P-III-1 Kapitánka hledá posádku

Úlohu lze vyřešit tak, že vyzkoušíme všechny možnosti, jak může Kapitánka sestavit posádku. Pokaždé ověříme, zda je střední pirát v posádce dostatečně šikovný.

Posádka musí být souvislým úsekem pirátů, stačí tedy zkoušet jenom souvislé úseky. Takových úseků je $\mathcal{O}(n^2)$. Pro každý úsek můžeme určit středního piráta tak, že piráty z úseku zkopírujeme do pomocného pole a to uspořádáme. Takové řešení má časovou složitost $\mathcal{O}(n^3 \log n)$.

Na konkrétních schopnostech nezáleží

Všimněte si, že na konkrétních schopnostech pirátů příliš nezáleží. Zajímá nás jenom to, zda je daný pirát aspoň tak nadaný, jako Denis. Podmínku o středním pirátovi si zjevně můžeme přeformulovat následovně: *Posádka je plavbyschopná, pokud alespoň polovinu jejích členů tvoří piráti aspoň tak dobří, jako Denis.*

Při vyhodnocování, zda je konkrétní posádka plavbyschopná, proto stačí spočítat v ní dostatečně dobré piráty. Tím zlepšíme výše uvedené řešení na časovou složitost $\mathcal{O}(n^3)$.

Rychlejší řešení

Výše uvedené pozorování dokážeme elegantně matematicky popsat. Představme si, že máme jen dvě schopnosti pirátů: všichni horší než Denis mají schopnost -1, všichni aspoň tak dobří jako Denis mají schopnost $+1$.

Po této úpravě můžeme přeformulovat podmínku o plavbyschopné posádce: *Posádka je plavbyschopná právě tehdy, když má nezáporný součet schopností.*

Označíme si upravené pole naplněné hodnotami -1 a $+1$ jako A. Postupně pro každé i nyní provedeme následující výpočet: Začneme od i-tého piráta. Do posádky postupně přidáváme další piráty ($i+1$, $i+2$, …) a průběžně si počítáme součet jejich schopností. Vždy, když je po přidání piráta tento součet nezáporný, máme jednu možnou posádku.

Všimněte si, že jsme pokaždé v konstantním čase určili, zda je uvažovaná posádka pro Kapitánku vyhovující. Dostáváme tak řešení s časovou složitostí $\mathcal{O}(n^2)$.

Vzorové řešení

Klíčovým nástrojem k získání optimálního řešení jsou prefixové součty. Nechť $P[i]$ je součet prvních i prvků pole A. Hodnoty $P[i]$ spočítáme v lineárním čase: $P[0] = 0$ a $\forall i : P[i+1] = A[i] + P[i]$.

Pomocí prefixových součtů umíme v konstantním čase zjistit součet libovolného úseku pole A. Úsek obsahující piráty od indexu a do indexu $b-1$ včetně má součet $P[b] - P[a]$. Chceme nalézt všechny neprázdné úseky s nezáporným součtem. Zajímá nás tedy, pro kolik dvojic $a < b$ platí $P[b] \geq P[a]$. Jinými slovy, pro každý prefixový

součet potřebujeme zjistit, kolik *dřívějších* prefixových součtů je *menších nebo stejně velkých.*

Jedním možným efektivním řešením je použít intervalový strom, v němž si pro každou hodnotu h budeme pamatovat, kolikrát jsme ji už viděli v poli P. Takto dokážeme postupně pro každý index b v čase $\mathcal{O}(\log n)$ zjistit, kolik dobrých úseků na něm končí, a následně také v čase $\mathcal{O}(\log n)$ do stromu přidáme hodnotu právě zpracovaného prvku. Dostáváme tak řešení s celkovou časovou složitostí $\mathcal{O}(n \log n)$.

Existuje ovšem i lineární řešení. Prefixové součty budeme, stejně jako v předchozím řešení, postupně zpracovávat zleva doprava. V pomocné proměnné si budeme udržovat počet dříve zpracovaných prefixových součtů, které jsou menší nebo rovny tomu aktuálnímu. Zároveň budeme mít pomocné pole Q, přičemž v $Q[s]$ si budeme pamatovat počet dosud nalezených prefixových součtů s hodnotou přesně s.

Všimněte si, že v naší úloze se sousední prefixové součty (tj. sousední hodnoty v poli P) liší vždy jen o $+1$ nebo -1. Když se chceme posunout na další index, umíme proto snadno spočítat počet dříve zpracovaných prefixových součtů, které jsou menší nebo stejné. Stačí zjistit, zda se prefixový součet právě změnil o $+1$ nebo -1, a podle toho buď přičíst, nebo odečíst příslušný prvek pomocného pole Q. Nakonec ještě musíme pole Q aktualizovat, tedy po spočítání nového prefixového součtu s zvýšit $Q[s]$ o 1.

Jelikož se nám prefixové součty mění jenom o 1, zřejmě leží všechny v intervalu $\langle -n; n \rangle$, a proto nám stačí pomocné pole velikosti $\mathcal{O}(n)$. Výpočet prefixových součtů i druhý průchod s výpočtem odpovědi mají také zjevně časovou složitost $\mathcal{O}(n)$.

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

int main() {
    int N, D;
    cin >> N >> D;
    vector<int> A(N), P(N+1,0);
    for (int n=0; n<N; ++n) {
        int a; cin >> a;
        A[n] = (a>=D ? 1 : -1);
        P[n+1] = P[n] + A[n];
    }

    // videl[N+i] udává, kolik již zpracovaných prvků P mělo hodnotu přesně i
    vector<int> videl(2*N+1,0);
    // počet již zpracovaných prvků P, které byly <= aktuálnímu prvku
    int leq = 0;
    int odpoved = 0;

    for (int n=1; n<=N; ++n) {
        // přidáme prvek P[n-1] mezi již viděné prvky a mezi prvky <= než aktuální
        ++videl[ N+P[n-1] ];
        ++leq;
        // posuneme aktuální prvek na P[n], čímž buď přibudou,
        // nebo ubydou dříve zpracované prvky <= než aktuální prvek
        if (P[n] == P[n-1] + 1)
            leq += videl[ N+P[n] ];
        else
```

```
            leq -= videl[ N+P[n-1] ];
        odpoved += leq;
    }
    cout << odpoved << endl;
    return 0;
}
```

P-III-2 Závaží

Základním řešením této úlohy je řešení hrubou silou: vygenerujeme součty všech 2^n podmnožin závaží, v čase $\Theta(2^n \log(2^n)) = \Theta(n2^n)$ je uspořádáme a vybereme součet na indexu k.

Drobným zlepšením je, že místo třídění můžeme k nalezení prvku na indexu k použít lineární algoritmus, který je zobecněním algoritmu na nalezení mediánu posloupnosti. Pokud navíc šikovným způsobem generujeme všechny součty podmnožin, dostaneme algoritmus s časovou složitostí $\Theta(2^n)$.

Základní polynomiální řešení

Pro velká n si už samozřejmě nemůžeme dovolit vygenerovat všech n podmnožin. Je třeba využít skutečnosti, že k je (v porovnání s hodnotou 2^n) velmi malé. Budeme tedy chtít postupně generovat všechny podmnožiny uspořádané podle jejich součtu. Pomůže nám, když si celou situaci představíme jako orientovaný graf, jehož vrcholy představují jednotlivé množiny závaží – přesněji množiny jejich indexů.

Jedna možnost, jak může tento graf vypadat, je tato: Z každého vrcholu $v = \{a_1, \ldots, a_i\}$ vede pro každé $j \notin v$ hrana do vrcholu $v \cup \{j\}$. Například z vrcholu $\emptyset$ vede přesně n hran: do všech možných jednoprvkových množin. Z vrcholu $\{1\}$ a z vrcholu $\{2\}$ vede hrana do vrcholu $\{1, 2\}$. Každé hraně přiřadíme jako její délku hmotnost závaží, které právě přidáváme do množiny.

V takto postaveném grafu zjevně platí, že délka cesty z vrcholu $\emptyset$ do libovolného vrcholu v odpovídá celkové hmotnosti závaží, kterou daný vrchol v představuje.

Když nyní chceme postupně generovat všechny množiny závaží uspořádané podle velikosti, stačí na tento graf použít obyčejný Dijkstrův algoritmus na hledání nejkratších cest. Pořadí, v němž budeme vrcholy označovat za zpracované, bude přesně odpovídat pořadí množin uspořádanému podle hmotnosti.

Časová i paměťová složitost tohoto algoritmu je zjevně polynomiální vzhledem k n a k, přesně ji odhadnout ale není triviální a konkrétní odhad závisí na detailech zvolené implementace. Řádově platí, že postupně k vrcholů označíme za zpracované, a jelikož z každého z nich vede $\mathcal{O}(n)$ hran, dohromady přijdeme do $\mathcal{O}(nk)$ vrcholů. Protože každý z nich je nějaká $\mathcal{O}(n)$-prvková množina, bude časová složitost jejich zpracování řádově úměrná $\mathcal{O}(n^2k)$, plus se ještě možná někde objeví nějaký logaritmus navíc.

Pozorování o velikosti množin

Všimněte si, že všechny množiny, které se nacházejí na prvních k místech uspořádaného pořadí, musí nutně mít málo prvků. Proč? Nechť $\ell = \lceil \log_2 k \rceil$. Uvažujme libovolnou $(\ell + 1)$-prvkovou množinu M. Tato množina má $2^{\ell+1} - 1 > k$ vlastních

podmnožin a ty jsou všechny v uspořádaném pořadí před ní. Její pořadové číslo je proto určitě větší než k. Všechny množiny, které vygenerujeme, budou tudíž mít $\mathcal{O}(\log k)$ prvků.

Díky uvedenému pozorování dokážeme napsat implementaci výše popsaného řešení, která bude mít časovou složitost $\mathcal{O}(nk \cdot (\log k)^2)$ nebo podobnou. (Přesná časová složitost opět závisí na detailech implementace.)

Jiné polynomiální řešení

Než se dostaneme ke vzorovému řešení, uděláme ještě odbočku a ukážeme si jedno jiné řešení – o něco rychlejší a o dost jednodušší než to, které jsme právě popsali.

Začneme tím, že zvolíme jeden možný postup, jak vygenerovat hmotnosti všech podmnožin: budeme postupně přidávat závaží. Když nemáme žádné závaží, jediná hmotnost, kterou dokážeme vygenerovat, je nula. Představme si nyní, že už jsme zpracovali nějaká závaží a máme uspořádaný seznam hmotností všech jejich podmnožin. Jak nyní přidáme další závaží m_i? Už známe hmotnosti podmnožin, které toto závaží neobsahují. Hmotnosti podmnožin, které ho obsahují, spočítáme jednoduše tak, že ke každé hmotnosti v dosavadním seznamu přičteme m_i. Tím dostaneme dva uspořádané seznamy, které nakonec spojíme do jednoho stejným způsobem, jako například při třídění MergeSort.

Takové řešení by samozřejmě mělo časovou složitost exponenciální vzhledem k n. My ho ale snadno zlepšíme: po každém zpracovaném závaží si stačí zapamatovat k nejmenších hmotností, všechny ostatní jistě můžeme zahodit, neboť jsou příliš velké. Zpracování každého závaží tedy zvládneme v čase $\mathcal{O}(k)$. A protože závaží je n, dostáváme řešení s časovou složitostí $\mathcal{O}(nk)$.

Vzorové řešení

Vrátíme se teď k řešení, které množiny sestrojovalo postupným prohledáváním grafu. Hlavním problémem tohoto řešení je, že máme v grafu zbytečně mnoho hran. Například do vrcholu $\{3, 4, 7\}$ vede osm různých cest. Jelikož ale všechny mají stejnou délku, nenesou žádnou užitečnou informaci. Stačila by nám jenom jedna cesta. Ideální by bylo, kdybychom místo našeho původního grafu vymysleli nějaký jiný, který bude mít dvě vlastnosti:

- Do každého vrcholu vede právě jedna cesta, takže žádná hrana není zbytečná.
- Pro každou hranu $u \to v$ platí, že v je aspoň tak těžký jako u. Délky hran jsou tedy nezáporné.

Na libovolném takovém grafu bude Dijkstrův algoritmus zjevně stále fungovat, neboť když jsme ještě nějaký vrchol v neprohlásili za zpracovaný, tak nás vrcholy, do nichž se jde přes v, zatím nezajímají. Jsou totiž aspoň tak těžké jako vrchol v, a proto také ještě nemají být zpracované.

Jak vymyslet konstrukci takovéhoto grafu? Pomůže nám podívat se na problém z opačné strany. V hledaném grafu musí do každého vrcholu (kromě počátečního)

vést právě jedna hrana. Z opačné strany dostáváme, že z každého vrcholu musí existovat jednoznačná cesta do počátku. Můžeme tedy hledat nějaký snadný deterministický postup, jak libovolnou množinu závaží postupně převést na prázdnou.

Jedna taková konstrukce vypadá takto: Primárně se snažíme o 1 snížit největší index v množině, např. z množiny $\{3, 4, 7\}$ vyrobíme $\{3, 4, 6\}$. Jestliže to nelze provést, tak největší index jednoduše vyhodíme – např. z množiny $\{1, 6, 7\}$ dostaneme $\{1, 6\}$ a z množiny $\{1\}$ dostaneme $\emptyset$.

Zde je konkrétní příklad, jak z nějaké množiny vznikne opakováním těchto pravidel prázdná množina:

$$\{1, 3, 6\} \to \{1, 3, 5\} \to \{1, 3, 4\} \to \{1, 3\} \to \{1, 2\} \to \{1\} \to \emptyset.$$

Je zjevné, že žádný krok tohoto postupu nezvýší hmotnost množiny – buď jedno závaží nahradíme předcházejícím (které je nejvýše stejně těžké), nebo jedno závaží odstraníme.

Jak tedy bude vypadat náš graf? Budou v něm tytéž hrany, jenom s opačnou orientací:

- Z vrcholu $\emptyset$ vede jediná hrana do vrcholu $\{1\}$.
- Z vrcholu $v = \{a_1, \ldots, a_k\}$, v němž $a_k = n$, nevede žádná hrana.
- Jinak z něho vedou dvě hrany: do vrcholu, v němž do v přibude index $a_k + 1$, a do vrcholu, v němž je ve v obsažen index $a_k + 1$ místo indexu a_k.

Například jestliže $n > 6$, pak z vrcholu $\{1, 3, 6\}$ vedou hrany do vrcholů $\{1, 3, 6, 7\}$ a $\{1, 3, 7\}$.

Na tento graf použijeme Dijkstrův algoritmus a necháme ho běžet, dokud neprohlásí k vrcholů za zpracované. Jelikož z každého vrcholu vedou nejvýše dvě hrany, dohromady navštívíme a zpracujeme jenom $\mathcal{O}(k)$ vrcholů. Jako prioritní frontu můžeme použít obyčejnou haldu. Navíc prvky v ní stačí porovnávat podle vzdálenosti – náš graf je stromem a proto nepotřebujeme nijak kontrolovat duplikáty. Protože víme, že každý navštívený vrchol představuje množinu obsahující nejvýše $\mathcal{O}(\log k)$ čísel, pro každý navštívený vrchol nejpozději v čase $\mathcal{O}(\log k)$ vygenerujeme jeho množinu a potom také v $\mathcal{O}(\log k)$ přidáme jeho záznam do prioritní fronty. Celkově má tedy toto řešení časovou složitost $\mathcal{O}(k \log k)$.

Níže uvedená implementace je kvůli lepší čitelnosti trochu pomalejší, její časová složitost je $\mathcal{O}(k \cdot (\log k)^2)$. Zlepšit na $\mathcal{O}(k \log k)$ ji můžeme tím, že do prioritní fronty budeme ukládat jen ukazatele na jednotlivé vrcholy a dodefinujeme pro ně porovnávací funkci, která se bude dívat pouze na vzdálenost (nikoliv na jednotlivé prvky vektoru, v němž je uložen daný vrchol).

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

typedef pair< int, vector<int> > zaznam;

int main() {
    int N, K;
    cin >> N >> K;
    if (K == 1) { cout << 0 << endl; return 0; }

    vector<long long> M(N);
    for (auto &m : M) cin >> m;

    priority_queue <zaznam, vector<zaznam>, greater<zaznam> > PQ;
    PQ.push( { M[0], {0} } );

    for (int k=2; k<=K; ++k) {
        int d = PQ.top().first;
        vector<int> v = PQ.top().second;
        PQ.pop();

        if (k == K) cout << d << endl;

        int vs = v.size(), last = v[vs-1];
        if (last == N-1) continue;
        ++v[vs-1];
        PQ.push( { d+M[last+1]-M[last], v } );
        --v[vs-1];
        v.push_back(last+1);
        PQ.push( { d+M[last+1], v } );
    }
}
```

P-III-3 Stavebnice funkcí

Část A: minimum a maximum

Maximum z čísel a a b můžeme definovat takto: jestliže $a \geq b$, potom maximum je a, jinak je to b. Sestrojíme ho proto pomocí větvení – tedy konstrukcí, kterou jsme vymysleli v poslední části soutěžní úlohy krajského kola.

Nejsnadnější konstrukcí větvení je postup, v němž každou možnost výstupu vynásobíme predikátem, který říká, kdy chceme tento výstup. Maximum tedy můžeme zapsat následovně:

$$max(a, b) = geq(a, b) \cdot a + not(geq(a, b)) \cdot b.$$

Tuto konstrukci zrealizujeme pomocí Kompozitoru. Postupně sestrojíme:

- $tmpg_1 \equiv K[geq, v_1^2, mul]$
- $tmpg_2 \equiv K[K[geq, not], v_2^2, mul]$
- $max = K[tmpg_1, tmp_2, add]$

Existují i jiné konstrukce, například pomocí Cyklovače. Při ní můžeme nejprve nastavit výstupní hodnotu na b a potom ji $geq(a, b)$-krát změnit na a.

Minimum sestrojíme analogicky, stačí v konstrukci maxima vyměnit v_1^2 a v_2^2. Případně ještě jednodušší je použít vztah $\min(a, b) = a + b - \max(a, b)$.

Část B: poslední cifra

Jednou z možností bylo sestrojit si obecnou funkci počítající celočíselné dělení, nebo unární funkci počítající celočíselné dělení deseti. Takovou funkci sestrojíme jako pomocnou funkci v řešení části D této úlohy. Nyní si ukážeme jinou konstrukci, která je myšlenkově méně náročná.

Vytvoříme si pomocnou funkci, která se na číslech 0–8 chová stejně jako funkce s (successor, tj. následník), ale místo $s(9) = 10$ bude vracet nulu. (Je nám jedno, jak tato funkce odpoví na vstup větší než 9.)

Uvedenou funkci sestrojíme pomocí větvení, tedy podobnou konstrukcí, jako v části A. Jedna možnost vypadá následovně: $s_{10}(n) = geq(8, n) \cdot s(n)$. Tato funkce vrátí nulu pro libovolný vstup větší než 8.

Formálně tuto funkci sestrojíme takto: $s_{10} \equiv K[K[k_8^1, v_1^1, geq], s, mul]$.

Hledanou funkci *last* nyní vytvoříme pomocí Cyklovače. Stačí si uvědomit, že hodnotu $last(n)$ můžeme spočítat tak, že na nulu postupně n-krát použijeme funkci s_{10}. Inicializaci provedeme obyčejnou funkcí z (konstantní nula bez vstupů), jednu iteraci cyklu nám vypočítá funkce, která na svůj druhý vstup (neboli na starou hodnotu *tmp*) použije funkci s_{10}. Formálně tedy $last \equiv C[z, K[v_2^2, s_{10}]]$.

Část C: dolní celá část odmocniny

V této části úlohy použijeme dvě pozorování. Prvním z nich je, že $\lfloor\sqrt{n}\rfloor$ je největší x takové, že $n \geq x^2$. Druhým z nich je, že pro dolní celou část odmocniny platí $\forall n : \lfloor\sqrt{n}\rfloor \leq n$.

Jinými slovy, odmocninu z n můžeme nalézt tak, že projdeme všechna x od 0 do n a vybereme největší z nich, jehož kvadrát je ještě stále menší nebo roven n. To můžeme zapsat pseudokódem:

```
def sqrt ( n ):
    tmp = 0
    for i = 0 to n-1:
        if (i+1)*(i+1) <= n:
            tmp = i+1
    return tmp
```

Uvedený pseudokód nyní stačí přepsat do formální definice funkce pomocí Cyklovače.

Začneme pomocnou funkcí *sqrs*, pro niž platí $sqrs(n) = (n+1)^2$. Tuto funkci vyrobíme Kompozitorem: $sqrs \equiv K[s, s, mul]$. Podmínku $(i+1)^2 \leq n$ teď zapíšeme následovně: $podm \equiv K[v_2^2, K[v_1^2, sqrs], geq]$. Jednu iteraci cyklu v našem pseudokódu spočítá funkce f, která splňuje vztah

$$f(i, tmp) = podm(i, tmp) \cdot s(i) + not(podm(i, tmp)) \cdot tmp$$

(Slovně: je-li splněna podmínka, vrátíme $i + 1$, jinak vrátíme starou hodnotu *tmp*.)

Formálně tedy $f \equiv K[\,K[podm, K[v_1^2, s], mul],\ K[K[podm, not], v_2^2, mul],\ add]$.

Na závěr $sqrt \equiv C[z, f]$ a jsme hotovi.

Část D: Fibonacciho čísla (hlavní myšlenka)

Největším problémem při konstrukci Fibonacciho čísel je skutečnost, že na sestrojení následujícího čísla nám nestačí jedno, ale potřebujeme dvě předcházející. Na to náš Cyklovač, alespoň zdánlivě, není připraven – při jeho použití musíme následující hodnotu vypočítat z jedné předcházející.

Hlavní trik tedy bude spočívat v tom, že do té jedné hodnoty si budeme muset „zakódovat“ obě Fibonacciho čísla.

Existuje mnoho způsobů, jak zakódovat dvě přirozená čísla do jednoho – samozřejmě tak, abychom je uměli také jednoznačně získat nazpět. Zájemce o toto téma odkážeme na článek *https://en.wikipedia.org/wiki/Pairing_function*.

V našem řešení použijeme následující pozorování: $\forall n : F_n < 2^n$. Platnost dokážeme matematickou indukcí: platí $F_0 < 2^0$ i $F_1 < 2^1$, jako indukční krok dostáváme, že $\forall n \geq 2 : F_n = F_{n-1} + F_{n-2} < 2^{n-1} + 2^{n-2} < 2 \cdot 2^{n-1} = 2^n$.

Jestliže tedy známe číslo n a dostaneme číslo $kod(n) = F_{n+1} \cdot 2^n + F_n$, umíme z čísla $kod(n)$ sestrojit čísla F_n a F_{n+1}: číslo $kod(n)$ stačí celočíselně vydělit číslem 2^n, hodnoty F_{n+1} a F_n dostaneme jako podíl a zbytek po dělení.

Máme-li hodnotu $kod(n)$ a číslo n, dokážeme snadno spočítat hodnotu $kod(n+1)$: z hodnoty $kod(n)$ zjistíme F_{n+1} a F_n, z nich vypočítáme F_{n+2} a z hodnot F_{n+2} a F_{n+1} určíme výsledný kód. Funkci *kod* můžeme zapsat v pseudokódu:

```
def kod ( n ):
    tmp = 1                  # tedy tmp = F_1 * 2^0 + F_0
    for i = 0 to n-1:
        a = tmp mod pow(2,n)
        b = tmp div pow(2,n)
        c = a+b
        tmp = c * pow(2,n+1) + b
    return tmp
```

Takto definovaná funkce *kod* vrátí pro každé n hodnotu $F_{n+1} \cdot 2^n + F_n$. Samotné Fibonacciho číslo potom sestrojíme tak, že z příslušné hodnoty *kod* vytáhneme jenom to číslo, které nás zajímá: $fib(n) = kod(n) \bmod 2^n$.

Část D: Fibonacciho čísla (pomocné konstrukce)

Nejprve si sestrojíme několik pomocných funkcí.

První z nich bude funkce *div*, která počítá celočíselné dělení. Přesněji, chceme, aby pra všechna $y > 0$ platilo $div(x, y) = \lfloor x/y \rfloor$. Postupujeme podobně jako u odmocniny: odpovědí je největší z, pro které platí $x \geq yz$, a toto z najdeme pomocí Cyklovače (protože víme, že $z \leq x$, víme také, kolik nejvýše iterací bude třeba).

Funkci *vid* (*div* s prohozeným pořadím parametrů) zapíšeme v pseudokódu:

```
def vid ( y, x ):
    tmp = 0
    for i = 0 to y-1:
        if (i+1)*y <= x:
            tmp = i+1
    return tmp
```

Přepis této konstrukce do formálního zápisu již přenecháme čtenáři. Funkci *div* poté sestrojíme z funkce *vid* tak, že pomocí Kompozitoru vyměníme pořadí vstupů.

(Můžete si všimnout, že z naší konstrukce navíc plyne, že $\forall x : div(x, 0) = 0$. Toto však v dalším řešení nikde nebudeme potřebovat.)

Funkci *mod* (zbytek po celočíselném dělení) nyní sestrojíme snadno: platí

$$mod(x, y) = x - y \cdot div(x, y).$$

Dále si vytvoříme pomocné funkce *prvni* a *druhy*, které pro dané x a n spočítají $x \operatorname{div} 2^n$ a $x \bmod 2^n$. Pro transformaci opačným směrem si pomocí Kompozitoru složíme funkci *dvojice*, která pro vstup (x, y, n) vrátí hodnotu $x \cdot 2^n + y$.

Část D: Fibonacciho čísla (hlavní konstrukce)

Funkci *kod*, jejíž pseudokód jsme uvedli výše, chceme sestrojit pomocí Cyklovače. Potřebujeme proto vytvořit funkci, která spočítá jednu iteraci cyklu: z hodnot n a *tmp* (přičemž víme, že $tmp = F_{n+1} \cdot 2^n + F_n$) máme vypočítat novou hodnotu $F_{n+2} \cdot 2^{n+1} + F_{n+1}$.

Tuto funkci (nazveme ji *krok*) zapíšeme pomocí již sestrojených pomocných funkcí následovně:

$$krok(n, tmp) = dvojice(\ prvni(tmp, n) + druhy(tmp, n),\ prvni(tmp, n),\ n + 1).$$

Konstrukce funkce *krok* pomocí Kompozitoru je pracná, ale zjevná. Na závěr už jenom Cyklovačem sestrojíme funkci *kod* jako $C[k_1^0, krok]$ a následně Kompozitorem vytvoříme funkci *fib*, která splňuje $\forall n : fib(n) = druhy(kod(n), n)$.