A-III-1 Kapitánka hľadá posádku

Úlohu môžeme vyriešiť, tak že vyskúšame všetky možnosti, akú posádku môže Kapitánka zostaviť a zakaždým overíme, či je stredný pirát v posádke dostatočne šikovný.
Keďže posádka musí byť súvislý úsek pirátov, stačí skúšať iba tie. Tých je $O(n^2)$. Pre každý úsek vieme stredného piráta nájsť tak, že pirátov v úseku skopírujeme do pomocného poľa a to usporiadame. Takéto riešenie má časovú zložitosť $O(n^3 \log n)$.

Na konkrétnych schopnostiach nezáleží

Všimnime si, že na konkrétnych schopnostiach pirátov až tak veľmi nezáleží. Jediné, čo nás zaujíma, je, či je daný pirát aspoň tak nadaný ako Denis. Podmienku o strednom pirátovi si zjavne môžeme preformulovať nasledovne: *Posádka je plavbyschopná, ak aspoň polovicu jej členov tvoria piráti aspoň takí dobrí ako Denis.*
Pri vyhodnocovaní, či je konkrétna posádka plavbyschopná, preto stačí spočítať dostatočne dobrých pirátov v nej. Tým vieme vyššie uvedené riešenie vylepšiť na časovú zložitosť $O(n^3)$.

Rýchlejšie riešenie

Vyššie uvedené pozorovanie si vieme aj elegantne matematicky popísať. Predstavme si, že máme len dve schopnosti pirátov: všetci horší ako Denis majú schopnosť -1 a všetci, čo sú aspoň takí dobrí ako Denis, majú schopnosť $+1$.
Po tejto úprave vieme znova preformulovať podmienku o plavbyschopnej posádke: *Posádka je plavbyschopná práve vtedy, ak má nezáporný súčet schopností.*
Označme si upravené pole naplnené hodnotami -1 a $+1$ ako A. Postupne pre každé i teraz spravíme nasledovné: Začneme od i-teho piráta. Postupne do posádky pridávame ďalších $(i+1, i+2, \dots)$ a priebežne si počítame súčet ich schopností. Vždy, keď je po pridaní piráta tento súčet nezáporný, máme jednu možnú posádku.
Všimnime si, že tu sme vždy vedeli v konštantnom čase povedať, či je posádka pre kapitánku vyhovujúca, a teda dostávame riešenie s časovou zložitosťou $O(n^2)$

Vzorové riešenie

Kľúčovým prostriedkom na optimálne riešenie boli prefixové súčty. Nech $P[i]$ je súčet prvých i políčok poľa A. Hodnoty $P[i]$ vieme spočítať v lineárnom čase: $P[0] = 0$ a $\forall i : P[i+1] = A[i] + P[i]$.
Pomocou prefixových súčtov vieme v konštantnom čase vypočítať súčet ľubovoľného úseku poľa A. Úsek obsahujúci pirátov od indexu a po index $b-1$ vrátane má súčet $P[b] - P[a]$. My chceme spočítať všetky neprázdne úseky s nezáporným súčtom. Zaujíma nás teda, pre koľko dvojíc $a < b$ platí $P[b] \geq P[a]$. Teda chceme zistiť pre každý prefixový súčet, koľko spomedzi **skorších** prefixových súčtov je **menších alebo rovných** ako on.
Jedným možným efektívnym riešením je použiť intervalový strom, v ktorom si pre každú hodnotu h budeme pamätať, koľkokrát sme ju už videli v poli P. Takto vieme postupne pre každý index b v čase $O(\log n)$ zistiť, koľko dobrých úsekov na ňom končí, a následne tiež v čase $O(\log n)$ do stromu pridať hodnotu práve spracovaného prvku. Dostávame teda riešenie s celkovou časovou zložitosťou $O(n \log n)$.

Existuje však aj lineárne riešenie. Prefixové súčty budeme, rovnako ako v predchádzajúcom riešení, postupne spracovávať zľava doprava. V pomocnej premennej si budeme udržiavať počet skôr spracovaných prefixových súčtov, ktoré sú menšie alebo rovné ako ten aktuálny. Zároveň budeme mať pomocné pole, kde si budeme na i-tej pozícii pamätať počet doteraz videných prefixových súčtov s hodnotou presne i.
Všimnime si, že v našej úlohe sa susedné prefixové súčty (t.j. susedné hodnoty v poli P) líšia vždy len o $+1$ alebo -1. Keď sa chceme posunúť na ďalší index, vieme ľahko vypočítať počet skôr spracovaných prefixových súčtov, ktoré sú od neho menšie alebo rovné. Stačí zistiť, či sa práve prefixový súčet zmenil o $+1$ alebo -1 a podľa toho buď pripočítať alebo odpočítať príslušné políčko pomocného poľa. Následne v pomocnom poli inkrementujeme políčko zodpovedajúce práve spracovanej hodnote.
Keďže sa nám prefixové súčty menia len o 1, tak zrejme ležia všetky v intervale $\langle -n; n \rangle$ a teda nám stačí pomocné pole veľkosti $O(n)$. Výpočet prefixových súčtov aj druhý prechod s výpočtom odpovede tiež zjavne majú časovú zložitosť $O(n)$.

Listing programu (C++)

```
#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]; }

    vector<int> videl(2*N+1,0); // videl[N+i] hovorí, koľko už spracovaných prvkov P malo hodnotu presne i
    int leq = 0; // počet už spracovaných prvkov P ktoré boli menšie alebo rovné ako aktuálny
    int odpoved = 0;

    for (int n=1; n<=N; ++n) {
        // pridáme prvok P[n-1] medzi už videné, aj medzi prvky <= ako aktuálny
        ++videl[ N+P[n-1] ];
        ++leq;
        // posunieme aktuálny prvok na P[n], čím buď pribudnú alebo ubudnú skôr spracované <= od neho
        if (P[n] == P[n-1] + 1) leq += videl[ N+P[n] ]; else leq -= videl[ N+P[n-1] ];
        odpoved += leq;
    }
    cout << odpoved << endl;
}
```

A-III-2 Závažia

Základným riešením tejto úlohy je riešenie hrubou silou: vygenerujeme súčty všetkých 2^n podmnožín závaží, v čase $\Theta(2^n \log(2^n)) = \Theta(n2^n)$ ich usporiadame a vyberieme ten na indexe k.
Drobným zlepšením je, že namiesto triedenia vieme na nájdenie prvku na indexe k použiť lineárny algoritmus, ktorý je zovšeobecnením algoritmu na nájdenie mediánu postupnosti. Ak navyše šikovným spôsobom generujeme všetky súčty podmnožín, dostaneme algoritmus, ktorého časová zložitosť je len $\Theta(2^n)$.

Základné polynomiálne riešenie

Pre veľké n si už samozrejme nevieme dovoliť vygenerovať všetkých n podmnožín. Treba využiť, že k je (v porovnaní s hodnotou 2^n) veľmi malé. Budeme teda chcieť postupne generovať všetky podmnožiny usporiadané podľa ich súčtu. Pomôže nám, ak si celú situáciu predstavíme ako orientovaný graf, ktorého vrcholy predstavujú jednotlivé množiny závaží – presnejšie, množiny ich indexov.
Jedna možnosť, ako môže tento graf vyzerať: Z každého vrcholu $v = \{a_1, \ldots, a_i\}$ povedie pre každé $j \notin v$ hrana do vrcholu $v \cup \{j\}$. Teda napr. z vrcholu $\emptyset$ povedie presne n hrán: do všetkých možných 1-prvkových množín. Z vrcholu $\{1\}$ aj z vrcholu $\{2\}$ povedie hrana do vrcholu $\{1, 2\}$. Každej hrane priradíme ako dĺžku hmotnosť závažia, ktoré práve pridávame do množiny.
V takto postavenom grafe zjavne platí, že dĺžka cesty z vrcholu $\emptyset$ do ľubovoľného vrcholu v zodpovedá celkovej hmotnosti závaží, ktoré daný vrchol v predstavuje.
Ak teraz chceme postupne generovať všetky množiny závaží usporiadané podľa veľkosti, stačí na tento graf použiť obyčajný Dijkstrov algoritmus na hľadanie najkratších ciest. Poradie, v ktorom budeme vrcholy označovať za spracované, bude presne zodpovedať poradiu usporiadanému podľa hmotnosti.

Časová aj pamäťová zložitosť tohto algoritmu je zjavne polynomiálna od n a k, presne ju odhadnúť však nie je úplne triviálne a konkrétny odhad závisí od presných detailov zvolenej implementácie. Rádovo platí, že postupne k vrcholov označíme za spracované, a keďže z každého z nich vedie $O(n)$ hrán, dokopy objavíme $O(nk)$ vrcholov. A keďže každý z nich je nejaká $O(n)$-prvková množina, bude časová zložitosť ich spracovania rádovo úmerná $O(n^2k)$, plus sa ešte možno niekde zjaví nejaký logaritmus navyše.

Pozorovanie o veľkosti množín

Môžeme si všimnúť, že všetky množiny, ktoré sa nachádzajú na prvých k miestach usporiadaného poradia, musia nutne mať málo prvkov. Prečo? Nech $\ell = \lceil \log_2 k \rceil$. Uvažujme ľubovoľnú $(\ell + 1)$-prvkovú množinu M. Táto množina má $2^{\ell+1} - 1 > k$ vlastných podmnožín a tie sú všetky v usporiadanom poradí pred ňou, preto jej poradové číslo je určite väčšie ako k. Všetky množiny, ktoré vygenerujeme, budú teda mať $O(\log k)$ prvkov.

Vďaka tomuto pozorovaniu teraz vieme spraviť implementáciu vyššie popísaného riešenia, ktorá bude mať časovú zložitosť $O(nk(\log k)^2)$ alebo podobnú. (Opäť, presná časová zložitosť závisí od detailov implementácie.)

Iné polynomiálne riešenie

Skôr, než sa dostaneme k vzorovému riešeniu, spravíme ešte odbočku a ukážeme si jedno iné riešenie – o chlp rýchlejšie a o dosť jednoduchšie ako to, ktoré sme si ukázali vyššie.
Začneme tým, že zoberieme jeden možný postup, ako vygenerovať hmotnosti všetkých podmnožín: budeme postupne pridávať závažia. Ak nemáme žiadne závažia, jediná hmotnosť, ktorú vieme vygenerovať, je nula. Predstavme si teraz, že už sme spracovali nejaké závažia a máme usporiadaný zoznam hmotností všetkých ich podmnožín. Ako teraz pridať ďalšie závažie m_i? Už poznáme hmotnosti podmnožín, ktoré toto závažie neobsahujú. No a hmotnosti podmnožín, ktoré ho obsahujú, vypočítame jednoducho tak, že ku každej hmotnosti na doterajšom zozname pripočítame m_i. Takto dostaneme dva usporiadané zoznamy, ktoré nakoniec spojíme do jedného rovnakým spôsobom ako napr. pri triedení MergeSort.
Takéto riešenie by samozrejme malo časovú zložitosť exponenciálnu od n. My ho ale ľahko vieme vylepšiť: po každom spracovanom závaží si stačí zapamätať k najmenších hmotností, všetky ostatné zjavne môžeme zahodiť, lebo už sú určite priveľké. Spracovanie každého závažia teda zvládneme v čase $O(k)$. A keďže závaží je n, dostávame riešenie s časovou zložitosťou $O(nk)$.

Vzorové riešenie

Vráťme sa ale v myšlienkach k riešeniu, ktoré množiny zostrojovalo tak, že postupne prehľadávalo graf. Hlavným problémom tohto riešenia je, že máme v grafe zbytočne veľa hrán. Napr. do vrcholu $\{3, 4, 7\}$ vedie až osem rôznych ciest. Keďže ale všetky majú rovnakú dĺžku, nenesú žiadnu užitočnú informáciu. Stačila by nám aj jedna cesta. Ideálne by bolo, keby sme namiesto nášho pôvodného grafu vedeli vymyslieť nejaký iný, ktorý bude mať dve vlastnosti:

- Do každého vrcholu vedie práve jedna cesta, takže žiadna hrana nie je zbytočná.
- Pre každú hranu $u \to v$ platí, že v je aspoň tak ťažký ako u. Dĺžky hrán sú teda nezáporné.

Na ľubovoľnom takomto grafe bude Dijkstrov algoritmus zjavne stále fungovať, lebo ak sme ešte nejaký vrchol v neprehlásili za hotový, tak nás vrcholy, do ktorých sa ide cez v, ešte nezaujímajú, keďže sú aspoň také ťažké ako on, a teda tiež ešte nemajú byť hotové.
Ako vymyslieť konštrukciu takéhoto grafu? Pomôcť môže, keď sa na problém pozrieme z opačnej strany. V grafe, ktorý hľadáme, musí do každého vrcholu (okrem začiatku) viesť práve jedna hrana. Z opačnej strany teda dostávame, že z každého vrcholu musí existovať jednoznačná cesta do začiatku. Môžeme teda hľadať nejaký ľahko popísateľný deterministický postup, ako ľubovoľnú množinu závaží postupne prerobiť na prázdnu.
Jedna takáto konštrukcia vyzerá nasledovne: Primárne sa snažíme o jedno zmenšiť najväčší index v množine, napr. z množiny $\{3, 4, 7\}$ vyrobíme $\{3, 4, 6\}$. Ak sa toto nedá spraviť, tak najväčší index jednoducho vyhodíme – napr. z množiny $\{1, 6, 7\}$ spravíme $\{1, 6\}$ a z množiny $\{1\}$ spravíme $\emptyset$.
Tu je konkrétny príklad, ako postupne z nejakej množiny vznikne opakovaním týchto pravidiel prázdna množina:

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

Je zjavné, že žiadny krok tohto postupu nezvýši hmotnosť množiny – buď jedno závažie nahradíme predchádzajúcim (ktoré je nanajvýš rovnako ťažké) alebo jedno závažie odstránime.
Ako teda bude vyzerať náš graf? Budú v ňom tieto isté hrany, len opačným smerom.

- Z vrcholu $\emptyset$ vedie jediná hrana do vrcholu $\{1\}$.
- Z vrcholu $v = \{a_1, \ldots, a_k\}$ v ktorom $a_k = n$ už nevedú žiadne hrany.
- Inak z neho vedú dve hrany: do vrcholu v ktorom do v pribudne index $a_k + 1$ a do vrcholu, v ktorom je vo v index $a_k + 1$ namiesto indexu a_k.

Príklad: ak $n > 6$ tak z vrcholu $\{1, 3, 6\}$ povedú hrany do vrcholov $\{1, 3, 6, 7\}$ a $\{1, 3, 7\}$.

Na tento graf použijeme Dijkstrov algoritmus a necháme ho bežať, kým neprehlási k vrcholov za hotové. Keďže z každého vrcholu vedú nanajvýš dve hrany, dokopy objavíme a spracujeme len $O(k)$ vrcholov. Ako prioritnú frontu môžeme použiť obyčajnú haldu. Navyše prvky v nej stačí porovnávať podľa vzdialenosti – náš graf je strom, a teda nepotrebujeme nijak kontrolovať duplikáty. No a keďže vieme, že každý objavený vrchol je množina obsahujúca nanajvýš $O(\log k)$ čísel, tak pre každý objavený vrchol najskôr v $O(\log k)$ vygenerujeme jeho množinu a potom tiež v $O(\log k)$ pridáme jeho záznam do prioritnej fronty. Celkovo má teda toto riešenie časovú zložitosť $O(k \log k)$.
Nižšie uvedená implementácia je kvôli lepšej čitateľnosti trochu pomalšia, jej časová zložitosť je $O(k(\log k)^2)$. Zlepšiť na $O(k \log k)$ ju vieme tak, že do prioritnej fronty budeme ukladať len ukazovatele na jednotlivé vrrcholy a dodefinujeme pre ne porovnávaciu funkciu, ktorá sa bude pozerať len na vzdialenosť (a nie na jednotlivé prvky vektoru, v ktorom je uložený daný vrchol).

Listing programu (C++)

```
#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 } );
    }
}
```

A-III-3 Stavebnica funkcií

Podúloha A: minimum a maximum

Maximum z čísel a a b si vieme definovať nasledovne: ak $a \geq b$, tak maximum je a, inak je to b. Zostrojiť ho teda môžeme pomocou vetvenia – teda konštrukciou, ktorú sme vymysleli v poslednej podúlohe krajského kola. Asi najľahšou konštrukciou vetvenia je postup, pri ktorom každú možnosť výstupu vynásobíme predikátom, ktorý hovorí, kedy chceme tento výstup. Maximum teda môžeme zapísať nasledovne:

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

No a túto konštrukciu už ľahko zrealizujeme pomocou Kompozítora. Postupne vyrobíme:

- $tmp_1 \equiv K[geq, v_1^2, mul]$
- $tmp_2 \equiv K[K[geq, not], v_2^2, mul]$
- $max = K[tmp_1, tmp_2, add]$

Existuje aj viacero iných konštrukcií, napríklad pomocou Cyklovača. Pri takejto konštrukcii môžeme napríklad najskôr nastaviť výstupnú hodnotu na b a potom ju $geq(a, b)$-krát zmeniť na a.

Minimum vieme zostrojiť analogicky, stačí v konštrukcii maxima vymeniť v_1^2 a v_2^2. Prípadne na ešte jednoduchšiu konštrukciu vieme použiť vzťah $\min(a,b) = a + b - \max(a,b)$.

Podúloha B: posledná cifra

Jednou z možností, ako riešiť túto podúlohu, bolo zostrojiť si všeobecnú funkciu počítajúcu celočíselné delenie, alebo unárnu funkciu počítajúcu celočíselné delenie desiatimi. Takúto funkciu si zostrojíme ako pomocnú funkciu pre podúlohu D. Teraz si ukážeme jednu inú konštrukciu, ktorú považujeme za myšlienkovo jednoduchšiu.
Zostrojíme si pomocnú funkciu, ktorá sa na číslach 0-8 správa rovnako ako s (successor, t.j. nasledovník), ale namiesto $s(9) = 10$ bude vracať nulu. (Je nám jedno, čo táto funkcia spraví na vstupoch väčších ako 9.)
Takáto funkcia sa dá zostrojiť použitím vetvenia, teda podobnou konštrukciou ako v podúlohe A. Jedna možnosť vyzerá nasledovne: $s_{10}(n) = geq(8,n) \cdot s(n)$. Táto funkcia teda vráti nulu pre ľubovoľný vstup väčší ako 8.
Formálne vieme túto funkciu zostrojiť nasledovne: $s_{10} \equiv K[K[k_8^1, v_1^1, geq], s, mul]$.

Hľadanú funkciu $last$ teraz ľahko zostrojíme pomocou Cyklovača. Stačí, keď si uvedomíme, že hodnotu $last(n)$ vieme vypočítať tak, že na nulu postupne n-krát použijeme funkciu s_{10}. Inicializáciu teda spravíme obyčajnou funkciou z (konštantná nula bez vstupov), jednu iteráciu cyklu nám vypočíta funkcia, ktorá na svoj druhý vstup (čiže na starú hodnotu tmp) použije funkciu s_{10}. Formálne teda $last \equiv C[z, K[v_2^2, s_{10}]]$.

Podúloha C: dolná celá časť odmocniny

V tejto podúlohe použijeme dve pozorovania. Prvé z nich je, že $\lfloor\sqrt{n}\rfloor$ je najväčšie x také, že $n \geq x^2$. No a druhé z nich je, že pre dolnú celú časť odmocniny platí $\forall n : \lfloor\sqrt{n}\rfloor \leq n$.
Inými slovami, odmocninu z n vieme nájsť tak, že prezrieme všetky x od 0 po n a vyberieme najväčšie z nich, ktorého štvorec je ešte stále menší alebo rovný n. Toto vieme zapísať nasledovným pseudokódom:

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

A už nám teraz ostáva len vyššie uvedený pseudokód prerobiť na formálnu definíciu pomocou Cyklovača.
Začneme pomocnou funkciou $sqrs$, pre ktorú platí $sqrs(n) = (n+1)^2$. Túto funkciu vyrobíme Kompozítorom: $sqrs \equiv K[s, s, mul]$.
Samotnú odmocninu si teraz vyrobíme Cyklovačom, ale bude treba upresniť ešte jeden technický detail. Vo vyššie uvedenom pseudokóde používame okrem riadiacej premennej cyklu i aj hodnotu n, tú ale nemáme pri použití Cyklovača k dispozícii. Keby sme Cyklovačom vyrábali priamo unárnu funkciu $sqrt$, musela by každú iteráciu cyklu počítať binárna funkcia, ktorá novú hodnotu tmp vypočíta len z i a starej hodnoty tmp.
Spravíme to teda tak, že namiesto $sqrt$ definujeme binárnu funkciu $sqrtpom$, ktorá bude mať tú vlastnosť, že $sqrtpom(n,n) = sqrt(n)$. Pri definícii $sqrtpom$ už potom každú iteráciu cyklu budeme počítať funkciou f s nasledovnou vlastnosťou: ak $podm(i,n)$, tak $f(i,n,tmp)$ vráti $i+1$, inak $f(i,n,tmp)$ vráti tmp.
Takúto funkciu vieme zostrojiť nasledovne:

$$podm(i,n,tmp) = (i+1)^2 \leq n$$
$$f(i,n,tmp) = podm(i,n,tmp) \cdot s(i) + not(podm(i,n,tmp)) \cdot tmp$$

a jej formálny zápis vyzerá takto:

$$podm \equiv K[\ v_2^3,\ K[v_1^3, sqrs],\ geq\ ]$$
$$f \equiv K[\quad K[podm, K[v_1^3, s], mul],\quad K[K[podm, not], v_3^3, mul],\quad add\quad]$$

Teraz teda $sqrtpom \equiv C[k_0^1, f]$ a následne $sqrt \equiv K[v_1^1, v_1^1, sqrtpom]$.

Podúloha D: Fibonacciho čísla (hlavná myšlienka)

Hlavným problémom, na ktorý narazíme pri konštrukcii Fibonacciho čísel, je skutočnosť, že na zostrojenie nasledujúceho potrebujeme poznať nie jedno, ale hneď dve predchádzajúce. No a na to náš Cyklovač, aspoň zdanlivo, nie je pripravený – pri jeho použití musíme nasledujúcu hodnotu vypočítať z jednej predchádzajúcej. Hlavný trik teda bude v tom, že do tej jednej hodnoty si budeme musieť „zakódovať“ obe Fibonacciho čísla.
Existuje veľa spôsobov, ako zakódovať dve prirodzené čísla do jedného – samozrejme tak, aby sme ich vedeli aj jednoznačne získať naspäť. Záujemcov o túto tému odkážeme na článok `https://en.wikipedia.org/wiki/Pairing_function`.
V našom riešení použijeme nasledovné pozorovanie: $\forall n : F_n < 2^n$. Toto vieme dokázať matematickou indukciou: platí $F_0 < 2^0$ aj $F_1 < 2^1$, no a ako indukčný krok dostávame, že $\forall n \geq 2 : F_n = F_{n-1} + F_{n-2} < 2^{n-1} + 2^{n-2} < 2 \cdot 2^{n-1} = 2^n$.
Ak teda poznáme číslo n a poznáme číslo $kod(n) = F_{n+1} \cdot 2^n + F_n$, vieme z čísla $kod(n)$ zostrojiť čísla F_n aj F_{n+1}: číslo $kod(n)$ stačí celočíselne vydeliť číslom 2^n, hodnoty F_{n+1} a F_n dostaneme ako podiel a zvyšok po delení.
Keď poznáme hodnotu $kod(n)$ aj číslo n, vieme teraz ľahko vypočítať hodnotu $kod(n+1)$: z hodnoty $kod(n)$ zistíme F_{n+1} a F_n, z nich vypočítame F_{n+2} a z hodnôt F_{n+2} a F_{n+1} vypočítame výsledný kód. Funkciu kod teda vieme zapísať nasledovným pseudokódom:

```
def kod ( n ):
    tmp = 1                          # teda 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 funkcia kod pre každé n vráti hodnotu $F_{n+1} \cdot 2^n + F_n$. Samotné Fibonacciho čísla potom zostrojíme tak, že z príslušnej hodnoty kod vytiahneme len to číslo, ktoré nás zaujíma: $fib(n) = kod(n) \bmod 2^n$.

Podúloha D: Fibonacciho čísla (pomocné konštrukcie)

Aby sa nám výslednú funkciu zostrojovalo čo najpohodlnejšie, zostrojíme si najskôr niekoľko pomocných funkcií. Prvou z nich bude funkcia div, robiaca celočíselné delenie. Presnejšie, chceme, aby pre všetky $y > 0$ platilo $div(x, y) = \lfloor x/y \rfloor$. Toto spravíme podobne ako u odmocniny: povieme si, že odpoveďou je najväčšie z, pre ktoré platí $x \geq yz$, a toto z nájdeme pomocou Cyklovača (keďže vieme, že $z \leq x$, vieme, koľko nanajvýš iterácií treba).
Presnejšie, vid (div s prehodeným poradím parametrov) vieme zapísať nasledujúcim pseudokódom:

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

Prepis tejto konštrukcie do formálneho zápisu už prenechávame na čitateľa. Funkciu div následne zostrojíme z funkcie vid tak, že jej pomocou Kompozítora vymeníme poradie vstupov.
(Môžeme si všimnúť, že z našej konštrukcie navyše vyplýva, že $\forall x : div(x, 0) = 0$. Toto však v ďalšom riešení nikde nebudeme potrebovať.)

Funkciu mod (zvyšok po delení) už zostrojíme ľahko: platí $mod(x, y) = x - y \cdot div(x, y)$.

Teraz si už vieme vyrobiť pomocné funkcie $prvy$ a $druhy$, ktoré pre dané x a n vypočítajú $x \operatorname{div} 2^n$ a $x \bmod 2^n$. A pre transformáciu opačným smerom si vieme Kompozítorom poskladať aj funkciu $dvojica$, ktorá na vstupe (x, y, n) vráti hodnotu $x \cdot 2^n + y$.

Podúloha D: Fibonacciho čísla (hlavná konštrukcia)

Funkciu kod, ktorej pseudokód sme uviedli vyššie, si chceme vyrobiť pomocou Cyklovača. Potrebujeme teda vyrobiť funkciu, ktorá vypočíta jednu iteráciu cyklu: z hodnôt n a tmp (pričom vieme, že $tmp = F_{n+1} \cdot 2^n + F_n$) máme vypočítať novú hodnotu $F_{n+2} \cdot 2^{n+1} + F_{n+1}$.

Túto funkciu (nazvime ju napr. $krok$) vieme pomocou už zostrojených pomocných funkcií matematicky zapísať nasledovne:

$$\forall n:\ \forall\, tmp:\ krok(n, tmp) = dvojica(\ prvy(tmp, n) + druhy(tmp, n),\ prvy(tmp, n),\ n + 1)$$

Konštrukcia funkcie $krok$ pomocou Kompozítora je pracná ale zjavná. No a na záver už len Cyklovačom zostrojíme funkciu kod ako $C[k_1^0, krok]$ a následne Kompozítorom vyrobíme funkciu fib, ktorá spĺňa $\forall n : fib(n) = druhy(kod(n), n)$.

Podúloha D: Fibonacciho čísla (alternatívne riešenia)

Pri voľbe jednoznačného kódovania dvoch Fibonacciho čísel do jedného sa tiež dalo napríklad využiť skutočnosť, že dĺžky desiatkových zápisov dvoch po sebe idúcich Fibonacciho čísel sa líšia nanajvýš o jedna.
Úplne iný typ riešenia tejto podúlohy sa dal spraviť na základe kombinatorických úvah. Platí napríklad, že pre každé n je Fibonacciho číslo F_{n+2} rovné počtu n-znakových reťazcov núl a jedničiek, v ktorých nikde nemáme dve jedničky za sebou. Stačí teda v cykle prejsť cez všetky čísla od 0 po $2^n - 1$ a o každom z nich overiť, či má jeho zápis v dvojkovej sústave túto vlastnosť.

TRIDSIATY TRETÍ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Eduard Batmendijn, Michal Forišek, Samuel Gurský, Samuel Sládek, Emanuel Tesař
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2018