P-III-1 Suši bufet

Existuje mnoho různých efektivních řešení s podobnou nebo dokonce stejnou asymptotickou časovou složitostí. Stručně popíšeme několik různých se složitostí kolem $\Theta(n \log n)$ a pak se podíváme na jedno možné lineární řešení.

Mnohá efektivní řešení tohoto úkolu jsou založena na tom, že umí rychle dělat dvě operace: pro libovolný souvislý úsek talířků říci *součet* jeho vah a *minimum* jeho kvalit.

Součty úseků jsou lehké: stačí si předpočítat prefixové součty posloupnosti $v_1, \ldots, v_n$ a z těch už umíme v konstantním čase říci součet libovolného úseku. Minima úseků umíme rozumně efektivně (v logaritmickém čase) zjišťovat např. pomocí intervalového stromu.

Binární vyhledávání

Pro každé suši i si označme jako m_i nejmenší index $j \geq i$ takový, že pokud Kika sní všechny suši od i-tého po j-té včetně, sní alespoň z gramů suši. Pokud od i-tého suši dál už nelze sníst dostatečně mnoho gramů suši, definujme $m_i = \infty$.

Pokud bychom chtěli jen najít oběd dostatečné velikosti a maximální kvality, určitě by stačilo uvažovat úseky od i-tého suši po m_i-té (pro ty i, pro které ještě existuje konečné m_i) – tedy pro každý možný začátek ten oběd, při kterém přestaneme jíst jakmile dosáhneme z snědených gramů. Tím, že bychom jedli více, bychom kvalitu oběda mohli jen snížit, nikdy ne zvýšit.

Pro konkrétní i umíme hodnotu m_i najít pomocí binárního vyhledávání v čase $\mathcal{O}(\log n)$, přičemž používáme předpočtené prefixové součty posloupnosti v k tomu, abychom o každém úseku suši uměli v konstantním čase říci, jaký má součet vah.

Jakmile známe všechny hodnoty m_i, umíme pro každý z $\mathcal{O}(n)$ úseků od i do m_i v čase $\mathcal{O}(\log n)$ zjistit jeho minimum. Největší z těchto hodnot je zjevně hodnotou q_{max}, kterou hledáme.

No a jakmile známe q_{max}, hodnotu v_{max} umíme snadno zjistit v lineárním čase. Stačí si uvědomit, že všechny suši s kvalitou menší než q_{max} jsou zakázány, čímž se nám vstupní posloupnost rozpadne na několik souvislých úseků tvořených dostatečně kvalitními suši. Každému úseku spočítáme celkovou váhu. Hodnota v_{max} je zjevně rovna největší z těchto hodnot.

Dva ukazatele

Hodnoty m_i umíme určit i v lineárním čase pomocí techniky dvou ukazatelů. Když známe pro konkrétní i jeho hodnotu m_i a následně posuneme začátek úseku doprava (z i na $i + 1$), konec úseku se určitě neposune doleva: vždy bude platit $m_{i+1} \geq m_i$. Hodnotu m_{i+1} umíme tedy vypočítat tak, že ji inicializujeme na m_i a následně ji o 1 zvyšujeme, dokud je potřeba.

Jelikož ukazatele na začátek i konec úseku posouváme jen doprava, udělá toto řešení dohromady vždy méně než $2n$ posunů, a tedy běží v lineárním čase.

Nadále však potřebujeme umět určit i minimum kvality na každém z těchto úseků. Namísto minimového intervalového stromu to umíme dělat i průběžně: k aktuálnímu úseku si budeme pamatovat uspořádanou množinu jeho prvků (např. multiset v C++). Když posouváme začátek a konec úseku, zároveň vybíráme z množiny prvky, které v úseku přestaly být, a vkládáme prvky, které do něj přibyly. Vždy, když určíme novou hodnotu m_i, se podíváme na aktuální minimum množiny, a ta nám udává kvalitu právě zkoumaného oběda. S množinou dohromady uděláme $\mathcal{O}(n)$ operací, každou v čase $\mathcal{O}(\log n)$.

Kanón na vrabce

Range minimum query (RMQ) je obecná verze problému, který se právě snažíme řešit: Je dáno pole délky n, můžeme si jej předzpracovat a následně budeme potřebovat odpovídat na otázky tvaru „jaké je minimum z hodnot na indexech od i do j?“.

Existují řešení tohoto problému, která předzpracují vstup v čase $\mathcal{O}(n)$ a následně umí každou otázku zodpovědět v čase $\mathcal{O}(1)$. Kombinací techniky dvou ukazatelů a této datové struktury bychom tedy uměli celý soutěžní úkol vyřešit v lineárním čase. Obecné optimální algoritmy na RMQ jsou však komplikované. Odpustíme si jejich vysvětlování a místo toho si popíšeme jednodušší lineární řešení našeho úkolu.

Vzorové řešení

Pro každé i si označme ℓ_i a r_i index nejbližšího suši nalevo a napravo od i, které má kvalitu ostře menší než q_i. Pokud takové suši neexistuje, tak máme $\ell_i = 0$, resp. $r_i = n + 1$.

Rozmyslete si, že když známe všechna ℓ_i a r_i, umíme už celý úkol dořešit v lineárním čase. Některé suši muselo být nejhorší v Kičině obědě. My se teď pro každé suši zeptáme otázku: pokud jsi ty bylo to nejhorší suši, kolik nejvíc toho mohla Kika sníst? No a odpověď je zjevná: největší oběd, ve kterém je toto suši nejhorší ze všech, tvoří právě všechny suši na indexech od $\ell_i + 1$ po $r_i - 1$ včetně. Z indexů umíme v konstantním čase zjistit, kolik toho Kika snědla. Minimum zjišťovat ani nemusíme: přímo z otázky víme, že jím je q_i.

Z těchto n maximálních obědů uvažujeme jen ty, u kterých Kika snědla alespoň z gramů. Z jejich kvality vybereme tu největší, a následně z maximálních obědů té kvality vybereme ten, při kterém snědla nejvíc.

Hodnoty ℓ_i umíme vypočítat jedním přechodem polem zleva doprava, přičemž použijeme datovou strukturu *zásobník*. V tom si budeme pamatovat všechny indexy, které ještě někdy mohou být ℓ_i. Na začátku do něj vložíme index 0 (přičemž se tváříme, že na tomto indexu máme suši kvality $-\infty$). Každý další index i nyní zpracujeme následovně: Na vrchu zásobníku máme nějaký index $j < i$. Jsou dvě možnosti.

První je, že $q_j \geq q_i$. V takovém případě platí, že suši j již nyní nebude relevantní: je alespoň tak kvalitní jako q_i a zároveň platí, že pokud se v budoucnosti

zjeví suši k vyšší kvality, určitě nebude $\ell_k = j$, protože mezi nimi je suši i, které je alespoň tak špatné jako suši j. Pokud tedy nastane tato možnost, suši j jednoduše ze zásobníku vyhodíme (a pokračujeme ve zpracování suši i).

Druhá možnost je, že $q_j < q_i$. V takovém případě si zaznamenáme, že $\ell_i = j$, a následně vložíme suši i na vrch zásobníku a skončíme jeho zpracování.

Jelikož každé suši vložíme na zásobník právě jednou a nanejvýš jednou ho ze zásobníku vyhodíme, má tento algoritmus celkovou časovou složitost $\mathcal{O}(n)$. Pro důkaz správnosti můžeme matematickou indukcí dokázat, že po zpracování každého suši platí, že na zásobníku máme právě ty suši, které (v dosud zpracovaném úseku) mají napravo od sebe jen suši ostře větší kvality.

Díky symetrii umíme hodnoty r_i vypočítat tak, že použijeme tentýž algoritmus na obrácené pole. Tím jsme obdrželi řešení s lineární časovou složitostí.

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

vector<int> najdi_mensi_vlevo(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];

    // spočítáme prefixové součty vah
    vector<int> SV(N+1,0);
    for (int n=0; n<N; ++n) SV[n+1] = SV[n] + V[n];

    // pro každé suši najdeme najbližší horší vlevo...
    vector<int> L = najdi_mensi_vlevo(Q);
    // ... a vpravo
    reverse( Q.begin(), Q.end() );
    vector<int> R = najdi_mensi_vlevo(Q);
    reverse( Q.begin(), Q.end() );
    reverse( R.begin(), R.end() );
    for (int &r : R) r = N-1-r;

    // pro každé suši zjistíme, klik nejvíc sníme, jestliže je nejhorší snězené
    int Qmax = -1, Vmax = 0;
    for (int n=0; n<N; ++n) {
        int lo = L[n] + 1, hi = R[n]; // sním polo-otevřený interval [lo,hi)
        int Vcur = SV[hi] - SV[lo];   // toto je váha toho, co sním
        if (Vcur < Z) continue;       // nesnědl jsem dost
        if (Q[n] > Qmax) { Qmax = Q[n]; Vmax = Vcur; }
        if (Q[n] == Qmax) Vmax = max( Vmax, Vcur );
    }
    cout << Qmax << endl << Vmax << endl;
}
```

P-III-2 Pexeso

V kvadratickém čase umíme úlohu vyřešit snadno. Na začátku uděláme *kompresi souřadnic*: obrázkům přiřadíme nová čísla z rozsahu od 0 po $r-1$, kde $r \leq n$ je počet různých obrázků.

Potom postupně pro každý začátek začneme s prázdným úsekem a postupně posouváme konec doprava. Vždy, když do úseku přibude nový obrázek, zvětšíme si počet jeho výskytů (díky kompresi souřadnic na to můžeme použít obyčejné pole) a příslušně si přepočítáme počet obrázků, které mají právě dva výskyty.

První vzorové řešení: chytrý intervalový strom

Představme si, že procházíme vstupní pole zleva doprava. Pro každou hodnotu v poli platí, že když potkáme její druhý výskyt, naroste počet dvojic v úseku, který jsme již prošli, a když potkáme její třetí výskyt, počet dvojic zase klesne.

Poznačme si tyto informace do pole. Vezměme pole B inicializované na nuly. Druhému výskytu každé hodnoty (pokud existuje) nastavme $B[i] = +1$ a třetímu (opět, pokud existuje) nastavme $B[i] = -1$. Nyní zjevně pro každé x platí, že počet právě-dvojic na prvních x pozicích pole A je roven součtu prvních x hodnot pole B. Optimální řešení *začínající na začátku celého pole* tedy odpovídá největšímu z prefixových součtů pole B.

Podívejme se nyní, co se stane, když zahodíme prvek $A[0] = x$. V poli B se změní jen konstantně mnoho pozic: nový druhý a třetí výskyt hodnoty x jsou nyní jinde. Tím, že budeme postupně zahazovat prvky ze začátku pole A a přepočítávat pole B, postupně vyzkoušíme všechny možné začátky úseku.

Zbývá už jen jediný krok: pro každý začátek potřebujeme umět efektivně zjistit, jakou hodnotu má momentálně největší z prefixových součtů pole B. Abychom toto uměli dělat, postavíme si nad polem B intervalový strom. V každém vrcholu si budeme pamatovat dva údaje: součet celého úseku, který mu odpovídá, a největší z prefixových součtů pro jeho úsek.

Toto řešení má časovou složitost $\mathcal{O}(n \log n)$ a paměťovou $\mathcal{O}(n)$.

Druhé vzorové řešení: zametání obdélníků

Pro každý obrázek x, který se v poli nachází, se postupně podívejme na každé dva po sobě jdoucí výskyty. Pokud je tento obrázek součástí pexesa, bude v něm zjevně některá z těchto dvojic. Pro každou konkrétní dvojici po sobě jdoucích výskytů x se nyní podívejme na to, které kousky obsahují právě ji a žádné další x.

Nejkratší možný kousek samozřejmě začíná prvním a končí druhým z těchto výskytů. Začátek můžeme posunout doleva dokud nepřijedeme buď na začátek celého pole, nebo těsně před dřívější výskyt x. Nezávisle na tom můžeme konec posunout doprava až po konec pole nebo nejbližší další výskyt x. Pro každou konkrétní dvojici si takto sestrojíme a zapíšeme dva uzavřené intervaly: jeden pro index začátku a jeden pro index konce jejího odpovídajícího kousku.

Představme si nyní dvourozměrné pole, jehož řádky odpovídají všem možným začátkům a sloupce všem možným koncům kousku vstupu. Každé políčko (r, s) nad hlavní úhlopříčkou tedy odpovídá jednomu možnému kousku vstupu.

Výše jsme si zdůvodnili, že pro každou dvojici po sobě jdoucích výskytů téhož obrázku existuje nějaký interval $\langle r_1, r_2 \rangle$ a nějaký interval $\langle s_1, s_2 \rangle$ takový, že právě kousky, které začínají v prvním a končí ve druhém intervalu, budou obsahovat tuto dvojici ve svém pexese. Když si v našem poli vybarvíme všechna políčka, která odpovídají těmto kouskům, zjevně dostaneme obdélník.

Pro každou po sobě jdoucí dvojici takto dostáváme jeden obdélník. Těchto obdélníků je dohromady méně než n, protože každý konkrétní výskyt obrázku je druhým výskytem v nanejvýš jedné dvojici.

Všimněme si teď ještě, že když se díváme na různé po sobě jdoucí dvojice výskytů stejného obrázku x, dostáváme pro ně vždy obdélníky, které jsou navzájem disjunktní: pokud by se tyto obdélníky na nějakém políčku překrývaly, znamenalo by to, že tomu políčku odpovídající kousek obsahuje obě dvojice, to je ale ve sporu s tím, že naše kousky obsahují každý jen dva výskyty obrázku x.

Máme tedy $t < n$ obdélníků. Pro každý kousek umíme určit jeho velikost pexesa tak, že spočítáme, v kolika obdélnících leží. V naší tabulce tedy chceme najít políčko, které leží v nejvíce obdélnících. Tuto otázku umíme zodpovědět v lepším než kvadratickém čase vhodným použitím zametání.

Představme si, že nad naší tabulkou nakreslíme vodorovnou čáru. Nyní budeme tuto čáru postupně posouvat dolů. Pro každý obdélník někdy nastane moment, kdy čára trefí jeho horní stranu. Od tohoto okamžiku naše čára protíná obdélník. Jejich průnikem je stále vodorovná úsečka odpovídající sloupcům, ve kterých tento obdélník leží. Toto platí až do druhého okamžiku, kdy naše čára přijde na spodek obdélníku.

Během celého právě popsaného procesu tedy nastane dohromady přesně $2t$ událostí: každý obdélník jednou začne a jednou přestane protínat naši zametací přímku. Tyto události si můžeme všechny vygenerovat, uspořádat shora dolů a v tomto pořadí je zpracovat. Když zpracujeme začátek obdélníku, přibude nám nová úsečka, a když zpracujeme jeho konec, tato úsečka zase přestane existovat.

Náš dvourozměrný problém jsme tím zredukovali na posloupnost jednorozměrných problémů. Mezi každými dvěma událostmi (které nastanou v různých časech) máme nějakou sadu úseček, která odpovídá obdélníkům, které obsahují daný řádek tabulky. Pro každou takovou sadu hledáme políčko, které leží v největším počtu z těchto úseček.

Na tuto otázky umíme efektivně odpovědět tak, že si úsečky budeme ukládat do vhodné datové struktury. Jedna možnost je si postavit „líný“ intervalový strom nad sloupci naší tabulky. Do tohoto stromu budeme potom vkládat naše úsečky, až přibudou, a zase je odebírat, když přestanou existovat. V každém podstromu si přitom budeme průběžně udržovat informaci o tom, jaký největší počet úseček v něm má společný průnik. Takto umíme dosáhnout stejné časové složitosti $\mathcal{O}(n \log n)$ jako v předchozím řešení.

Tento program implementuje první vzorové řešení:

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

struct zaznam {
    int cely_soucet, max_prefixovy_soucet;
    zaznam() { cely_soucet = 0; max_prefixovy_soucet = 0; }
};

struct intervalac {
    vector< vector<zaznam> > data;
    intervalac(int N);
    int max_prefixovy_soucet();
    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_soucet() {
    return data[0][0].max_prefixovy_soucet;
}

void intervalac::nastav(int kde, int co) {
    int x = data.size() - 1;
    data[x][kde].cely_soucet = co;
    data[x][kde].max_prefixovy_soucet = max(0, co);
    while (true) {
        --x;
        if (x < 0) break;
        kde /= 2;
        data[x][kde].cely_soucet =
            data[x+1][2*kde].cely_soucet +
            data[x+1][2*kde+1].cely_soucet;
        data[x][kde].max_prefixovy_soucet = max(
            data[x+1][2*kde].max_prefixovy_soucet,
            data[x+1][2*kde].cely_soucet + data[x+1][2*kde+1].max_prefixovy_soucet
        );
    }
}

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_soucet();

    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_soucet() );
    }

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

P-III-3 O Vekslákbotovi a Pokladničce

Podúloha A (2 body): nadpoloviční většina

Klíčové je uvědomit si, že pokud z Pokladničky odstraníme libovolné dva žetony *různých barev*, nadpoloviční většina vždy zůstane zachována. Ať má např. červená barva nadpoloviční většinu. Představme si větší hromádku červených žetonů a menší hromádku zelených a modrých dohromady. Pokud odebereme červený a jiný žeton, odebrali jsme jeden z každé hromádky, a tedy červená hromádka zůstala větší. A pokud odebereme zelený a modrý, je to ještě zjevnější.

Každý program tvaru „dokud existují dva žetony různých barev, nějakou takovou dvojici odeber“ tedy eventuálně skončí s tím, že už existují jen žetony jedné barvy – té, která měla na začátku nadpoloviční většinu.

Teď už jen potřebujeme zredukovat počet těchto žetonů na jeden. To je naštěstí triviální: stačí přidat druhou část programu s instrukcemi tvaru „pokud vidíš dva stejné žetony, nahraď je jedním téže barvy“.

1. červená, zelená $\to \emptyset$
2. červená, modrá $\to \emptyset$
3. zelená, modrá $\to \emptyset$
4. 2červená $\to$ červená
5. 2zelená $\to$ zelená
6. 2modrá $\to$ modrá

Podúloha B (4 body): logaritmus

Začneme ošetřením speciálního případu: $g(1) = 0$, proto je-li v Pokladničce jen jeden červený žeton, rovnou skončíme. Ve všech ostatních případech začneme tím, že si vyrobíme jeden světle modrý žeton.

Hledáme nejmenší m takové, že $2^m \geq c$. Najdeme ho tak, že budeme kolem dokola opakovat následující kroky:

- Z m světle modrých žetonů vyrobíme m modrých a 2^m fialových.

- Zjistíme, jestli je fialových alespoň tolik jako červených.
- Pokud ano, aktuální počet modrých je hledanou odpovědí.
- Pokud ne, vyrobíme $m + 1$ světle modrých a začneme znovu.

Program pak vypadá následovně. Nejprve omezení:

- start ≤ 1
- umocňuj1 + umocňuj2 + kontroluj + resetuj + konec ≤ 1

1. 2červená $\rightarrow$ start, umocňuj1, 2červená, světlemodrá, fialová
2. umocňuj1, světlemodrá, fialová $\rightarrow$ umocňuj1, světlemodrá, 2tmavofialová
3. umocňuj1, světlemodrá $\rightarrow$ umocňuj2, modrá
4. umocňuj2, tmavofialová $\rightarrow$ umocňuj2, fialová
5. umocňuj2, světlemodrá $\rightarrow$ umocňuj1, světlemodrá
6. umocňuj2 $\rightarrow$ kontroluj
7. kontroluj, červená, fialová $\rightarrow$ kontroluj, tmavočervená
8. kontroluj, červená $\rightarrow$ resetuj, červená
9. kontroluj $\rightarrow$ konec
10. resetuj, tmavočervená $\rightarrow$ resetuj, červená
11. resetuj, modrá $\rightarrow$ resetuj, světlemodrá
12. resetuj $\rightarrow$ umocňuj1, světlemodrá, fialová

Všimněte si, že díky žetonu *start* se první instrukce provede jen jednou. Navíc, díky „2červená“ na levé straně se tato instrukce neprovede pro $c = 1$; tehdy se neprovede vůbec nic.

Počáteční instrukce nám také vyrobí první světle modrý a první fialový žeton. Ve stavech *umocňuj1* a *umocňuj2* postupně m-krát vynásobíme počet fialových dvěma, čímž dosáhneme, že fialových je 2^m. Zároveň světle modré přebarvíme na modré.

Následně ve stavu *kontroluj* současně přebarvujeme červené a mažeme fialové. Pokud fialové dojdou dříve, ještě jich bylo málo – vrátíme tedy červené do původního stavu, současné modré přebarvíme zpět na světle modré a přidáme jim ještě jednu světle modrou navíc. Pokud červené a fialové dojdou najednou nebo dokonce dojdou červené dříve, už nemusíme dělat vůbec nic: aktuální počet modrých je správný.

Podúloha C (4 body): Fibonacci

Podobně jako v krajském kole a ve výše uvedené podúloze B, i zde použijeme speciální barvy žetonů pro reprezentaci stavu, ve kterém se právě výpočet nachází. Jelikož ale nesmíme používat omezení, budeme si muset pohlídat, aby se nám stavy nepomíchaly.

Na úvod si všimněme, že umíme rozeznat počáteční stav (existují červené žetony a nic jiného) a tehdy jednou udělat něco speciálního. Také umíme i bez použití omezení skončit v požadovaném koncovém stavu (stačí nemít žádnou instrukci, která má na levé straně jen modré žetony).

Jak by vypadal „normální“ výpočet n-tého Fibonacciho čísla? Např. takto:

1. $p \leftarrow F_0$
2. $q \leftarrow F_1$
3. Zopakuj n-krát:
4. $\quad r \leftarrow p + q$
5. $\quad p \leftarrow q$
6. $\quad q \leftarrow r$

Výstup: V proměnné p je hodnota F_n

K reprezentaci samotných Fibonacciho čísel během výpočtu budeme používat pampeliškové, qětinové a růžové žetony (zkráceně tedy „proměnné“ p, q, r). Výpočet našeho programu pro Pokladničku bude probíhat v kolech a po každém celém kole budou počty těchto žetonů odpovídat výše uvedenému programu.

Jelikož $F_0 = 0$ a $F_1 = 1$, na začátku potřebujeme mít nula žetonů typu p a jeden žeton typu q. Ten si vyrobíme následovně: všechny červené přebarvíme na černé, přičemž jako první budeme mít instrukci „červená, qětinová → černá, qětinová“ a jako druhou instrukci „červená → černá, qětinová“. Ve zcela prvním kroku výpočtu se tedy použije druhá instrukce, vyrobí nám jedno q, a od té doby dále se bude používat první, která ho jen zachová.

Každé kolo výpočtu našeho programu pak proběhne následovně:

- Na začátku cyklu máme $p = F_i$, $q = F_{i+1}$ a $r = 0$.
- Odstraň jeden černý žeton. Pokud to neumíš udělat, už jsme na konci výpočtu.
- Fáze 1: Za každou pampeliškovou přidej jednu růžovou.
- Když pampeliškové dojdou, máme $p = 0$ a $r = F_i$. V dosud nedotčeném q je stále F_{i+1}.
- Fáze 2: Za každou qětinovou přidej jednu růžovou a jednu pampeliškovou.
- Když qětinové dojdou, máme $p = F_{i+1}$, $q = 0$ a $r = F_i + F_{i+1} = F_{i+2}$.
- Fáze 3: Za každou růžovou přidej jednu qětinovou.
- Když růžové dojdou, máme $p = F_{i+1}$, $q = F_{i+2}$ a $r = 0$, čímž jsme úspěšně odsimulovali jednu iteraci výše uvedeného pseudokódu.

Prohlédněme si nyní výsledný program:

1. červená, qětinová → černá, qětinová
2. červená → černá, qětinová
3. fáze1, pampeliškový → fáze1, růžová
4. fáze1 → fáze2
5. fáze2, qětinová → fáze2, pampeliškový, růžová
6. fáze2 → fáze3
7. fáze3, růžová → fáze3, qětinová

8. fáze3 $\to \emptyset$
9. černá $\to$ fáze1
10. pampelišková $\to$ modrá
11. qětinová $\to \emptyset$

První dvě instrukce se provedou jen na začátku a od té doby na ně můžeme zapomenout. Počet černých žetonů nám nyní říká, kolik iterací výpočtu následujícího Fibonacciho čísla chceme dělat. Každá iterace začíná tím, že odstraníme jeden černý žeton a vyrobíme si žeton fáze1. Pomocné žetony fáze1, fáze2, fáze3 nám pomohou zajistit, aby výměny barev proběhly v tom pořadí, které chceme.

Když už i černé žetony dojdou, víme, že máme přesně F_n pampeliškových (a také přesně F_{n+1} qětinových). Pampeliškové žetony změníme na modré, qětinové zahodíme a jsme hotovi.