a jeden žetón typu q. Ten si vyrobíme nasledovne: všetky červené prefarbíme na čierne, pričom ako prvú budeme mať inštrukciu „červená, kvietočková → čierna, kvietočková“ a ako druhú inštrukciu „červená → čierna, kvietočková“. V úplne prvom kroku výpočtu sa teda použije druhá inštrukcia, vyrobí nám jedno q, a odvtedy ďalej sa bude používať prvá, ktorá ho len zachová.
Každé kolo výpočtu nášho programu následne prebehne nasledovne:

- Na začiatku cyklu máme $p = F_i$, $q = F_{i+1}$ a $r = 0$.
- Odstráň jeden čierny žetón. (Ak to nevieš spraviť, už sme na konci výpočtu.)

- Fáza 1: Za každú púpavovú pridaj jednu ružovú.
- Keď sa púpavové minú, máme $p = 0$ a $r = F_i$. V zatiaľ nedotknutom q je stále F_{i+1}.
- Fáza 2: Za každú kvietočkovú pridaj aj jednu ružovú, aj jednu púpavovú.
- Keď sa kvietočkové minú, máme $p = F_{i+1}$, $q = 0$ a $r = F_i + F_{i+1} = F_{i+2}$.
- Fáza 3: Za každú ružovú pridaj jednu kvietočkovú.
- Keď sa ružové minú, máme $p = F_{i+1}$, $q = F_{i+2}$ a $r = 0$, čím sme úspešne odsimulovali jednu iteráciu vyššie uvedeného pseudokódu.

Pozrime si teraz výsledný listing programu:

Listing programu

```
cervena, qietockova -> cierna, qietockova
cervena -> cierna, qietockova

faza1, pupavova -> faza1, ruzova
faza1 -> faza2
faza2, qietockova -> faza2, pupavova, ruzova
faza2 -> faza3
faza3, ruzova -> faza3, qietockova
faza3 ->

cierna -> faza1

pupavova -> modra
qietockova ->
```

Prvé dve inštrukcie sa vykonajú len na začiatku a odvtedy na ne môžeme zabudnúť. Počet čiernych žetónov nám teraz hovorí, koľko iterácií výpočtu nasledovného Fibonacciho čísla chceme robiť. Každá iterácia začína tým, že odstránime jeden čierny žetón a vyrobíme si žetón faza1. Pomocné žetóny `faza1`, `faza2`, `faza3` nám pomôžu zabezpečiť, aby výmeny farieb prebehli v tom poradí, ktoré chceme.
No a keď sa už aj čierne žetóny minú, vieme, že máme presne F_n púpavových (a tiež presne F_{n+1} kvietočkových). Púpavové žetóny zmeníme na modré, kvietočkové zahodíme a sme hotoví.

TRIDSIATY DEVIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek, Paulína Smolárová, Truc Lam Bui
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2024