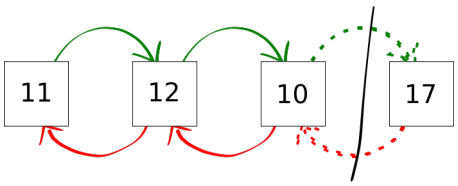
B-II-1 Kopa kníh 2

Základné riešenie

Kopu reprezentujeme ako dvojsmerný spájaný zoznam. Každá položka zoznamu si pamätá tri veci: ukazovateľ na predchádzajúcu položku (alebo špeciálnu hodnotu NULL ak predchádzajúca položka neexistuje), ukazovateľ na nasledujúcu položku (alebo NULL ak neexistuje) a číslo knihy. Vrch kopy bude zodpovedať poslednej položke v zozname; pre každú kopu si zvlášť pamätáme ukazovateľ na túto poslednú položku (alebo NULL, ak je kopa prázdna). Jednotlivé udalosti implementujeme nasledovne:

1. Pridanie knihy na vrch kopy: vytvoríme novú položku a reprezentujúcu túto knihu a zapojíme ju za položku b, ktorá je momentálne na konci zoznamu. To znamená len toľko, že položke a nastavíme nasledovníka na b a položke b nastavíme predchodcu na a.
2. Odobranie knihy z vrchu kopy: ak posledná položka $b = \text{NULL}$, tak kopa je prázdna a len povieme -1. V opačnom prípade sa novým vrchom kopy stane predchodca b; vrátime číslo knihy v b.
3. Presunutie vrchných n kníh na novú kopu: iba n-krát za sebou odoberieme knihu z vrchu kopy, a tieto knihy potom postupne (v ďalších n krokoch, v správnom poradí) vložíme na novú kopu.



(Obrázok 1: Pridanie knihy na vrch, resp. odobratie z vrchu. Zelené šípky sú ukazovatele na nasledovníkov, červené šípky sú ukazovatele na predchodcov.)

Všetky kopy si budeme pamätať v poli veľkosti $10^6 + 1$. (Zadanie zaručuje, že ich identifikačné čísla nepresiahnu 10^6, takže toto stačí.) Na začiatku všetky kopy inicializujeme na prázdne.

Časová zložitosť udalostí typov 1 a 2 je $O(1)$, pre udalosť typu 3 je to $O(n)$. Na reprezentovanie kopy veľkosti m potrebujeme $O(m)$ pamäte (m položiek a každá z nich je konštantnej veľkosti). Celková pamäťová zložitosť je teda $O(q)$, kde q je počet udalostí – každá udalosť totiž zvýši počet kníh v knižnici nanajvýš o 1, takže celkový počet kníh nikdy neprekročí q.

V elektronickej verzii riešení tu bude listing programu, v tlačenej ho pre úsporu miestom vynecháme.

Toto riešenie mohlo získať 4 body: rýchlosť udalostí typu 3 závisí od n, a je dostatočne rýchla iba pre malé n.

Plné riešenie

Predchádzajúce riešenie bolo pomalé, lebo sme pre udalosti typu 3 museli nájsť n-tu položku od konca, a to sme (s informáciami, ktoré sme si pamätali) vedeli spraviť len v $O(n)$ čase. Pomôže nám, ak si pre každú položku budeme pamätať nový ukazovateľ: na položku, ktorá je v zozname o n skôr.
Potrebujeme ukázať nasledovné veci:

1. Že naďalej vieme udalosti typu 1 a 2 simulovať v konštantnom čase,
2. že s touto informáciou vieme riešiť aj udalosti typu 3,
3. a že si túto novú informáciu vieme efektívne udržiavať.

Pre zjednodušenie terminológie budeme volať položku o x predtým ako *x-predchodca*. Pozrime sa teraz na to, ako po novom implementovať naše tri typy udalostí.

Odkopnutie vrchných n kníh z vrchu kopy. Potrebujeme spraviť dve veci, jednak odobrať n kníh z vrchu kopy, a dvak založiť novú kopu s týmito knihami.

1. Po odobraní n vrchných kníh sa novým vrchom kopy stane n-predchodca aktuálnej poslednej položky. Označme ho v. Nad v už v pôvodnej kope nič nebude, tak tu spájaný zoznam rozpojíme. Nasledovníkovi v nastavíme predchodcu na NULL (on bude prvou položkou v zozname pre novú kopu) a položke v zase nastavíme nasledovníka na NULL.

2. Po rozpojení máme dve spájané zoznamy. Jeden z nich je pôvodná kopa, druhý z nich obsahuje knihy na novej kope. Avšak nestačí nám len nastaviť tento druhý spájaný zoznam ako novú kopu – problémom je, že knihy v jeho vnútri si pamätajú nesprávnych n-predchodcov. Totiž nová kopa je príliš malá na to, aby akákoľvek kniha v nej mala n-predchodcu. Nastaviť ich všetkých na NULL si ale nemôžeme dovoliť, lebo by to trvalo až $O(n)$ času.

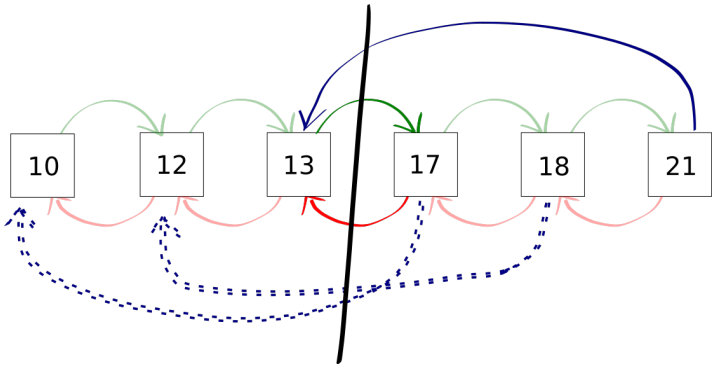
   Toto vyriešime tak, že si budeme udržiavať veľkosť každej kopy. Ak máme kopu veľkosti do n, kopnutie ju posunie celú. A ak máme väčšiu kopu, tak jej posledná položka naozaj má n-predchodcu, a teda sa môžeme pozrieť, kam ukazuje na príslušný ukazovateľ pre poslednú položku na tejto kope.

   Udržiavať veľkosť kopy je ľahké, keďže tá sa mení predvídateľne:

   (a) pridanie knihy: $+1$

   (b) odobranie knihy: -1

   (c) kopnutie zmenší kopu o $v = \min(n, \text{veľkosť kopy})$ a vyrobíme novú kopu s veľkosťou v



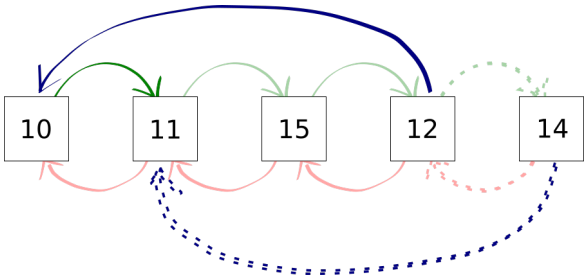
(Obrázok 2: Posunutie vrchných n kníh na novú kopu pre $n = 3$. Modré šípky sú ukazovatele na n-predchodcov. Po posunutí by mali byť bodkované modré šípky preč, v skutočnosti ich len budeme ignorovať.)

Nová kniha na vrch kopy. Pridáme na koniec spájaného zoznamu novú položku. Potrebujeme ešte určit jeho n-predchodcu.

- Ak je po pridaní knihy kopa veľkosti nanajvýš n, tak položka nemá n-predchodcu.
- Ak má nová kopa veľkosť presne $n + 1$, tak n-predchodcom je prvá položka spájaného zoznamu.
- Ak máme ešte väčšiu kopu, uvažujme nasledovne. Kopa je dosť veľká na to, aby doteraz posledná položka mala svojho n-predchodcu. Označme ho P. Položka P je potom zjavne $(n + 1)$-predchodcom našej novej položky. A kto je teda n-predchodcom našej novej položky? Predsa nasledovník položky P!

  Inými slovami, pre veľké kopy je n-predchodcom novej položky vždy nasledovník n-predchodcu predchádzajúcej položky.

Pre ujasnenie ilustrujeme posledný prípad na obrázku:



(Obrázok 3: Príklad pridania knihy na vrch v prípade, že kopa je po pridaní väčšia ako $n+1$. V príklade platí $n=3$. Bodkované šípky sú ukazovatele, ktoré pribudnú po udalosti. Hrubšie sú zvýraznené dva ukazovatele, na ktoré sa pozrieme: od 12 ku 10 a od 10 ku 11.)

Táto implementácia udalosti typu 1 vyžaduje, aby sme si pre každú kopu pamätali tiež jej prvú položku, ak existuje. To ale nie je problém: prvá položka kopy sa mení jedine vtedy, keď:

1. Sa kopa stane prázdnou – vtedy prvá položka prestane existovať.
2. Vznikne nová kopa pridaním jednej knihy (triviálne).
3. Vznikne nová kopa posunutím n kníh z inej kopy. Už sme si popísali, ako vtedy zoznam na správnom mieste rozpojiť, no a prvá položka v pravej polovici sa stane prvou knihou novej kopy.

Odoberanie knihy z vrchu kopy. Ak je kopa prázdna, tak len povieme -1 a nič sa nemení. Ak je neprázdna, tak len z konca odoberieme poslednú položku (a jej predchodca sa stane novým koncom).

Časová zložitosť každého typu udalosti je $O(1)$. Čo sa pamäťovej zložitosti týka, oproti základnému riešeniu si pre každú položku si pamätáme o konštantne veľa údajov navyše (n-predchodca), a pre každú kopu si pamätáme o konštantne veľa údajov navyše (veľkosť kopy, prvá položka). Reprezentácia kopy v pamäti je teda stále priamo úmerná veľkosti kopy, a teda celková pamäťová zložitosť je stále $O(q)$ (kde q je počet udalostí).

Listing programu (C++)

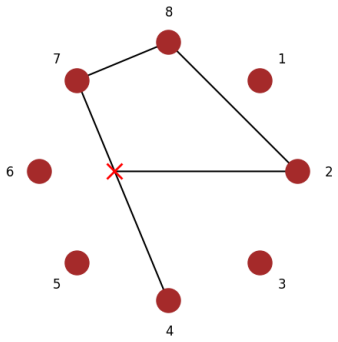
```
#include <iostream>
#include <memory>
#include <cassert>
using namespace std;

struct Polozka
{
  int kniha;
  shared_ptr<Polozka> predchodca = {}, nasledovnik = {}, kPredchodca = {};

  Polozka(int kniha0) : kniha(kniha0) {}
};

struct Kopa
{
  shared_ptr<Polozka> prvaPolozka = {}, poslednaPolozka = {};
  int velkostKopy = 0;

  void pridajKnihu(int kniha, int k)
  {
    shared_ptr<Polozka> novaPolozka(new Polozka(kniha));
    if (poslednaPolozka == nullptr)
    {
      prvaPolozka = novaPolozka;
      poslednaPolozka = novaPolozka;
      velkostKopy = 1;
    }
    else
    {
      poslednaPolozka->nasledovnik = novaPolozka;
      novaPolozka->predchodca = poslednaPolozka;

      velkostKopy += 1;
      if (velkostKopy == k + 1)
      {
        novaPolozka->kPredchodca = prvaPolozka;
      }
```

```
      else if (velkostKopy > k + 1)
      {
        novaPolozka->kPredchodca = poslednaPolozka->kPredchodca->nasledovnik;
      }

      poslednaPolozka = novaPolozka;
    }
  }

  int odoberKnihu()
  {
    if (poslednaPolozka == nullptr)
    {
      return -1;
    }

    if (velkostKopy == 1)
    {
      int vysl = poslednaPolozka->kniha;
      prvaPolozka = nullptr;
      poslednaPolozka = nullptr;
      velkostKopy = 0;
      return vysl;
    }

    int vysl = poslednaPolozka->kniha;
    auto predposlednaPolozka = poslednaPolozka->predchodca;

    // odpojime vrch od spodku
    poslednaPolozka->predchodca = nullptr;
    predposlednaPolozka->nasledovnik = nullptr;

    poslednaPolozka = predposlednaPolozka;
    velkostKopy -= 1;
    return vysl;
  }

  Kopa kopni(int k)
  {
    if (velkostKopy <= k)
    {
      // cela kopa sa presunula
      Kopa vysl = *this;
      *this = {};
      return vysl;
    }

    auto novyVrch = poslednaPolozka->kPredchodca;
    assert(novyVrch != nullptr && "nemozne, lebo kopa je velka aspon k+1");

    auto spodokNovejKopy = novyVrch->nasledovnik;
    novyVrch->nasledovnik = nullptr;
    spodokNovejKopy->predchodca = nullptr;

    Kopa vysl;
    vysl.prvaPolozka = spodokNovejKopy;
    vysl.poslednaPolozka = poslednaPolozka;
    vysl.velkostKopy = k;

    poslednaPolozka = novyVrch;
    velkostKopy -= k;
    return vysl;
  }
};

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

  Kopa kopy[100023] = {};

  while (true)
  {
    int typUdalosti;
    cin >> typUdalosti;
    if (typUdalosti == 0)
    {
      break;
    }

    switch (typUdalosti)
    {
    case 1:
    {
      int kniha, cisloKopy;
      cin >> kniha >> cisloKopy;
      kopy[cisloKopy].pridajKnihu(kniha, k);
      break;
```

```
    }
    case 2:
    {
      int cisloKopy;
      cin >> cisloKopy;
      cout << kopy[cisloKopy].odoberKnihu() << "\n";
      break;
    }
    case 3:
    {
      int kopniKam, novaKopa;
      cin >> kopniKam >> novaKopa;
      assert(kopy[novaKopa].posledna Polozka == nullptr && "cislo novej kopy je uz pouzite");
      kopy[novaKopa] = kopy[kopniKam].kopni(k);
      break;
    }
    default:
    {
      assert(false && "neznamy typ udalosti");
    }
    }
  }

  return 0;
}
```

B-II-2 Umelecká záhrada

V ľahšej verzii úlohy za 8 bodov vieme, že Peťo musí svojou šnúrou pospájať úplne všetky koly. Pozrime sa na príklad zo zadania. Všimnime si, že keď Peťo spojil prvé dva koly, tak rozdelil záhradku na dve neprázdne časti. Následne, hneď ďalším úsekom plotu určil, do ktorej časti sa vyberie, a ktorú naopak *odplotí*. Na obrázku takto *odplotil* koly 5 a 6. No a to je problém, lebo k týmto kolom sa už bez križovania šnúr nikdy nedostane.



Otázkou teda je ako takúto situáciu spoznať keď postupne naťahujeme šnúru. Potrebujeme zakaždým overiť, že v *odplotenej* časti už nie sú žiadne nepoužité koly. To ale znamená, že každý nový pripojený kôl musí susediť s aspoň jedným už pripojeným, inak by určite aspoň jeden jeho sused ostal *odplotený*. Výsledkom bude, že pospájané koly budú celý čas tvoriť súvislý úsek obvodu záhrady. Nám teda stačí, aby sme si tento úsek (respektíve jeho konce) pamätali. Na začiatku ide len o kôl, od ktorého začíname. Pre každý ďalší kôl overíme, či sa nachádza vedľa jedného z krajných kolov nášho úseku, a následne sa práve spracovaný kôl stane novým krajným kolom.

Toto riešenie si celý čas pamätá len dva koly: začiatok a koniec úseku kolov, cez ktoré už vedie šnúra. To nám dáva pamäťovú zložitosť $\mathcal{O}(1)$. Náš program pri každom ďalšom kole z postupnosti overí, či susedí s dvoma koncami nášho intervalu. To vieme spraviť v konštantnom čase, čo znamená, že celková časová zložitosť bude lineárna od počtu kolov, teda $\mathcal{O}(n)$.

Listing programu (Python)

```
N, K = [ int(_) for _ in input().split() ]
A = [ int(_)-1 for _ in input().split() ] # koly cislujeme od nuly

zac, kon = A[0], A[0] # usek uz navstivenych kolov

for kol in A[1:]:
    if kol == (zac-1) % N: # dalsi kol je tesne pred usekom
        zac = kol
    elif kol == (kon+1) % N: # tesne za usekom
        kon = kol
    else: # hocikde inde
        print('umeNIE')
        exit()

print('umeANO')
```

Všeobecné riešenie

Ak je počet použitých kolov k menší ako n, vyššie popísané riešenie už nemôžeme použiť. *Odplotené* koly už nie sú problém, pretože sa môže stať, že sa k nim Peťo nikdy nebude chcieť dostať.
Stále však môžeme využiť pozorovanie, ktoré sme učinili v predošlej podúlohe, len trochu ináč. Predstavme si, že Peťo práve natiahol prvý úsek šnúry a rozdelil tak nepoužité koly na dva úseky. Keď teraz začne naťahovať ďalší úsek, bude jednoznačne určené, ktoré z nich *odplotí* – a teda aj to, ktoré ostanú *dosiahnuteľné*. Dosiahnuteľné koly zjavne naďalej tvoria jeden súvislý úsek kolov. No a keď Peťo donaťahuje aktuálny úsek šnúry a začne naťahovať ďalší, opäť sa dostaneme do stavu, v ktorom sme už boli – úsek doteraz dosiahnuteľných kolov sme rozdelili na dve časti a potom jedna z nich prestala byť dosiahnuteľná, čím vznikol nový (kratší, ale stále súvislý) úsek dosiahnuteľných kolov.

V programe riešiacom všeobecnú verziu úlohy si teda opäť stačí pamätať dva koly: začiatok a koniec súvislého úseku kolov, ktoré ešte vieme dosiahnuť šnúrou. Pre každý nový kôl zistíme, či v tomto úseku leží, a ak áno, natiahneme k nemu šnúru a potom sa pozrieme, ktorý z dvoch podúsekov kolov, ktoré takto vznikli, ostane dosiahnuteľný.

Pamäťová zložitosť ostáva rovnaká ako v predchádzajúcom riešení: $\mathcal{O}(1)$, teda konštantná. Časová sa zmení zo závislosti od celkového počtu kolov len na závislosť na počte navštívených kolov a teda bude $\mathcal{O}(k)$. Na záver si môžeme všimnúť, že naše riešenie už z hľadiska zložitostí vylepšiť nejde, keďže pamätať si toho menej ako konštantný počet premenných nevieme a zároveň časovú zložitosť nemáme ako znížiť, keďže inak by sme nenačítali celý vstup (v ktorom aj posledný kôl môže zmeniť odpoveď).

Implementácia

Úsek kolov si v programe budeme pamätať ako dvojicu čísel (z, k). Táto dvojica predstavuje úsek, ktorý začína kolom z a ide v smere ručičiek až po kôl k vrátane. Pritom bude platiť, že na koloch z a k už je špagát, zatiaľ čo na koloch medzi nimi ešte nie je.
V programe pre pohodlie načítame celé pole čísel kolov do pamäte. Ak by sme chceli pamäťou skutočne šetriť, stačilo by ich čítať po jednom.

Listing programu (Python)

```
def lezi_v_useku(z, k, x): # lezi x vo vnutri useku (z,k)?
    if k > z:
        return z < x < k
    else:
        return (z < x) or (x < k)


N, K = [ int(_) for _ in input().split() ]
A = [ int(_) for _ in input().split() ]

# usek dosiahnutelnych kolov su na zaciatku vsetky koly
zac, kon = A[0], A[0]

for i in range(1, K):
    # ak ideme mimo dosiahnutelneho useku, je to zle, inak vieme natiahnut dalsi kus snury
    if not lezi_v_useku(zac, kon, A[i]):
        print('umeNIE')
        exit()
    # ak uz sme spracovali cely vstup, je vsetko korektne ...
    if i+1 == K:
        print('umeANO')
        exit()
    # ... a inak podla nasledujuceho smeru snury zistime, ktory podusek ostane dosiahnutelny
    if lezi_v_useku(zac, A[i], A[i+1]):
        kon = A[i]
    else:
        zac = A[i]
```

B-II-3 Sklenené guľôčky

Aj keď sa nám nepodarí vymyslieť vzorové riešenie, vždy je dobré získať čiastočné body za hocičo pomalé, čo dáva správnu odpoveď.

V tejto úlohe, priamočiare pomalé riešenie je postupne vložiť do spektskloskopu všetky dvojice guľôčok. Pre tie dvojice, ktoré majú rovnakú farbu, dostaneme odpoveď 1, pre ostatné dostaneme odpoveď 2.
Takéto riešenie sa pre $2n$ guľôčok spýta $\frac{2n(2n-1)}{2} = n(2n-1)$ otázok. Pre $n = 1000$ je to o trochu menej ako 2 000 000.

Toto riešenie sa dá pomerne jednoducho vylepšiť. Ak už zistíme, že guľôčka 1 a 47 majú rovnakú farbu, nemusíme sa viac pýtať na dvojice obsahujúce 1 ani 47. Vylepšený algoritmus môže fungovať tak, že v každom momente bude hľadať pár k nespárovanej guľôčke s najmenším číslom, a to tak, že ju skúsi spárovať postupne s každou inou.
Ešte jedno drobné zlepšenie je, že ak sme už vložili guľôčku a do spektskloskopu s každou nespárovanou okrem jednej, poslednú dvojicu už nemusíme skúšať, lebo vieme, že prístroj odpovie „1“.

Ak máme $2n$ guľôčok, potrebujeme $2n-2$ otázok, aby sme zaručene našli dvojicu. Ostane nám $2n-2$ guľôčok. Na nájdenie i-teho páru teda potrebujeme $2n-2i$ použití spektskloskopu.
Dokopy je to $\sum_{i=1}^{n}(2n-2i) = n(n-1)$, resp. 999 000 pre $n = 1000$.

Vzorové riešenie

Ukážeme si spôsob, ktorým šikovnejšie (na približne $2\log_2 n$ otázok) nájdeme jednu dvojicu guľočok. Využijeme to, že do spektskloskopu môžeme vložiť naraz aj viac než dve guľôčky.
Predstavme si, že vložíme do prístroja guľôčky 1 až 1000, a potom 2 až 1000. Nech `a = odmeraj(1..1000)` a `b = odmeraj(2..1000)`. Pokiaľ `a == b+1`, tak vieme, že guľôčka 1 nemá rovnakú farbu ako žiadna z guľôčok 2..1000. Ak `a == b`, tak vieme, že guľôčka 1 má rovnakú farbu ako niektorá z guľôčok 2..1000. Žiadna iná možnosť nemôže nastať. (Rozmyslite si, prečo to platí.)

Pomocou dvoch otázok sme práve zistili niečo, na čo by sme ich predtým v najhoršom prípade potrebovali až 999. Zovšeobecnením tohto triku dokážeme postupne zmenšovať okruh kandidátov na pár k prvej guľôčke: rovnako, ako sme ho práve zmenšili z $2n-1$ na nanajvýš n ho v ďalšom kroku vieme zmenšiť na nanajvýš $\lceil \frac{n}{2} \rceil$, potom na $\lceil \frac{n}{4} \rceil$, a tak ďalej až kým nám neostane len jeden možný kandidát.
Na nájdenie prvého páru teda potrebujeme nanajvýš $2\lceil \log_2(2n) \rceil$ otázok. Dokopy na nájdenie n dvojíc potrebujeme určite menej ako n-násobok tohto čísla, čiže menej ako $2n\lceil \log_2(2n) \rceil$ otázok. Pre $n = 1000$ to je výrazne pod 25 000.
Ak by sme chceli presnejší odhad počtu otázok pre toto riešenie, mohli by sme spočítať súčet 1000 logaritmov postupne sa zmenšujúceho počtu guľôčok a dostali by sme, že otázok dokonca položíme zaručene menej ako 20 000.

Listing programu (Python)

```
def najdi_par(x, moznosti):
    # pre maly pocet gulocok sa neoplati delit na polovicu
    if len(moznosti) < 4:
        if len(moznosti) == 1 or odmeraj([x, moznosti[0]]) == 1:
            return moznosti[0]
        if len(moznosti) == 2 or odmeraj([x, moznosti[1]]) == 1:
            return moznosti[1]
        return moznosti[2]

    l_polovica = moznosti[: len(moznosti) // 2]
    r_polovica = moznosti[len(moznosti) // 2 :]
    if odmeraj([x] + l_polovica) > odmeraj(l_polovica):
        return najdi_par(x, r_polovica)
    else:
        return najdi_par(x, l_polovica)

n = 1000  # resp. int(input())
pary = [None for _ in range(2 * n)]
nesparovane = list(range(2 * n))  # cislujeme od 0
for _ in range(n):
    x = nesparovane[0]
    y = najdi_par(x, nesparovane[1:])
    pary[x] = y
    pary[y] = x
    nesparovane.remove(x)
    nesparovane.remove(y)

vypis_odpoved(pary)
```

Upozornenie na nesprávnu úvahu

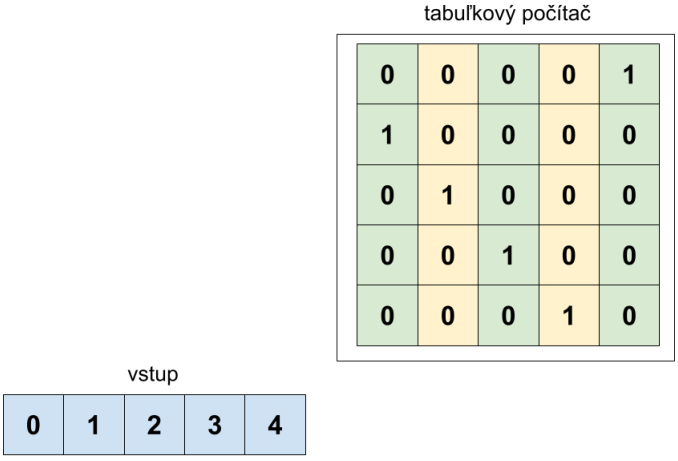
Ak na poli A funkcia `odmeraj` vráti hodnotu menšiu ako dĺžka poľa A, môžeme si byť istí, že toto pole obsahuje nejakú dvojicu. Potiaľto je táto úvaha v poriadku, môžeme však teraz ľahko podľahnúť pokušeniu skúsiť ľubovoľnú jednu dvojicu nájsť tak, že rozdelíme pole A na dve zhruba rovnako veľké časti B a C, zavoláme `odmeraj(B)` a potom budeme tvrdiť nasledovnú vec: ak dostaneme na výstupe hodnotu menšiu ako dĺžka B, pokračujeme rekurzívne ďalej s poľom B a inak pokračujeme rekurzívne ďalej s poľom C, a tak dokola až kým nám neostane len jedna k sebe patriaca dvojica guľôčok a nič iné.
V čom je problém s touto úvahou? V tom, že môže nastať aj tretí prípad, na ktorý sme zabudli: môže sa stať, že aj samotné B aj samotné C obsahujú navzájom rôzne farby, lebo keď sme pole A rozdelili, tak z každej dvojice, ktorú obsahovalo, jedna guľôčka skončila v B a druhá v C.

B-II-4 Tabuľkový počítač 2

Podúloha a): oscilátor

Našou úlohou je vytvoriť postupnosť A, ktorej prvky sú postupne: $0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1 \ldots$

V postupnosti A sa cyklicky opakujú (oscilujú) hodnoty 0 až 4. V klasickom programovacom jazyku by niečo také veľmi dobre zachytávalo modulo 5 (zvyšok po delení). My však máme prístup iba k sčítaniu a násobeniu, ktorými modulo simulovať nevieme.
Spomeňme si, že v domácom kole sme veľmi často využívali konštrukciu, v ktorej sme posúvali hodnoty na vstupnom pásiku doľava. Na začiatku sme na vstupný pásik napísali niekoľko prvých hodnôt, z ktorých sme potom vypočítali tú ďalšiu. Z pásika $(A_0, A_1, A_2, A_3, A_4)$ sme jedným prechodom cez tabuľkový počítač dostali pásik $(A_1, A_2, A_3, A_4, A_5)$. No a ak zabezpečíme, že hodnota $A_5 = A_0$ máme vlastne naše riešenie.
To veľmi ľahko docielime, keď do piateho stĺpca tabuľkového počítača dáme program $(1, 0, 0, 0, 0)$. Takýto program totiž zo vstupného pásiku získa práve hodnotu na prvej pozícii a keďže je tento program v piatom stĺpci, zapíše ho na piatu pozíciu výstupu. Celý tabuľkový program aj so vstupným pásikom vyzerá nasledovne:



Podúloha b): striedavé mocniny

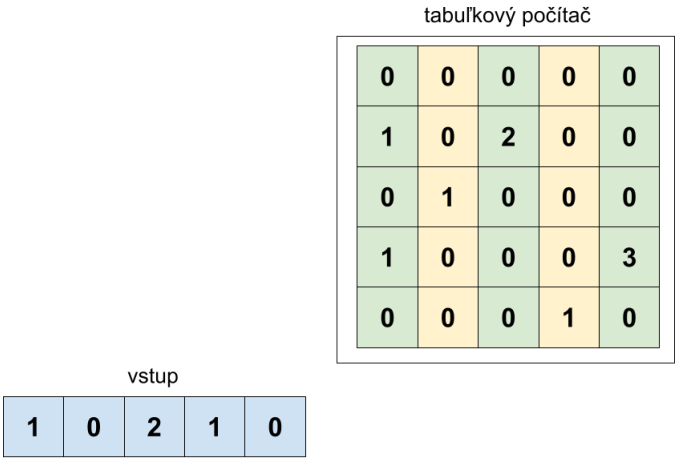
Máme vytvoriť postupnosť B, ktorej prvky sú postupne: $2^0, 3^0, 2^1, 3^1, 2^2, 3^2, 2^3 \ldots$

Tabuľkový počítač pre generovanie mocnín 2 sme vytvorili už v domácom kole, kde celý program tvorilo jedno políčko s číslom 2. Všimnime si však, že v postupnosti B potrebujeme zväčšiť mocninu iba každé druhé kolo.

Zachovajme preto program, ktorý políčko na vstupnej páske vynásobí 2, do tohto políčka však budeme na striedačku vkladať hodnoty 0 a 2^x.
To vieme robiť princípom z predchádzajúcej podúlohy – oscilovaním. Vstupná páska bude mať dve políčka, ktoré budú nadobúdať nasledovné hodnoty: $(2^x, 0) \to (0, 2^{x+1}) \to (2^{x+1}, 0)$.
Na to využijeme dva násobiče. Prvý presunie číslo z druhej pozície na prvú – program $(0, 1)$. Druhý na druhú pozíciu zapíše dvojnásobok hodnoty z prvej pozície – program $(2, 0)$.

Mocniny čísla 3, ktoré sa zväčšujú každé druhé kolo vieme robiť tým istým princípom na ďalších dvoch políčkach vstupnej pásky.

Ostáva už len pridať prvé výstupné políčko, do ktorého sa vždy zapíše správna hodnota. Naša vstupná páska bude vyzerať a meniť sa nasledovne: $(\text{výstup}, 2^x, 0, 0, 3^x) \to (\text{výstup}, 0, 2^{x+1}, 3^x, 0) \to (\text{výstup}, 2^{x+1}, 0, 0, 3^{x+1})$.
Všimnite si, ako sme rozsynchronizovali druhé a tretie mocniny tak, aby sa v každom kole zväčšila iba jedna z nich. Už ľahko nahliadneme, že prvé políčko, kde generujeme výstup, vždy vypočítame ako súčet druhého a štvrtého políčka – buď to bude (aktuálna mocnina 2) plus 0, alebo 0 plus (aktuálna mocnina 3).



Podúloha c): Prefixové súčty

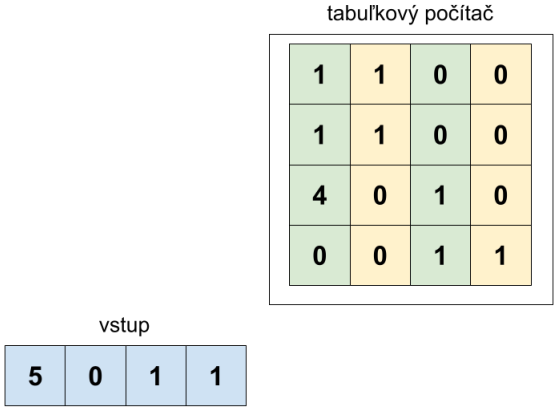
Postupnosť C, kde $C_0 = 5$ a pre $n \geq 1$ platí $C_n = 4n + C_{n-1} + C_{n-2} + \cdots + C_1 + C_0$

Už v domácom kole sme sa stretli s postupnosťami, ktoré záviseli od predchádzajúcich prvkov. Riešením bolo pamätať si tieto predchádzajúce prvky na vstupnej páske a v každej iterácii z nich vypočítať novú hodnotu.
Rozdielom však bolo, že v domácom kole bol počet predchádzajúcich prvkov vždy konštantný. Tu tento princíp využiť nevieme, pretože nemôžeme mať ľubovoľne veľkú vstupnú pásku, ktorá by obsahovala všetky predchádzajúce prvky.
Pozrime sa na dve za sebou idúce prvky:

$$C_n = 4n + C_{n-1} + C_{n-2} + \cdots + C_1 + C_0$$

$$C_{n+1} = 4(n+1) + C_n + C_{n-1} + C_{n-2} + \cdots + C_1 + C_0$$

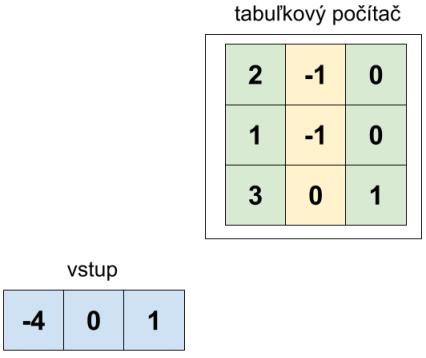
Všimnime si, že väčšina týchto dvoch súčtov je rovnaká, jediné čo sa zmenilo je lineárny prvok $4(n+1)$ a pridanie hodnoty C_n, ktorú sme mali vypočítanú v predchádzajúcom kole. Označme súčet prvých n prvkov postupnosti C ako P_n – tieto hodnoty sa zvyknú volať *prefixové súčty*. Potom $C_n = 4n + C_{n-1} + P_{n-2}$. Zároveň vidíme, že pre postupnosť P platí, že $P_n = C_n + P_{n-1}$.
Pomocou postupnosti P vieme vzorec pre C aj pre P počítať za pomoci jednej predchádzajúcej hodnoty. A takéto postupnosti sa ľahko počítajú tabuľkovými počítačmi.
Naša vstupná páska preto bude vyzerať a meniť sa nasledovne: $(C_n, P_{n-1}, n+1, 1) \to (C_{n+1}, P_n, n+2, 1)$.
Programy, ktoré menia jednotlivé hodnoty pásky tak, aby platil tento prechod je už následne ľahké dopočítať.



Podúloha d): Oscilujúce prefixové súčty

Postupnosť D, kde $D_0 = -4$ a pre $n \geq 1$ platí
$D_n = 3 + 2 \cdot D_{n-1} - D_{n-2} + D_{n-3} - D_{n-4} + \cdots \pm D_0$

Pri riešení tejto úlohy ostáva spojiť všetko, čo sme sa doteraz naučili. Ľubovoľne dlhú postupnosť striedavo odčítaných a pričítaných hodnôt si potrebujeme nahradiť jednoduchšie počítateľnou postupnosťou.
Nech $P_{n-2} = -D_{n-2} + D_{n-3} - D_{n_4} + \cdots \pm D_0$
Ako vieme vypočítať z hodnoty P_{n-2} hodnotu P_{n-1}? Keď si P_{n-1} rozpíšeme, všimneme si, že všetky prvky D majú opačné znamienko ako v hodnote P_{n-2} a navyše sme ešte na začiatku odčítali hodnotu D_{n-1}. Z toho vyplýva, že $P_{n-1} = -D_{n-1} - P_{n-2}$.
Navyše platí, že $D_n = 3 + 2 \cdot D_{n-1} + P_{n-2}$.
Naša vstupná páska teda bude vyzerať a meniť sa nasledovne: $(D_n, P_{n-1}, 1) \to (D_{n+1}, P_n, 1)$.



Pár slov za koncom

Tabuľkové počítače, s ktorými ste sa stretali v týchto úlohách, v matematike voláme *matice*. Vstupné pásiky sú *vektory*, no a operácia „kroku výpočtu“, teda jedno použitie násobičov a výroba nového pásika zo starého, je vlastne operácia *vynásobenia vstupného vektora príslušnou maticou*. Keď sa o pár rokov s týmito objektmi bližšie zoznámite pri štúdiu lineárnej algebry, budete už trochu tušiť, že tento veľmi jednoduchý nástroj vie robiť veci o dosť komplikovanejšie, ako by sa na prvý pohľad mohlo zdať :)

TRIDSIATY DEVIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

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