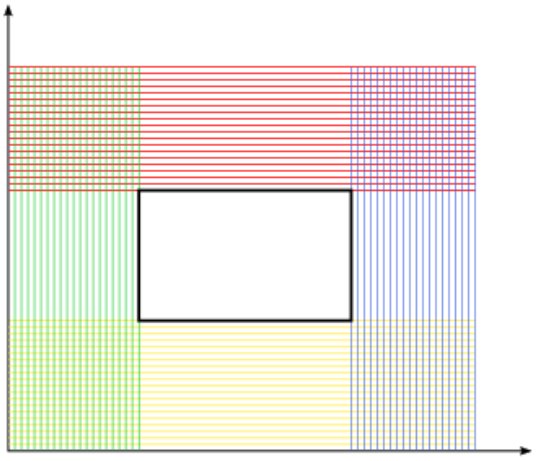
## B-I-1 Volebný bazén

### Zistíme, kde sa dá stavať

Jednou z prvých otázok, ktoré si položíme je, kde vôbec sa dá bazén postaviť. Hlavná podmienka je, že bazén sa musí nachádzať v mieste, kde sa prekrývajú všetky preferované oblasti Ivanových spoluobčanov. Ako vieme túto oblasť identifikovať a zistiť, či vôbec existuje?
Začneme najjednoduchším prípadom, keď je občan iba jeden ($n = 1$). Vtedy je jeho preferovaná oblasť výslednou oblasťou kde môžeme plánovať stavbu bazéna. Ak k nemu pridáme druhého občana ($n = 2$), potrebujeme do úvahy vziať aj jeho preferencie. Výsledná oblasť, kde môžeme stavbu bazéna zvažovať, je teraz zjavne prienikom týchto dvoch oblastí. Prienik neexistuje (presnejšie, je prázdny) ak jedna z obdĺžnikových oblastí celá leží naľavo, napravo, nad alebo pod tou druhou. Inými slovami, v jednej z vyšrafovaných oblastí.

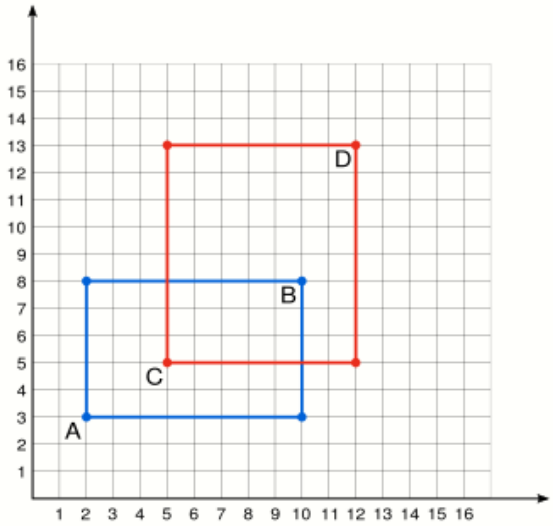


Túto situáciu vieme overiť jednoduchou podmienkou:

### Listing programu (Python)

```
from dataclasses import dataclass

@dataclass
class Bod:
    x: int
    y: int

@dataclass
class Obdlznik:
    l_dol: Bod
    p_hor: Bod

def maju_prienik(o1: Obdlznik, o2: Obdlznik) -> bool:
    # prvy obdlznik je vpravo od druheho alebo vlavo od druheo
    if o1.l_dol.x >= o2.p_hor.x or o1.p_hor.x <= o2.l_dol.x:
        return False
    # prvy obdlznik je nad druhym alebo pod druhym
    if o1.l_dol.y >= o2.p_hor.y or o1.p_hor.y <= o2.l_dol.y:
        return False
    return True
```

Ak prienik existuje, ako ho zostrojíme? Označme prvý obdĺžnik $O_1$, pričom tento je daný bodmi $A$ (ľavý, dolný), $B$ (pravý, horný) a druhý obdĺžnik $O_2$, daný bodmi $C$, $D$:




Zjavne výsledným prienikom týchto obdĺžnikov bude obdĺžnik $O$. Pokúsme sa teraz identifikovať jeho ľavý dolný roh. Jeho ľavý dolný roh môže byť len tak vľavo ako „pravejší“ z najľavejších bodov $A$, $C$ a len tak dole ako „hornejší“ z najdolnejších bodov $A$, $C$. Inými slovami, ľavý dolný roh $R_{ld}$ obdĺžnika $O$ bude mať súradnice $[\max(A_x, C_x), \max(A_y, C_y)]$. Obdobné tvrdenie platí aj pre pravý horný roh. Jeho súradnice vieme vyjadriť ako $[\min(B_x, D_x), \min(B_y, D_y)]$.
Čo sme odpozorovali? Pri jednom občanovi bol oblasťou vhodnou pre stavbu bazéna obdĺžnik. Pri zvážení preferencií druhého občana sme museli zistiť, či jeho preferovaná oblasť zdieľa nejakú plochu s našou doteraz identifikovanou oblasťou. Zistili sme, že ak tieto dve oblasti majú spoločný prienik, výslednou oblasťou je zase obdĺžnik. Takto vieme pokračovať pre všetkých občanov pričom na konci nám ostane výsledná obdĺžniková oblasť, kde môžeme bazén postaviť.

Poznámka: V nižšie uvedenej ukážkovej implementácii vzorového riešenia explicitne kontrolujeme existenciu neprázdneho prieniku tak, ako sme to popísali vyššie. Nie je to však potrebné. Namiesto toho môžeme len použiť vyššie popísané vzorce s maximami a minimami na spočítanie súradníc ľavého dolného a pravého horného rohu prieniku všetkých $n$ obdĺžnikov na vstupe. Až na konci stačí skontrolovať, či vypočítaný prienik skutočne existuje – teda či vypočítaný ľavý dolný roh naozaj leží aj viac vľavo, aj viac dole ako vypočítaný pravý horný roh.

### Rozmyslíme si, ako umiestniť bazén

Keď sme sa dostali sem, už poznáme presnú oblasť, v ktorej môže bazén ležať, a vieme, že je to nejaký neprázdny obdĺžnik $O$.
Preskočme teraz na chvíľu jeden krok riešenia a predstavme si, že nám k tomu niekto navyše prezradil presné rozmery $x$ a $y$ bazéna, ktorý máme postaviť. Ako by sme teraz úlohu dokončili a zvolili konkrétne umiestnenie bazéna? Najjednoduchšie je spraviť to vždy tak, že ľavý dolný roh bazéna umiestnime presne na ľavý dolný roh obdĺžnika $O$.
Jediné, čo nám ešte chýba do úplného riešenia, je teda určiť presné rozmery bazéna.

### Nájdeme presné rozmery bazéna

Označme si šírku obdĺžnika $O$ symbolom $X$ a výšku symbolom $Y$. Hľadané rozmery bazéna $x$ a $y$ musia spĺňať podmienku $x \cdot y = s$ (bazén musí mať správnu plochu) a zároveň musí platiť $1 \leq x \leq X$ a $1 \leq y \leq Y$ (bazén sa musí zmestiť do obdĺžnika $O$).
Zistiť, či také $x$ a $y$ existujú, vieme jednoducho spraviť v čase nanajvýš lineárnom od $s$. Postupne v cykle prejdeme všetky možnosti pre $x$ od $x = 1$ až po $x = \min(X, s)$. Pre každé takéto $x$ overíme, či bez zvyšku delí

$s$. Ak áno, dopočítame $y$ pomocou vzťahu $y = s/x$ a overíme, či takto získaná hodnota $y$ nie je väčšia ako $Y$. Pomocou tohto prístupu získavame prvé riešenie, ktorým vieme získať od 5 do 8 bodov v závislosti od použitého programovacieho jazyka. Jeho časová zložitosť je $O(n + s)$. Pamäťová zložitosť je konštantná, nakoľko si v každom bode programu pamätáme iba aktuálny prienik oblastí.

### Listing programu (Python)

```
from dataclasses import dataclass


@dataclass
class Bod:
    x: int
    y: int


@dataclass
class Obdlznik:
    l_dol: Bod
    p_hor: Bod


def spocitaj_prienik(o1: Obdlznik, o2: Obdlznik) -> Obdlznik:
    l_dol = Bod(max(o1.l_dol.x, o2.l_dol.x), max(o1.l_dol.y, o2.l_dol.y))
    p_hor = Bod(min(o1.p_hor.x, o2.p_hor.x), min(o1.p_hor.y, o2.p_hor.y))
    return Obdlznik(l_dol, p_hor)


def maju_prienik(o1: Obdlznik, o2: Obdlznik) -> bool:
    # prvy obdlznik je vpravo od druheho alebo vlavo od druheo
    if o1.l_dol.x >= o2.p_hor.x or o1.p_hor.x <= o2.l_dol.x:
        return False
    # prvy obdlznik je nad druhym alebo pod druhym
    if o1.l_dol.y >= o2.p_hor.y or o1.p_hor.y <= o2.l_dol.y:
        return False
    return True


s, n = [int(_) for _ in input().split()]
rect = []
for _ in range(n):
    x1, y1, x2, y2 = [int(_) for _ in input().split()]
    rect.append(Obdlznik(Bod(x1, y1), Bod(x2, y2)))

prienik = rect[0]
for obd in rect[1:]:
    if not maju_prienik(prienik, obd):
        print("Neexistuje")
        exit(0)
    prienik = spocitaj_prienik(prienik, obd)

prienik_x, prienik_y = (
    prienik.p_hor.x - prienik.l_dol.x,
    prienik.p_hor.y - prienik.l_dol.y,
)

for x in range(1, prienik_x + 1):
    y = s // x
    if x * y == s and y <= prienik_y:
        print(
            prienik.l_dol.x, prienik.l_dol.y, prienik.l_dol.x + x, prienik.l_dol.y + y
        )
        exit(0)

print("Neexistuje")
```

K vzorovému riešeniu nám už chýba len posledný myšlienkový krok. Môžeme si všimnúť, že všetky vyhovujúce $x$ aj $y$ sú deliteľmi $s$. Delitele čísla $s$ vieme vďaka často používanému triku nájsť aj rýchlejšie ako len prejdením všetkých čísel od 1 po $s$. Ak je totiž $d$ deliteľom $s$, tak potom je deliteľom $s$ aj číslo $s/d$. (To, že $d$ delí $s$ znamená, že existuje prirodzené číslo $k$, také, že $s = k \cdot d$. Potom ale $k$ je zjavne tiež deliteľom $s$ a zjavne $k = s/d$.) Uvedomme si ešte, že $d$ a $s/d$ nemusia byť nutne rôzne čísla. Príkladom je prípad $s = 25$, $d = 5$, $s/d = 5$. Tento prípad zjavne nastáva len vtedy, keď $s = d^2$, teda keď $s$ je štvorec a $d$ je jeho presná odmocnina.

Výhoda teda je, že ak hľadáme delitele postupne začínajúc od $d = 1$, pri nájdení deliteľa $d$ sa môžeme pozrieť aj na jeho dvojičku $s/d$ a rovno sme tak našli dva delitele. Navyše keďže skúšaná hodnota $d$ postupne rastie, jej dvojička $s/d$ musí nutne klesať. To ale znamená, že v momente keď $d$ prerastie $s/d$, môžeme s hľadaním deliteľov prestať, lebo sme už všetky delitele našli.

Prečo? V každej dvojičke deliteľov musí ten menší z nich mať hodnotu $\leq \sqrt{s}$, lebo keď zoberieme dve čísla väčšie ako $\sqrt{s}$, ich súčin bude väčší ako $s$. No a podmienku $d > s/d$ môžeme upraviť práve do tvaru $d^2 > s$, čiže $d > \sqrt{s}$. Preto v okamihu, keď $d$ prekročí $s/d$, sme už postupne hodnotou $d$ prešli všetky možnosti pre menšieho deliteľa v dvojičke.
Toto riešenie teda všetky delitele čísla $s$ nájde v čase priamo úmernom $\sqrt{s}$. Inými slovami, jeho celková časová zložitosť je $O(n + \sqrt{s})$. Pamäťová ostáva nezmenená.
Pri implementácii si ešte treba dať pozor na to, že keď nájdeme všetky dvojice deliteľov čísla $s$, tak každú z nich treba vyskúšať aj ako $(x, y)$, aj ako $(y, x)$. Toto skúšanie si prípadne vieme odpustiť, ak namiesto neho vždy použijeme menšieho deliteľa pre ten rozmer, ktorý má obdĺžnik $O$ kratší.

### Listing programu (Python)

```
from dataclasses import dataclass
from math import ceil, sqrt


@dataclass
class Bod:
    x: int
    y: int


@dataclass
class Obdlznik:
    l_dol: Bod
    p_hor: Bod


def spocitaj_prienik(o1: Obdlznik, o2: Obdlznik) -> Obdlznik:
    l_dol = Bod(max(o1.l_dol.x, o2.l_dol.x), max(o1.l_dol.y, o2.l_dol.y))
    p_hor = Bod(min(o1.p_hor.x, o2.p_hor.x), min(o1.p_hor.y, o2.p_hor.y))
    return Obdlznik(l_dol, p_hor)


def maju_prienik(o1: Obdlznik, o2: Obdlznik) -> bool:
    # prvy obdlznik je vpravo od druheho alebo vlavo od druheo
    if o1.l_dol.x >= o2.p_hor.x or o1.p_hor.x <= o2.l_dol.x:
        return False
    # prvy obdlznik je nad druhym alebo pod druhym
    if o1.l_dol.y >= o2.p_hor.y or o1.p_hor.y <= o2.l_dol.y:
        return False
    return True


s, n = [int(_) for _ in input().split()]
rect = []
for _ in range(n):
    x1, y1, x2, y2 = [int(_) for _ in input().split()]
    rect.append(Obdlznik(Bod(x1, y1), Bod(x2, y2)))

prienik = rect[0]
for obd in rect[1:]:
    if not maju_prienik(prienik, obd):
        print("Neexistuje")
        exit(0)
    prienik = spocitaj_prienik(prienik, obd)

prienik_x, prienik_y = (
    prienik.p_hor.x - prienik.l_dol.x,
    prienik.p_hor.y - prienik.l_dol.y,
)

for i in range(1, ceil(sqrt(s)) + 1):
    if i > min(prienik_x, prienik_y):
        break
    if s % i == 0 and (s // i) <= max(prienik_x, prienik_y):
        # i <= s//i
        dlzka_x, dlzka_y = i, s // i
        if prienik_x > prienik_y:
            dlzka_x, dlzka_y = dlzka_y, dlzka_x
        print(
            prienik.l_dol.x,
            prienik.l_dol.y,
            prienik.l_dol.x + dlzka_x,
            prienik.l_dol.y + dlzka_y,
        )
        exit(0)

print("Neexistuje")
```

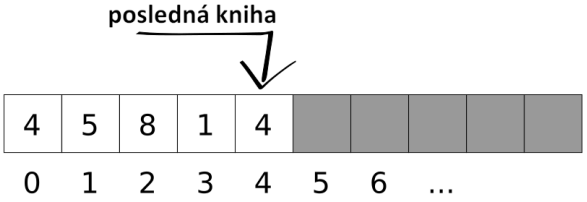
## B-I-2 Kopa kníh

### Bez robotického ramena

Ak sa robotické rameno nikdy nepoužije, tak nám stačí riešiť tieto dve operácie:

- pridanie knihy na vrch kopy
- odobranie knihy z vrchu kopy, s tým že vrátime informáciu o čísle odobratej knihy

Kopu si vieme reprezentovať jednoducho ako pole s ukazovateľom na vrchnú knihu v kope. Keď chceme pridať knihu na vrch, iba do políčka „pod ukazovateľom“ zapíšeme číslo knihy a ukazovateľ posunieme o $+1$. Ak chceme knihu odobrať, ukazovateľ posunieme o $-1$ (a číslo odobranej knihy je jasné, lebo ukazovateľ ukazoval na vrch kopy). Každú z operácii vieme vykonať v čase $O(1)$.




(Takejto dátovej štruktúre hovoríme *zásobník*, po anglicky *stack*.)

### Prvé úplné riešenie

K predchádzajúcemu riešeniu vieme pridať otáčanie robotického ramena pomocou hrubej sily: priamo v poli knihy poprehadzujeme tak, ako by to spravilo rameno. Presnejšie, postupne prejdeme časťou poľa v rozsahu ramena a vymeníme prvú knihu s poslednou, druhú s predposlednou, ... pričom si dávame pozor, aby sme skončili v polovici. (Ak by sme vymieňali ďalej, až po koniec poľa, tak by sme sa vrátili do pôvodného stavu, lebo každú dvojicu kníh – $k$-ta zospodu a $k$-ta zvrchu – by sme vymenili dvakrát.)

Časová zložitosť je $O(k)$ na každú operáciu ramena, kde $k$ je veľkosť ramena. Najhorší prípad nastane, keď na vstupe iba stále otáčame ramenom – v takom prípade spracovanie všetkých $n$ operácii zaberie až $O(n \cdot k)$ času. Čo sa pamäte týka, aké veľké pole potrebujeme? Musíme v ňom vedieť uchovať celú kopu, ktorá môže byť veľká až $n$ (prípad, keď knihy len prichádzajú na kopu). Pamäťová zložitosť je teda $O(n)$.

### Listing programu (C++)

```
#include <iostream>
using namespace std;

struct Kopa
{
    int vrchKopy = -1;
    int pole[1023456];

    void pridajKnihu(int id)
    {
        vrchKopy++;
        pole[vrchKopy] = id;
    }

    int odoberKnihu()
    {
        int id = pole[vrchKopy];
        vrchKopy--;
        return id;
    }

    void otocVrchnychNiekolkoKnih(int k)
    {
        k = min(k, vrchKopy + 1); // ak je kopa mala, otacame len to co mame
        for (int i = 0; i < k - 1 - i; i++)
        {
            int tmp = pole[vrchKopy - i];
```

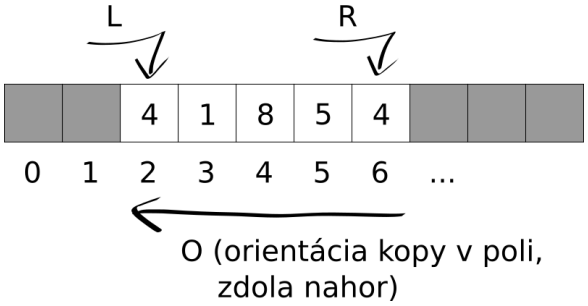
```
            pole[vrchKopy - i] = pole[vrchKopy - (k - 1 - i)];
            pole[vrchKopy - (k - 1 - i)] = tmp;
        }
    }
};

int main()
{
    int k;
    cin >> k;

    Kopa kopa = {};

    bool hotovo = false;
    while (!hotovo)
    {
        int cmd;
        cin >> cmd;
        switch (cmd)
        {
        case -2:
        {
            kopa.otocVrchnychNiekolkoKnih(k);
            break;
        }
        case -1:
        {
            kopa.odoberKnihu();
            break;
        }
        case -10:
        {
            cout << kopa.odoberKnihu() << endl;
            break;
        }
        case 0:
        {
            hotovo = true;
            break;
        }
        default:
            kopa.pridajKnihu(cmd);
        }
    }

    return 0;
}
```

### Veľké rameno

Zaoberajme sa teraz prípadom, keď rameno vždy otočí celú kopu. (Vo vstupoch 4-6 platilo $k \geq n$ a v dôsledku toho mali všetky použitia ramena túto vlastnosť.)

Postupujme rovnako ako v predchádzajúcim riešení. Kým len prichádzajú a odchádzajú knihy, tak je všetko v poriadku. Zrazu ale príde operácia ramena. V predošlom riešení sme manuálne celú kopu v poli otočili. Možno by nám ale stačilo si len niekde zaznačiť, že kopa je otočená? Totiž informácia o poradí kníh v kope tam stále je (iba sú knihy v opačnom poradí). To by mohlo byť výhodné napríklad vtedy, ak by hneď prišla ďalšia operácia ramena: vrátili sme sa do pôvodného stavu, ale stálo nás to len $O(1)$ času. Čo ak ale odoberieme alebo pridáme knihu? Potrebujeme nejako vedieť pracovať s tým, že kde v poli je začiatok kopy. Tak si aj na to spravme ukazovateľ. Nová dátová štruktúra vyzerá nasledovne:

- máme pole $P$, indexy $l, r$ (“left” a “right”) a boolean $o$ (“orientation”)
- Ak $o$, tak kopa od najspodnejšej po najvrchnejšiu knihu je tvorená knihami $P[l], P[l+1], \ldots, P[r]$.
- Ak $\neg o$, tak kopa od najspodnejšej po najvrchnejšiu knihu vyzerá $P[r], P[r-1], \ldots, P[l]$.
- (Ak $r < l$, tak kopa je prázdna.)




Implementácia jednotlivých operácii:

- Otočenie ramena: zaznačíme si, že poradie je opačné ako predtým.
- Pridanie knihy na vrch kopy: posunieme buď ukazovateľ na začiatok o $-1$ alebo ukazovateľ na koniec kopy o $+1$ podľa toho, či je kopa v poli naopak alebo nie (resp. že ktorý ukazovateľ reprezentuje vrch kopy).
- Odobratie knihy z vrchu kopy: posunieme buď ukazovateľ na začiatok o $+1$ alebo ukazovateľ na koniec o $-1$ podľa toho, či je kopa v poli naopak alebo nie.

Toto riešenie má ešte jeden háčik. Niektoré operácie nám vedia pohnúť ukazovateľ na začiatok záporným smerom. Čo ak sa takto dostaneme do záporných indexov? V niektorých programovacích jazykoch toto nemusí byť problém, ale čo ak pracujeme v jazyku, kde všetky polia začínajú od nuly? Ako zabezpečiť, aby sme nevyšli mimo poľa?
Ak by sme vedeli, že začiatok kopy nikdy nepadne pod napr. $-1023456$, tak by nám len stačilo všetky prvky posunúť o 1023456 doprava. My vieme, že každá operácia môže začiatok kopy posunúť prinajhoršom o $-1$, a keďže máme $n$ operácií, začiatok kopy nikdy nepadne pod $-n$. Podobne koniec kopy posunieme doprava prinajhoršom $n$-krát. Stačí nám teda na začiatku alokovať pole veľkosti $2 \cdot n$ a kopu kníh začať v jeho strede.
Riešenie, ktoré sme si práve popísali, funguje len pre vstupy, v ktorých pri každom použití ramena otočíme celú kopu kníh. Za tohto dodatočného predpokladu je ale efektívnejšie ako to predchádzajúce: každú operáciu vieme spraviť v konštantnom čase, teda v $O(1)$. Používame pole veľkosti $2n$, pamäťová zložitosť je teda $O(n)$.

### Listing programu (c++)

```
// otacacia_kopa.h

struct OtacaciaKopa
{
    bool kladnySmer = true;
    int l = 1023456, r = 1023455;
    int pole[2043456];

    void pridajKnihu(int id)
    {
        if (kladnySmer)
        {
            r++;
            pole[r] = id;
        }
        else
        {
            l--;
            pole[l] = id;
        }
    }

    int odoberKnihu()
    {
        if (kladnySmer)
        {
            int id = pole[r];
            r--;
            return id;
        }
        else
        {
            int id = pole[l];
            l++;
            return id;
```

```
        }
    }

    void otoc()
    {
        kladnySmer = !kladnySmer;
    }

    int velkost() const
    {
        return r - l + 1;
    }
};
```

Pre zhrnutie, dostali sme dátovú štruktúru, ktorá podporuje nasledovné operácie:

- obrátenie kopy
- vloženie prvku na vrch kopy
- odobratie prvku z vrchu kopy

Túto dátovú štruktúru budeme volať *otáčacia kopa*. Neskôr sa k nej ešte vrátime.

### Všeobecný prípad

Riešme teraz všeobecný prípad, kedy sa môže stať, že rameno otočí iba časť kopy.

Simulujme predchádzajúce riešenie. Kým sa celá kopa zmestí do ramena, tak je všetko v poriadku. Zrazu na kopu príde $(k+1)$-vá knihu. Vtedy naše riešenie môže byť nesprávne: ak by potom prišla operácia ramena, tak by sme mylne otočili vrchných $k+1$ kníh namiesto len $k$. Teda v otočení by mylne participovala aj spodná kniha. Tak ju odoberme a dajme si ju niekam nabok; potom je operácia ramena v poriadku. Teraz ak by vrchná kniha odišla, tak potrebujeme tú spodnú knihu pridať naspäť na spodok kopy (lebo už je opäť v rozsahu ramena). A to je vlastne celé, iba to potrebujeme zovšeobecniť:

- Knihy v rozsahu ramena (horných $k$ kníh v kope, resp. všetky, ak ich je momentálne menej) budeme mať v otáčacej kope.
- Všetky ostatné knihy hlbšie v kope budeme mať uložené v zásobníku (ako v úplne prvom riešení).

Jednotlivé operácie potom budeme simulovať nasledovne:

- Pri použití ramena len otočíme otáčaciu kopu.
- Keď pridávame knihu, pridáme ju na vrch otáčacej kopy. Ak je teraz v tejto kope viac ako $k$ kníh, tak odoberieme knihu zospodu a presunieme ju na vrch zásobníka.

  (Odobratie knihy zospodu môžeme implementovať samostatne, alebo môžeme kopu otočiť, odobrať knihu zhora a znova ju otočiť.)
- Keď odoberáme knihu, odoberieme knihu z vrchu otáčacej kopy. Ak máme nejaké knihy v zásobníku, tak hornú z nich odtiaľ vyberieme a vrátime ju na spodok otáčacej kopy. (Vkladanie na spodok vieme implementovať rovnako ako výber.)

Analýza zložitosti: Pri simulovaní každej operácie spravíme nanajvýš konštantne veľa operácií s otáčacou kopou – buď jedno otočenie, alebo v nejakom poradí jedno vloženie a jeden výber knihy. A taktiež spravíme nanajvýš konštantne veľa operácií so zásobníkom – buď nič alebo jedno vloženie alebo jeden výber.
Časová zložitosť je teda $O(1)$ na operáciu. Čo sa pamäte týka, potrebujeme si pamätať otáčaciu kopu ($O(n)$) a zásobník s ostatnými knihami (tiež $O(n)$), celková pamäťová zložitosť je teda $O(n)$.

### Listing programu (C++)

```
#include <iostream>
#include "otacacia_kopa.h"
using namespace std;

struct Kopa
{
```

```
    int velkostRamena;
    OtacaciaKopa rameno = {};

    int pocetBokom = 0;
    int bokom[1023456];

    Kopa(int velkostRamena0) : velkostRamena(velkostRamena0) {}

    void pridajKnihu(int id)
    {
        rameno.pridajKnihu(id);
        if (rameno.velkost() > velkostRamena)
        {
            rameno.otoc();
            int dajBokom = rameno.odoberKnihu();
            rameno.otoc();

            bokom[pocetBokom] = dajBokom;
            pocetBokom++;
        }
    }

    int odoberKnihu()
    {
        int id = rameno.odoberKnihu();
        if (pocetBokom > 0)
        {
            rameno.otoc();
            rameno.pridajKnihu(bokom[pocetBokom - 1]);
            rameno.otoc();
            pocetBokom--;
        }
        return id;
    }

    void otoc()
    {
        rameno.otoc();
    }
};

int main()
{
    int k;
    cin >> k;

    Kopa kopa(k);

    bool hotovo = false;
    while (!hotovo)
    {
        int cmd;
        cin >> cmd;
        switch (cmd)
        {
        case -2:
        {
            kopa.otoc();
            break;
        }
        case -1:
        {
            kopa.odoberKnihu();
            break;
        }
        case -10:
        {
            cout << kopa.odoberKnihu() << endl;
            break;
        }
        case 0:
        {
            hotovo = true;
            break;
        }
        default:
            kopa.pridajKnihu(cmd);
        }
    }

    return 0;
}
```

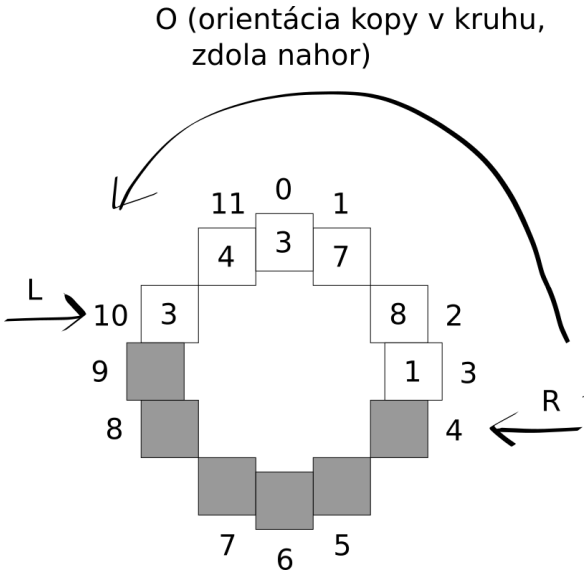
### Bonus: otáčacia kopa s menšou pamäťovou zložitosťou

Jedna neefektívnosť našej implementácie otáčacej kopy je jej pamäťová zložitosť. Predstavme si že $n$ je veľké

(také veľké, že si nemôžeme dovoliť mať pole veľkosti $2 \cdot n$), ale máme zaručené, že veľkosť kopy nikdy nepresiahne nejakú hodnotu $c$ takú, že pole veľkosti $O(c)$ sa ešte do pamäte zmestí. Naša implementácia otáčacej kopy je nepoužiteľná kvôli pamäťovým nárokom a napraviť to nie je úplne jednoduché. Rozmyslite si, že nestačí alokovať pole veľkosti $2 \cdot c$ a robiť to isté, čo vyššie. Totiž ak budeme napr. striedavo vkladať prvky na začiatok kopy a odoberať ich z konca, časom sa začiatok kopy dostane mimo poľa.

Jedno elegantné riešenie je nasledovné: majme pole $P$ veľkosti $c+1$ a "zlepme" jeho začiatok a koniec do kruhu tak, že indexy rastú v smere hodinových ručičiek (okrem prípadu keď od $c$ prejdeme na 0). Našu kopu potom reprezentujme ako výsek tohto kruhu:

- Majme ukazovatele $l$, $r$ a boolean $o$ udávajúci orientáciu kopy v poli.
- Ak $l < r$, tak prvky kopy sú $P[l], P[l+1], \ldots, P[r-1]$ (resp. v opačnom poradí, podľa $o$).
- Ak $l = r$, kopa je prázdna.
- Ak $l > r$, tak prvky kopy sú $P[l], P[l+1], \ldots, P[c], P[0], P[1], \ldots, P[r-1]$ (resp. v opačnom poradí, podľa $o$).




Keď pridávame na vrch, tak len podľa orientácie kopy pohneme $r$ v smere / $l$ proti smeru hodinových ručičiek. Keď z vrchu odoberáme, tak podobne ale v opačnom smere. Otočenie kopy zodpovedá zmenenie orientácie $o$ na opačnú.
Všimnime si, že pole je o jedno dlhšie ako maximálna veľkosť kopy, a teda v ňom vždy bude aspoň jedno voľné políčko. Toto sme spravili naschvál. Totiž ak by sme celé pole zaplnili knihami, po vložení poslednej z nich by sme mali $l = r$, čo u nás už zodpovedá úplne prázdnemu poľu.

### Bonus 2: to je vlastne obojsmerná fronta!

Otáčacia kopa je síce dosť neštandardná dátová štruktúra, v skutočnosti je ale skoro ekvivalentná inej, celkom zaužívanej dátovej štruktúre. Tou je *obojsmerná fronta* (anglicky "double-ended queue", v skratke "deque"). Tá podporuje nasledovné 4 operácie: vloženie alebo odobratie prvku zo začiatku alebo konca fronty.

Pod "skoro ekvivalentná" myslíme to, že jednu dátovú štruktúru vieme jednoduch implementovať pomocou druhej a naopak. Implementácia deque pomocou otáčacej kopy:

- Vloženie/odobranie prvku z konca fronty: na vrch kopy pridáme resp. z vrchu kopy odoberieme prvok.
- Vloženie/odobranie prvku zo začiatku fronty: kopu otočíme, potom na vrch pridáme resp. z vrchu odoberieme prvok, a nakoniec kopu opäť otočíme.

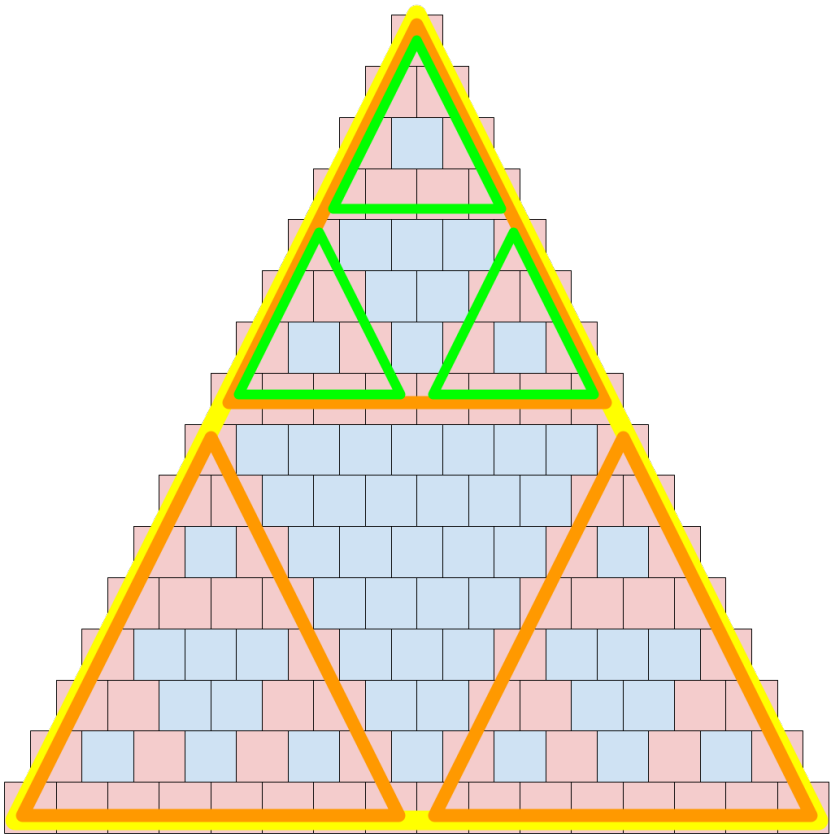
Implementácia otáčacej kopy pomocou deque:

- Otočenie kopy: do nejakej boolovskej premennej si zaznačím, že kopa má opačnú orientáciu, ako predtým. (Pre jednoduchosť označme tie dve možné orientácie 0 a 1.)
- Vloženie prvku na vrch: ak je orientácia 0, tak vkladáme na koniec, inak na začiatok fronty.
- Odobratie prvku z vrchu: ak je orientácia 0, tak odoberáme z konca, inak zo začiatku fronty.

## B-I-3 Pyramída z kociek

### Podúloha a)

Začnime tým, že si nakreslíme o niečo väčší obrázok, napríklad so 16-timi poschodiami.



Keď sa na obrázok lepšie pozrieme, hneď sa nám vynoria nejaké opakujúce vzorce. Napríklad, vidíme, že celá pyramída výšky 16 sa skladá z troch rovnako vyzerajúcich pyramíd výšky 8 (na obrázku znázornených oranžovou farbou), medzi ktoré je vložená obrátená pyramída z modrých kociek.
Ale takéto pozorovanie vieme spraviť aj pre oranžovú pyramídu – skladá sa z troch rovnakých zelených pyramíd výšky 4, medzi ktorými je obrátená pyramída z modrých kociek. A dokonca aj zelenú pyramídu by sme vedeli ešte ďalej rozdeliť na tri rovnaké pyramídy výšky 2, medzi ktorými sú modré kocky.
Otázkou však je, prečo má pyramídu takúto opakujúcu sa (odborne rekurzívnu) štruktúru, a či sa to bude zachovávať aj naďalej. Teda či sa pyramída výšky 32 bude opäť skladať z troch pyramíd výšky 16, medzi ktorými budú modré kocky.
Všimnime si poschodie 16, ktoré sa skladá zo samých červených kociek. Ako vyzerá poschodie 17? Pod dvoma červenými kockami musí byť kocka modrá, celé toto poschodie s výnimkou krajných dvoch kociek (ktoré sú vždy červené) je teda modré.

Ďalej si uvedomme, že modré kocky nespôsobujú zmenu farby – pod dvoma modrými je opäť modrá, pod modrou a červenou je červená. Takže zmena môže nastať len keď susedia dve červené kocky.
Zoberme si teraz jednu z krajných červených kociek na poschodí 17. Z jednej strany je okraj pyramídy a z druhej modrá kocka, ktorá ako sme si povedali farbu neovplyvňuje. Pod toutou kockou sa teda začne tvoriť rovnaká pyramída, akú sme mali na samom vrchu. Aj tá totiž začína s jednou červenou kockou. Poschodie 18 bude priamo pod touto červenou kockou teda vyzerať rovnako ako druhé poschodie, poschodie 19 ako tretie, atď.
Toto by mohlo pokračovať do nekonečna, problémom však je, že my máme jednu červenú kocku na oboch krajoch poschodia 17. A pod oboma sa začne tvoriť rovnaká pyramída. Problem teda nastane, keď sa tieto dve pyramídy začnú navzájom ovplyvňovať. Na začiatku sú však oddelené modrými kockami – poschodie 17 má 17 kociek, dve krajné sú červené, medzi nimi je teda 15 modrých kociek.
Čo sa stane na poschodí 18? Obe krajné pyramídy sa rozrastú o jedno, budú sa skladať z 2 kociek. A medzi nimi zostanú modré kocky, ktorých bude $18 - 2 \cdot 2 = 14$. Na ďalšom poschodí sa opäť obe pyramídy o 1 zväčšia, poschodie sa však zväčší len o 1 kocku, počet modrých kociek medzi nimi teda opäť klesne o 1, na 13.
Kedy sa teda pyramídy dotknú, aby sa začali ovplyvňovať? Bude to na poschodí $17 + 15 = 32$ – na 17-tom poschodí sú od seba vzdialené o 15 kociek a každé ďalšie poschodie táto vzdialenosť o 1 klesne. Toto poschodie je zároveň 16-te poschodie našich dvoch tvoriacich sa pyramíd a začínali sme predsa tým, že 16-te poschodie je celé tvorené červenými kockami.
Keďže poschodie 32 je celé tvorené dvoma poschodiami pyramíd výšky 16, opäť bude celé z červených kociek. A tam budeme vedieť našu úvahu opäť zopakovať, totiž poschodie 33 bude mať opäť len dve červené kocky na samých krajoch.
Doteraz platilo, že celé červené poschodia sú práve tie, ktorých číslo je mocnina dvojky. Dokážme si, že to bude platiť aj vo zvyšku pyramídy. Nech poschodie $n = 2^k$ je poschodie, o ktorom už vieme, že je celé červené. Potom poschodie $n + 1$ je tvorené dvoma červenými kockami, ktoré sú oddelené $n - 1$ modrými kockami. Pod oboma červenými kockami sa začnú tvoriť pyramídy, ktoré sa prvýkrát stretnú na poschodí $(n + 1) + (n - 1) = 2n$. Poschodie $2n$ je teda tvorené dvoma kópiami poschodia $n$. No a keďže to sa skladalo zo samých červených kociek, aj poschodie $2n$, teda $2^{k+1}$ sa skladá zo samých červených kociek. (A zjavne platí, že každé z poschodí medzi nimi má v strede nejaké modré kocky, a teda nie je celé červené.)

### Podúloha b)

Našou úlohou je zistiť farby niekoľkých konkrétnych kociek. Prvá je kocka 10 na 20. poschodí. Táto úloha sa dá vyriešiť aj tak, že si pyramídu výšky 20 nakreslíme celú (predsa len to nie je tak veľa), pokúsme sa však využiť pozorovanie z predchádzajúcej podúlohy.
Vieme, že poschodie 16 sa skladá zo samých červených kociek a na poschodí 17 sa začnú tvoriť dve rovnaké pyramídy. Poschodie 20 teda vyzerá tak, že na krajoch sú dve kópie poschodia 4 pôvodnej pyramídy oddelené modrými kockami. Poschodie 4 obsahuje iba červené kocky, a preto poschodie 20, ktoré má 20 kociek má najprv 4 červené kocky, následne 12 modrých kociek a na konci opäť 4 červené kocky. Kocka číslo 10 je teda zjavne *modrá*.

Ďalšou v poradí je kocka 266 na 276. poschodí. Ako vyzerá toto poschodie? Najbližšia mocnina dvojky k 276 je $256 = 2^8$, toto poschodie má teda samé červené kocky. A keďže pod celočerveným poschodím sa začnú na krajoch opakovane tvoriť rovnaké pyramídy (ktoré sa stretnú až na dvojnásobnom poschodí), poschodie 276 sa musí skladať z dvoch kópií poschodia 20, ktoré sú od seba oddelené modrými kockami.
Presnejšie, na poschodí 276 zodpovedajú kocky 1 až 20 kockám z 20. poschodia, kocky 21 až 256 sú modré a kocky 257 až 276 opäť zodpovedajú kockám 20. poschodia. Kocka 266 spadá do tretieho intervalu, teda do kópie 20. poschodia. A v tejto kópii (kocky 257 až 276) je kocka 266 desiata v poradí. Kocka 266 na 276. poschodí je teda kópiu kocky 10 na 20. poschodí. Ktorej farbu sme určili v predchádzajúcej úlohe, aj táto kocka je preto *modrá*.

Ostáva kocka 157 na 2023. poschodí. Využime ten istý prístup. Najbližšia menšia mocnina 2 k číslu 2023 je číslo 1024. Poschodie 2023 je teda tvorené dvoma kópiami poschodia $2023 - 1024 = 999$. Kocka 157 je zjavne 157-ma kocka v ľavej kópii poschodia 999.
Chceme teda vedieť, akú farbu má kocka 157 na 999. poschodí. Opäť zistíme najbližšie vyššie poschodie z

červených kociek – poschodie 512. Poschodie 999 je teda tvorené dvoma kópiami poschodí $999 - 512 = 487$ (oddelených modrými kockami), kocka 157 stále patrí do ľavej kópie.
Keďže ďalšia menšia mocnina dvojky je 256, poschodie 487 je tvorené dvoma kópiami poschodia $487 - 256 = 231$, kocka 157 je stále v ľavej kópii. Predtým je červené poschodie 128, poschodie 231 je teda tvorené dvoma kópiami poschodia $231 - 128 = 103$.
A tu to začína byť zaujímavé, lebo sa musíme zamyslieť, kde sa nachádza kocka 157 na poschodí 231. Toto poschodie má 231 kociek, obsahuje dve kópie poschodia 103, tie sú teda oddelené $231 - 2 \cdot 103 = 25$ modrými kockami. Kocky 1 až 103 patria teda prvej kópii, kocky 104 až 128 sú modré a 129 až 231 patria druhej kópii. Kocka 157 patrí do tejto druhej kópie a je $157 - 129 + 1 = 29$ v poradí.
Postupne sme teda zistili, že farba kocky 157 na 2023. poschodí je rovnaká ako farba kocky 29 na 103. poschodí. A môžeme pokračovať. Poschodie 103 je tvorené dvoma kópiami poschodia $103 - 64 = 39$, kocka 29 je v ľavej kópii. Poschodie 39 je tvorené dvoma kópiami poschodia $39 - 32 = 7$, ktoré sú oddelené $39 - 2 \cdot 7 = 25$ modrými kockami. Kocky 1 až 7 patria prvej kópii, kocky 8 až 32 sú modré a kocky 33 až 39 patria druhej kópii. Kocka 29 je zjavne v modrom úseku, a preto je modrá. A keďže sme sa vždy pozerali iba na kópiu kocky z vyššieho poschodia, znamená to, že aj kocka 157 na 2023. poschodí je *modrá*.

### Podúloha c)

Po správnom vyriešení podúlohy b) je už zovšeobecnenie nášho postupu jednoduché. Nech chceme zistiť farbu kocky $k$ na poschodí $p$. Najbližiu mocninu 2 menšiu ako $p$ si označme hodnotou $m$. Vieme, že poschodie $p$ je tvorené dvoma kópiami poschodia $p - m$, ktoré sú oddelené modrými kockami. Ľavá z týchto kópií pokrýva kocky 1 až $p - m$, pravá kocky $p - (p - m) + 1 = m + 1$ až $p$, zvyšné kocky sú modré. Nastáva teda jeden z troch prípadov:

- Ak platí, že $k \leq p - m$ farba tejto kocky je rovnaká ako farba kocky $k$ na poschodí $p - m$, stačí teda určiť tú. Tým sme dostali nový menší problém, ktorý môžeme vyriešiť rovnakým (rekurzívnym) spôsobom.
- Ak platí, že $k \geq m+1$, tak farba kocky, ktorá nás zaujíma, je rovnaká ako farba kocky $k-(m+1)+1 = k-m$ na poschodí $p - m$. To opäť vieme rekurzívne (t.j. opakovaním takejto istej úvahy) vyriešiť.
- Pre všetky ostatné hodnoty $k$ vieme hneď povedať, že naša kocka je modrá.

Ostáva určiť časovú zložitosť takéhoto riešenia. Keďže $m$ je najbližia mocnina 2 menšia ako $p$, tak platí $2m \geq p$, čiže $m \geq p/2$. Hodnota $p - m$ je teda nanajvýš rovná $p/2$. Číslo skúmaného poschodia sa teda po každom prepočte zmenší na aspoň polovicu. No a takýchto zmenšovaní nemôže byť veľa. (Napr. číslo $p = 2^{30}$, čo je niečo vyšše miliardy, vieme takto zmenšiť len nanajvýš 30-krát, kým sa dostaneme na prvé poschodie.)
Maximálny počet prepočtov pre konkrétnu začiatočnú hodnotu $p$ vieme vypočítať ako logaritmus $p$ o základe 2. Výsledná časová zložitosť je teda $O(\log p)$.

Má to ešte jeden drobný háčik. Ako zistiť najbližšiu menšiu mocninu 2? Ak by sme to totiž robili postupným skúšaním (2, 4, 8, 16 ...) trvalo by to $O(\log p)$ pre každé skúmané poschodie a našu časovú zložitosť by sme si mierne pokazili na $O(\log^2 p)$. Namiesto toho si túto hodnotu vypočítame len raz, a potom ju budeme postupne zmenšovať na polovicu kým nebude vyhovovať. Tento postup bude trvať $O(\log p)$ dokopy za celý beh programu.

### Listing programu (Python)

```
def farba(k, p, mocnina):
    if p == 1:
        return 'cervena'
    while mocnina >= p:
        mocnina //= 2
    if k <= p - mocnina:
        return farba(k, p - mocnina, mocnina)
    elif k >= mocnina + 1:
        return farba(k - mocnina, p - mocnina, mocnina)
    else:
        return 'modra'

k, p = map(int, input().split())
mocnina = 1
while mocnina < p:
    mocnina *= 2
print(farba(k, p, mocnina))
```

## B-I-4 Tabuľkový počítač

Pripomeňme si postupnosti, ktoré máme vygenerovať:

a) (2 body) Postupnosť $A$, kde $A_n = 2^n$. Začiatok tejto postupnosti: 1, 2, 4, 8, 16, 32, ...

b) (2 body) Postupnosť $B$, kde $B_0 = -1$, $B_1 = 1$, $B_2 = 4$ a pre $n \geq 3$ platí $B_n = 3 \cdot B_{n-1} + 8 \cdot B_{n-3}$. Začiatok tejto postupnosti: $-1$, 1, 4, 4, 20, 92, 308, ...

c) (3 body) Postupnosť $C$, kde $C_n = 5^n - 4^n$. Začiatok tejto postupnosti: 0, 1, 9, 61, 369, 2101, ...

d) (3 body) Postupnosť $D$, kde $D_0 = 15$, $D_1 = 4$ a pre $n \geq 2$ platí $D_n = D_{n-1} + D_{n-2} + 2n$. Začiatok tejto postupnosti: 15, 4, 23, 33, 64, 107, 183, ...

### Podúloha A: mocniny dvoch

Na túto postupnosť nám stačí v podstate najjednoduchší možný tabuľkový počítač: taký s $k = 1$.
Pásiky, ktoré bude spracúvať, budú mať teda len jedno políčko. Na začiatku na tomto políčku musí byť zapísaná začiatočná hodnota našej postupnosti, teda číslo 1.
V tabuľkovom počítači budeme mať len jeden násobič. Toho programom bude pásik s jedným políčkom, na ktorom bude napísané číslo 2. Tento násobič zjavne svoj jediný vstup vynásobí dvoma a výsledok dá na výstup.
Zhrnutie: začiatočný pásik bude (1) a jediný násobič bude (2).

### Podúloha B: rekurentná postupnosť

Pri Fibonacciho postupnosti platilo, že keď poznáme dve po sebe idúce hodnoty, vieme z nich vypočítať nasledujúcu. Naša postupnosť $B$ sa správa podobne, ale keďže pri výpočte $B_n$ používame až hodnotu $B_{n-3}$, potrebujeme až tri po sebe idúce hodnoty.
Zvolíme si teda $k = 3$, čiže budeme pracovať s pásikmi, ktoré majú tri políčka.
Na začiatočný pásik napíšeme prvé tri hodnoty postupnosti: $(B_0, B_1, B_2)$, teda $(-1, 1, 4)$.
Teraz si ešte potrebujeme rozmyslieť, ako presne budú vyzerať jednotlivé násobiče v tabuľkovom počítači. Chceli by sme dosiahnuť, aby po vložení ľubovoľného pásika s hodnotami $(B_i, B_{i+1}, B_{i+2})$ vyšiel na výstupe pásik s hodnotami $(B_{i+1}, B_{i+2}, B_{i+3})$.
Toto dosiahneme nasledovne:

- Prvý násobič bude mať program $(0, 1, 0)$. Na prvé políčko výstupu sa tak dostane hodnota z druhého políčka vstupu, čo je presne to, čo chceme.
- Podobne, druhý násobič bude mať program $(0, 0, 1)$.
- Posledný, tretí násobič potrebuje z hodnôt $(B_i, B_{i+1}, B_{i+2})$, ktoré dostane na vstupe, vypočítať hodnotu $B_{i+3}$. Na to použijeme práve rekurentnú definíciu tejto postupnosti. Z tej vieme, že $B_{i+3} = 8 \cdot B_i + 0 \cdot B_{i+1} + 3 \cdot B_{i+2}$, a teda násobič s programom $(8, 0, 3)$ spraví presne to, čo potrebujeme.

### Podúloha C: rozdiel dvoch mocnín

V podúlohe A sme už vymysleli, ako generovať jednu exponenciálne rastúcu postupnosť. Ak si zoberieme dlhší pásik papiera, vieme takýchto postupností samozrejme naraz generovať ľubovoľne veľa.
Všimnime si napríklad, čo sa stane, ak pre $k = 3$ a začiatočný pásik $(0, 1, 1)$ zoberieme tabuľkový počítač, v ktorom druhý násobič bude $(0, 5, 0)$ a tretí zase $(0, 0, 4)$. Ľahko nahliadneme, že bez ohľadu na to, čo robí prvý násobič, budeme na druhom políčku pásiku postupne generovať mocniny piatich a na treťom políčku mocniny štyroch. Na výstupe sa teda postupne objavia pásiky $(?, 5, 4)$, $(?, 5^2, 4^2)$, $(?, 5^3, 4^3)$, atď.
Zatiaľ sme si nechali nepoužité prvé políčko pásika – teda to, na ktorom sa majú postupne objavovať prvky našej postupnosti. Zjavne by sme chceli, aby na každom pásiku toto políčko obsahovalo rozdiel jeho druhého a tretieho políčka.
To ale vieme ľahko dosiahnuť. Ak máme ľubovoľný vstup $(?, x, y)$, vieme, že druhé a tretie políčko výstupu budú $5x$ a $4y$, prvé políčko výstupu teda má byť $5x - 4y$.

Ako prvý násobič preto použijeme násobič s programom $(0, 5, -4)$ a sme hotoví.

Trochu iné riešenie: začneme už s pásikom $(0, 5, 4)$ a prvý násobič dostane program $(0, 1, -1)$. Pre tento tabuľkový počítač potom platí, že po $i$ iteráciách budeme mať na výstupnom pásiku postupne hodnoty $(5^i - 4^i,\ 5^{i+1},\ 4^{i+1})$.

### Podúloha D: rekurencia s lineárnym členom

Na záver sme si nechali jeden trošku ťažší oriešok.
Začnime tým, že si rozmyslíme, ako vygenerovať samostatnú postupnosť $2n$.
Na to nám stačia dve políčka. Trik je v tom, že na jednom z nich si budeme udržiavať konštantu: hodnotu 1 alebo 2 (obe možnosti fungujú).
Konkrétne to môže vyzerať napr. nasledovne: Prvý vstupný pásik bude $(0, 2)$. Druhý násobič bude $(0, 1)$, takže aj všetky výstupné pásiky budú mať na druhom políčku hodnotu 2. No a prvý násobič bude $(1, 1)$. Ten zoberie aktuálnu hodnotu z prvého políčka a pripočíta k nej hodnotu z druhého políčka. A keďže vieme, že tam máme dvojku, na výstupe dostaneme číslo o 2 väčšie ako bolo na vstupe. Po $i$ iteráciách budú teda na výstupnom pásiku čísla $(2i, 2)$.

Teraz sa už môžeme pustiť do generovania celej postupnosti $D$.
Podobne ako u Fibonacciho si budeme potrebovať pamätať dve po sebe idúce hodnoty (na čo treba dve políčka) a k nim navyše budeme chcieť vygenerovať aj hodnotu $2n$ (na čo treba ďalšie dve políčka). Zvolíme si teda $k = 4$.
Budeme chcieť, aby sme na pásiku po $i$ iteráciách mali postupne zapísané nasledujúce hodnoty: $(D_i, D_{i+1}, 2i, 2)$.
Začiatočný vstupný pásik teda bude mať na sebe hodnoty $(D_0, D_1, 2 \cdot 0, 2) = (15, 4, 0, 2)$.
Pri konštrukcií programov pre jednotlivé násobiče začnime od tých jednoduchších:

- Aby sme na poslednom políčku mali stále dvojku, použijeme štvrtý násobič s programom $(0, 0, 0, 1)$.
- S tretím políčkom spravíme to isté, čo vyššie: tretí násobič bude $(0, 0, 1, 1)$, čím dosiahneme, že v každej iterácii sa obsah tohto políčka zväčší o dva.
- Prvý násobič bude $(0, 1, 0, 0)$, teda „posunieme“ hodnotu z druhého vstupu na prvý výstup.
- No a jediné netriviálne už je len rozmyslieť si, ako presne bude vyzerať druhý násobič. Ten má zo vstupov $(D_i, D_{i+1}, 2i, 2)$ vypočítať hodnotu $D_{i+2} = D_{i+1} + D_i + 2(i + 2)$. Lahko nahliadneme, že presne tento výpočet spraví násobič s programom $(1, 1, 1, 2)$.

**TRIDSIATY DEVIATY ROČNÍK OLYMPIÁDY V INFORMATIKE**

Príprava úloh: Michal Anderle, Truc Lam Bui, Andrej Korman, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2023