P-III-1 Znovu čokoláda

Nejprve si ukážeme řešení s časovou složitostí $\mathcal{O}(R^2S)$. Bude založeno na jednoduché myšlence: vyzkoušíme všechny dvojice řádků a pro každou dvojici v čase $\mathcal{O}(S)$ spočítáme všechny obdélníky, které právě tam mají svůj horní a dolní okraj.

Když už jsme si zvolili horní a dolní řádek, máme vymezen pás políček. Některé jeho sloupce jsou celé bílé, ty můžeme použít. Některé obsahují aspoň jedno šedé políčko a ty použít nemůžeme. Když budeme vědět, které sloupce použít můžeme a které ne, dostaneme vlastně jednorozměrnou verzí původní úlohy. Máme řádek nul a jedniček a chceme v něm spočítat počet úseků tvořených pouze jedničkami. To uděláme tak, že pro každé místo zjistíme počet úseků jedniček, které na tomto místě končí, a všechny tyto počty sečteme. Počet úseků jedniček, které na daném místě končí, je zjevně roven počtu jedniček, které uvidíme, když půjdeme z daného místa doleva až po nejbližší nulu (nebo na začátek řádku). Tento počet si můžeme průběžně počítat, když zpracováváme čísla na řádku: při zpracování jedničky vždy zvýšíme hodnotu počítadla, a při zpracování nuly počítadlo vynulujeme. Takto celý řádek zpracujeme v čase lineárním vzhledem k jeho délce, tzn. v čase $\mathcal{O}(S)$.

Zbývá dořešit, jak zjistíme, které sloupce můžeme použít a které ne. K tomu nám stačí procházet dvojice řádků v systematickém pořadí. Pro dvojici (r_1, r_1) tuto informaci máme „zadarmo“ přímo na vstupu. Když pro dvojici řádků (r_1, r_2) víme, které sloupce se ještě dají použít, potom pro dvojici $(r_1, r_2 + 1)$ tuto informaci snadno určíme. Jsou to ty sloupce, které bylo možné použít pro (r_1, r_2) a zároveň mají jedničku i v řádku $r_2 + 1$.

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

int R, S, A[5012][5012], zije[5012];

int main() {
  cin >> R >> S;
  for (int r=0; r<R; r++) for (int s=0; s<S; s++) cin >> A[r][s];
  long long result = 0;
  for (int r1=0; r1<R; r1++) {
    for (int s=0; s<S; s++) zije[s]=1;
    for (int r2=r1; r2<R; r2++) {
      for (int s=0; s<S; s++) zije[s] &= A[r2][s];
      int run = 0;
      for (int s=0; s<S; s++) if (zije[s]) result += (++run); else run=0;
    }
  }
  cout << result << endl;
  return 0;
}
```

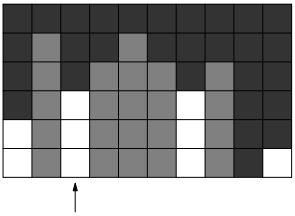
Nyní si předvedeme řešení s optimální časovou složitostí $\mathcal{O}(RS)$. Naše řešení bude založeno na následující základní myšlence: Postupně pro každý řádek spočítáme všechny obdélníky, které právě zde mají svůj dolní okraj.

Při procházení nějakého řádku budeme o každém jeho políčku potřebovat vědět, jakou má *výšku* – tj. jak dlouhá nahoru jdoucí souvislá posloupnost jedniček na něm začíná. Výšky pro nový řádek spočítáme z výšek pro předcházející řádek stejně, jako jsme si počítali v předchozím řešení to, které sloupce ještě můžeme použít.

Představme si, že vezmeme všech S sloupců a seřadíme je od nejvyššího po nejnižší. V tomto pořadí je budeme zpracovávat. Vždy, když zpracujeme sloupec, započítáme všechny obdélníky, které právě přibyly. Jsou to ty obdélníky, které mají dolní okraj na právě zpracovávaném řádku, obsahují aspoň jedno políčko právě zpracovávaného sloupce a celé leží v již zpracovaných sloupcích.

Tímto způsobem každý obdélník jistě započítáme právě jednou – tehdy, když zpracujeme poslední ze sloupců, v nichž leží. Zbývá tedy už jen jediné – tyto počty obdélníků efektivně určit.

Situace uprostřed zpracování řádku s výškami 2, 5, 3, 4, 5, 4, 3, 4, 0, 1. Šedou barvou jsou vyznačeny už zpracované sloupce, šipkou je označen právě zpracovávaný sloupec.



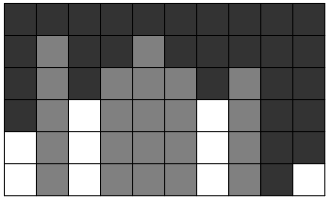
Když zpracováváme nějaký sloupec, potřebujeme umět určit počet jemu přiřazených obdélníků. Jelikož každý z nich má už pevně zvolen spodní okraj, je určen třemi hodnotami: výškou a tím, jak daleko doleva a jak daleko doprava od právě zpracovávaného sloupce sahá.

To, jak daleko doleva a doprava může sahat, je samozřejmě nanejvýš rovno počtu již zpracovaných sloupců ležících bezprostředně vlevo a vpravo. Například v situaci na obrázku by při zpracování sloupce označeného šipkou mohly obdélníky sahat až 1 políčko doleva a 3 políčka doprava. Všechny zpracované sloupce jsou aspoň tak vysoké, jako aktuální sloupec. Proto každé přípustné kombinaci hodnot (výška, počet políček doleva, počet políček doprava) skutečně odpovídá platný obdélník. (Toto je důvod, proč sloupce zpracováváme uspořádané podle výšky a ne v jiném pořadí.) Počet obdélníků přiřazených aktuálnímu sloupci zjistíme tudíž jednoduše z uvedených tří hodnot.

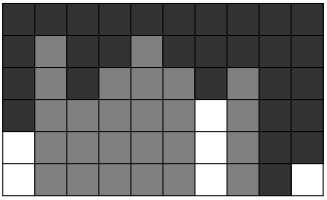
Ještě musíme dořešit, odkud efektivně zjistíme, jak daleko doleva a doprava mohou sahat obdélníky pro aktuální sloupec. Všimněte si, že v každém okamžiku tvoří už zpracované sloupce několik souvislých úseků. Pro každý souvislý úsek si budeme pamatovat dvě čísla: na jeho levém okraji jak daleko doprava, a na jeho pravém okraji jak daleko doleva sahá. Takto se při zpracování sloupce v konstantním čase dozvíme informace, které potřebujeme (jsou uloženy v jeho bezprostředních sousedech), a po jeho zpracování také dokážeme tyto informace v konstantním čase

upravit – známe začátek i konec úseku obsahujícího právě zpracovaný sloupec, tak do jeho konců zapíšeme jeho novou délku.

Na následujících obrázcích vidíte uložené informace před zpracováním a po zpracování šipkou označeného sloupce z předcházejícího obrázku.



doleva	1			3		1			
doprava	1		3			1			



doleva						5		1	
doprava		5						1	

Na závěr ještě jednou shrneme naše řešení. Postupně pro každý z R řádků provedeme následující kroky: Nejprve v čase $\mathcal{O}(S)$ spočítáme výšky pro tento řádek. Potom seřadíme všechny sloupce podle výšky – jelikož výšky jsou celá čísla od 0 do R, dokážeme to provést v čase $\mathcal{O}(R+S)$ například přihrádkovým tříděním (bucket sort). Následně pro každý sloupec v konstantním čase zjistíme počet jemu přiřazených obdélníků a rovněž v konstantním čase upravíme uložené délky souvislých úseků. I tato fáze zpracování řádku tedy proběhne v čase $\mathcal{O}(S)$.

Celková časová složitost algoritmu vychází $\mathcal{O}(R\cdot(R+S))$. Jelikož můžeme předpokládat, že $R \leq S$ (jinak vstup transponujeme, čímž se počet obdélníků nezmění), je výsledná časová složitost rovna $\mathcal{O}(RS)$. Tato složitost je zjevně optimální, když musíme přečíst a zpracovat takto velký vstup.

```
#include <algorithm>
#include <cstdio>
using namespace std;

int R, S;
int A[5012][5012], vv[5012][5012];
int vyska[5012], doleva[5012], doprava[5012], pocty[5012];

int main() {
  scanf("%d %d ", &R, &S);
  for (int r=1; r<=R; r++) for (int s=1; s<=S; s++) scanf("%d", &A[r][s]);
  long long result = 0;
  for (int r=1; r<=R; r++) {
    for (int s=1; s<=S; s++) if (A[r][s]) vyska[s]++; else vyska[s]=0;
    fill(doleva, doleva+S+2, 0);
    fill(doprava, doprava+S+2, 0);
    fill(pocty, pocty+r+2, 0);
    for (int s=1; s<=S; s++) vv[ vyska[s] ][ pocty[vyska[s]]++ ] = s;
    for (int v=r; v>0; v--) for (int ss=0; ss<pocty[v]; ss++) {
      int s = vv[v][ss];
      result += v * (doleva[s-1]+1) * (doprava[s+1]+1);
      int vlevo = s-doleva[s-1], vpravo = s+doprava[s+1], delka = vpravo-vlevo+1;
```

```
      doprava[vlevo] = doleva[vpravo] = delka;
    }
  }
  printf("%Ld\n", result);
  return 0;
}
```

P-III-2 Šachovnice

Tak jako každá matematická hra,* i ta naše se v každém okamžiku nachází v nějaké pozici (stavu). Pozici naší hry můžeme jednoznačně popsat tím, že uvedeme aktuální souřadnice všech koní.

Vyhrávající strategií rozumíme postup, který hráči zajistí vítězství bez ohledu na tahy protihráče. Pozici nazveme *vyhrávající,* jestliže hráč, který je v ní na tahu, má vyhrávající strategii. Pozici, která není vyhrávající, označíme jako *prohrávající*. Každá pozice hry je tedy buď vyhrávající, nebo prohrávající.

Pozice, z níž neexistuje tah, se nazývá *koncová*. Všechny koncové pozice v naší hře jsou podle pravidel prohrávající.

V soutěžní úloze máme proti sobě optimálně hrajícího protihráče. Zajímá nás, zda je počáteční pozice pro nás vyhrávající, nebo ne. Při ohodnocování pozic nám pomohou následující myšlenky:

- Je-li pozice koncová, je prohrávající.
- Jestliže z dané pozice všechny tahy vedou do vyhrávajících pozic, potom je tato pozice prohrávající.
- Jestliže v dané pozici existuje tah vedoucí do nějaké prohrávající pozice, potom je tato pozice vyhrávající.

Když všechny tahy vedou do vyhrávajících pozic, ať si zvolíme kterýkoliv z nich, vždy tím dostaneme soupeře do vyhrávající pozice. Pokud se potom bude soupeř držet nějaké vyhrávající strategie, hru prohrajeme. Proto je taková pozice prohrávající. Naopak, když existuje tah do prohrávající pozice, provedeme ho a tím dostaneme soupeře do této pro něho prohrávající pozice.

Uvedenou myšlenku snadno přepíšeme do rekurzivní funkce, která nám o pozici sdělí, zda je vyhrávající nebo prohrávající.

Hra s jedním nebo dvěma koňmi

Podle nenápadné rady v zadání úlohy se nejprve zamyslíme nad jednodušší verzí hry. Předpokládejme, že na šachovnici je jenom jeden kůň.

Problém předcházejícího přístupu spočívá v tom, že je příliš pomalý. Při rekurzivních voláních se vlastně zkoušejí všechny možné průběhy hry a při tom se mnohé pozice vyhodnocují vícekrát. Pomoci si přitom můžeme poměrně snadno využitím pole – jakmile o nějaké pozici zjistíme, zda je vyhrávající nebo prohrávající, zapíšeme si to do pomocného pole. Tím dosáhneme toho, že každou pozici budeme

* Přesněji konečná kombinatorická hra s úplnou informací.

zpracovávat pouze jednou a pro každou zpracovávanou pozici se vykoná konstantní počet operací.

Kolik pozic budeme zpracovávat? Každým tahem se přiblížíme k políčku $(0,0)$, neboli součet souřadnic se nám sníží aspoň o jedna. To znamená, že počet zpracovávaných stavů můžeme zhruba shora odhadnout počtem bodů, které mají součet souřadnic menší nebo stejný jako počáteční pozice. Pro počáteční pozici (X,Y) je takových pozic $\mathcal{O}((X+Y)^2)$ a taková je tedy i časová složitost popsaného algoritmu.

Uvedenou myšlenku lze snadno rozšířit i na více koní. Máme-li ve hře N koní a každý začíná na souřadnicích se součtem nejvýše S, je počet dosažitelných pozic řádově roven S^{2N}. Toto řešení je proto použitelné jen pro velmi malé hodnoty N.

Rychlejší charakterizace pozic pro jednoho koně

Vraťme se k nejjednodušší možné hře s jediným koněm. Jestliže chceme rychleji určit, která pozice je vyhrávající a která prohrávající, měli bychom nalézt charakterizaci pozice, která nevyužívá znalosti o okolních pozicích. Často pomůže spočítat si pro několik nejmenších pozic, zda jsou vyhrávající, a hledat nějaký vztah.

Když v našem případě vyhodnotíme všechny pozice, jejichž součet souřadnic nepřesahuje 13, dostaneme následující tabulku:

```
0
0 0
1 1 1
1 1 1 1
0 0 1 1 0
0 0 1 1 0 0
1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
0 0 1 1 0 0 1 1 0
0 0 1 1 0 0 1 1 0 0
1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
0 0 1 1 0 0 1 1 0 0 1 1 0
0 0 1 1 0 0 1 1 0 0 1 1 0 0
```

Políčko $(0,0)$ leží v levém dolním rohu tabulky. Nulou jsme označili prohrávající pozice, jedničkou vyhrávající. Není těžké všimnout si pravidelného vzorku – celá šachovnice je složena z dlaždic velikosti 4×4, na nichž jsou 4 políčka vlevo dole prohrávající a všechna zbývající vyhrávající. Můžeme tedy vyslovit následující hypotézu:

$$\text{Pozice } (X,Y) \text{ je prohrávající právě tehdy, když}$$
$$X \bmod 4 \in \{0,1\} \text{ a } Y \bmod 4 \in \{0,1\}.$$

Tuto hypotézu dokážeme matematickou indukcí podle součtu $X+Y$. Pro malé hodnoty jsme už tvrzení dokázali sestrojením uvedené tabulky, stačí tedy provést

indukční krok. Mějme nějakou pozici (X, Y), přičemž už víme o všech pozicích, které mají součet souřadnic menší než $X+Y$, že pro ně naše hypotéza platí. Dokážeme, že platí i pro (X, Y). Rozebereme několik možností podle toho, jaké zbytky po dělení 4 dávají X a Y.

- Obě souřadnice dávají zbytek 0 nebo 1:
  Ať táhneme jakkoliv, jednu souřadnici vždy změníme o 2 a druhou o 1. Ta souřadnice, kterou jsme změnili o 2, bude nyní dávat po dělení 4 zbytek 2 nebo 3. Podle indukčního předpokladu tedy bude tato nová pozice pro hráče na tahu vyhrávající.
  Tím jsme zdůvodnili, že ať táhneme jakkoliv, vždy dostaneme soupeře do vyhrávající pozice. Naše pozice je proto prohrávající, což jsme chtěli dokázat.
- Jedna souřadnice dává zbytek 2 nebo 3, druhá 0 nebo 1:
  Táhneme tak, že první souřadnici zmenšíme o 2. Pokud druhá dává zbytek 1, tak ji o 1 zmenšíme, jinak ji o 1 zvětšíme.
- Obě souřadnice dávají zbytek 2: Zmenšíme X o 2 a Y o 1.
- Obě souřadnice dávají zbytek 3: Zmenšíme X o 2 a zvětšíme Y o 1.
- Jedna souřadnice dává zbytek 2 a druhá 3: První souřadnici zmenšíme o 1 a druhou o 2.
  Ve všech čtyřech případech jsme ukázali tah, kterým soupeře dostaneme do pozice, která je podle indukčního předpokladu prohrávající. Z toho plyne, že naše pozice musela být ve všech těchto případech vyhrávající, což jsme chtěli dokázat.

Pro hru s jedním koněm tedy umíme v konstantním čase určit, zda je pozice vyhrávající nebo ne.

Obecná hra

Při hledání obecného řešení nám pomůže dívat se na vzor rozložení políček uvedený výše a zkoušet pohybovat na šachovnici dvěma, případně jiným malým počtem koňů. Tímto způsobem můžeme objevit následující charakterizaci pozic:

- Jsou-li všechny koně na políčkách s nulou, je pozice hry prohrávající.
- Pokud existuje kůň, který je na políčku s jedničkou, je pozice hry vyhrávající.

Důkaz: Pozici, v níž jsou všechny koně na políčkách s nulou, budeme nazývat modrá, ostatní pozice nazveme červené. Snadno nahlédneme, že platí následující tři tvrzení:

- Všechny koncové pozice hry jsou modré.
- Jsme-li v modré pozici, která není koncová, potom libovolným tahem z ní přejdeme do červené pozice.
  Musíme totiž pohnout aspoň jedním koněm a protože ve hře s jedním koněm všechny tahy z prohrávající pozice (tj. z políčka s nulou) vedou do vyhrávajících pozic (na políčka s jedničkou), pohybující se kůň skončí na políčku s jedničkou, takže nová pozice bude opravdu červená.

- Jsme-li v červené pozici, můžeme z ní platným tahem přejít do modré pozice.

  Zvolíme všechny koně, které stojí na políčkách s jedničkou. Pro každého z nich existuje skok na nějaké políčko s nulou. Tyto skoky vykonáme, čímž dosáhneme výsledné modré pozice.

Vidíme, že modré a červené pozice přesně splňují naši definici prohrávajících a vyhrávajících pozic. Nutně tedy jsou všechny modré pozice prohrávající a všechny červené vyhrávající.

Při hře s N koňmi stačí pro každého z nich v konstantním čase zjistit, zda je na políčku s nulou nebo na políčku s jedničkou. V případě, že existuje kůň na políčku s jedničkou, pozice je vyhrávající. V takovém případě umíme o každém koni na políčku s jedničkou zjistit v konstantním čase, kam je ho třeba přesunout. Časová složitost tohoto řešení je tedy $\mathcal{O}(N)$. Všimněte si, že můžeme jednotlivé koně zpracovávat po jednom, takže nám stačí konstantní paměť.

```
#include <iostream>
using namespace std;
int N,x,y;
int dx[4] = {-2,-2,-1,1}, dy[4] = {1,-1,-2,-2}; // povolené tahy koně

// Zjistí, zda je kůň na prohrávající pozici ve hře s 1 koněm
bool prohravajici(int x, int y) {
  return  (x%4 < 2) && (y%4 < 2);
}

int main() {
  cin >> N;
  int prohra=1;
  for (int i=0; i<N; i++) {
    cin >> x >> y;
    if (!prohravajici(x, y)) {
      if (prohra == 1) cout << "Jenicek" << endl;
      prehra = 0;
      // Je třeba pohnout každým koněm, který není na prohrávající pozici
      cout << x << " " << y << " ";
      for (int j=0; j<4; j++) {
        int nx = x + dx[j], ny = y + dy[j];
        if (nx < 0 || ny < 0) continue;
        if (prohravajici(nx, ny)) {
          cout << nx << " " << ny << endl;
          break;
        }
      }
    }
  }
  if (prohra == 1) cout << "Marenka" << endl;
  return 0;
}
```

P-III-3 Počítač Kvak

Podúloha a: Kontrola, zda je posloupnost rostoucí

Začneme oblíbeným trikem: na konec posloupnosti si jako zarážku vložíme 0. Nyní přečteme do registru a první číslo posloupnosti a za zarážku vložíme číslo 1. Od tohoto okamžiku budeme číst čísla z roury střídavě do registrů b a a. Vždy, když přečteme nové číslo, porovnáme ho s předcházejícím. Je-li nové číslo větší, vložíme do roury další 1. Pokud ne, našli jsme právě hledanou první chybu. Všimněte si, že počet 1, které jsme dosud do roury vložili, je přesně roven indexu, který máme vypsat. Stačí tedy odstranit z roury všechno až po zarážku a následně všechny 1 sečíst, abychom dostali požadovaný výsledek.

Jestliže se nám podaří přečíst celou zadanou posloupnost až po zarážku, program jednoduše ukončíme.

Celý program může vypadat následovně:

```
get a ; put 0
jz a konec
jump cti_b

label cti_b
  put 1 ; get b
  jz b konec
  jgt b a cti_a
jump vyprazdni

label cti_a
  put 1 ; get a
  jz a konec
  jgt a b cti_b
jump vyprazdni

label vyprazdni
  get a ; jz a scitej
jump vyprazdni

label scitej
  get a ; jempty vypis ; put a ; add
jump scitej

label vypis
  put a ; print
label konec
```

Podúloha b: Hledání majoritního prvku

První, snadno realizovatelné řešení je založeno na následující myšlence: Nechť a a b jsou dva prvky zadané posloupnosti, které jsou navzájem různé. Když oba tyto prvky vynecháme, dostaneme opět posloupnost, která má majoritní prvek (a je jím tentýž prvek, jako předtím). Odstranili jsem totiž nejvýše jeden z prvků, kterých byla většina, a aspoň jeden z prvků, kterých většina nebyla.

Náš program bude stále opakovat tento postup: Čte posloupnost a ověřuje, zda jsou všechny prvky stejné. Pokud dočte až na konec a zjistí, že ano, jeden z nich vypíše a skončí. V opačném případě najde dvojici různých prvků, tuto dvojici odstraní, posloupnost dočte až do konce a začne ji zpracovávat znovu.

Program má kvadratickou časovou složitost vzhledem k délce zadané posloupnosti. Každou iteraci uvedeného postupu vykonáme v lineárním čase a každým jejím vykonáním zmenšíme délku posloupnosti o 2.

Mohli bychom sice v jedné iteraci vyhazovat i více dvojic, jestliže je nacházíme postupně, ale v nejhorším případě bude i takové řešení potřebovat kvadraticky mnoho kroků – například pro posloupnost obsahující k jedniček a následně $k+1$ dvojek. Proto raději zůstaneme u programu, který snáze zapíšeme.

```
put 0

label kolo
  get a
  label loop
    get b ; jz b konec
    jeq a b dale
  jump docti

  label dale
    put b
  jump loop

  label docti
    get a ; jz a docetl ; put a
  jump docti

  label docetl
    put 0
jump kolo

label konec
put a ; print
```

Existuje ale i lepší řešení. Předpokládejme nejprve, že je celková délka n naší posloupnosti sudá. Rozdělíme posloupnost na $n/2$ párů. V některých z nich jsou obě čísla stejná, v některých jsou čísla různá. Už jsme zdůvodnili, že každý pár tvořený různými čísly můžeme vynechat. Zůstane nám tedy několik párů, v nichž jsou čísla stejná. Co s nimi? Z každého páru si jedno číslo necháme a druhé zahodíme.

Uvědomte si, že touto operací nic nezkazíme. Pokud měl nějaký prvek většinu předtím, byl nadpoloviční počet dvojic tvořen tímto prvkem. Po „vydělení dvěma“ bude mít tento prvek stále nadpoloviční počet výskytů. Ukázali jsme tedy, že je-li n sudé, dokážeme v lineárním čase převést úlohu na úlohu nejvýše poloviční velikosti.

Zbývá vyřešit, jak to bude vypadat pro liché n. Označme x prvek, který má v zadané posloupnosti většinu výskytů. Dvojici prvků nazveme *dobrá,* pokud se oba prvky rovnají x, a *špatná,* pokud jsou oba prvky stejné, ale odlišné od x. Poslední

prvek posloupnosti označíme p a zatím ho odložíme stranou. Ostatních $2m$ prvků rozdělíme do m dvojic stejně jako předtím.

Jestliže $p \neq x$, jsou všechny výskyty x (kterých je aspoň $m+1$) rozděleny v dvojicích. To je přesně předcházející případ, víme tedy, že dobrých dvojic máme více než špatných.

Jestliže $p = x$, přinejhorším se může stát, že máme přesně m výskytů x v dvojicích a ty jsou rozloženy tak, že je stejný počet dobrých a špatných dvojic. Dobrých dvojic jistě nemůže být méně než špatných.

Nyní rozlišíme dva případy. Pokud je dobrých a špatných dvojic dohromady lichý počet, už máme vyhráno. Jelikož dobrých dvojic musí být aspoň tolik, jako špatných, je jasné, že v tomto případě je dobrých dvojic více. Můžeme tedy postupovat stejně jako v případě sudého n (a poslední prvek p jednoduše zahodit).

Pokud je dobrých a špatných dvojic dohromady sudý počet, ponecháme si z každé dvojice po jednom prvku *a navíc* k nim přidáme poslední prvek p. Čeho tím dosáhneme? Jestliže bylo dobrých dvojic více než špatných, bylo jich nutně více aspoň o dvě (aby byl celkový počet sudý), takže přidáním p nic nezkazíme. Jestliže bylo dobrých dvojic stejně jako špatných, jistě platí $p = x$ a po přidání prvku p bude mít x opět většinu.

Na závěr ještě jednou shrneme celý algoritmus pro $n = 2m + 1$: Postupně vytvoříme m dvojic čísel. Každou dvojici, v níž jsou prvky rozdílné, zahodíme, a z každé dvojice, v níž jsou oba prvky stejné, jeden necháme a druhý zahodíme. Přitom si počítáme paritu počtu prvků, které jsme si nechali. Když se dostaneme k poslednímu prvku p, který už nemá pár, podle spočítané parity rozhodneme, zda ho necháme nebo zahodíme.

Popsaný algoritmus v lineárním čase zpracuje posloupnost, kterou má v rouře, a vytvoří z ní novou posloupnost přibližně poloviční délky. Přitom platí, že nová posloupnost má stejný majoritní prvek jako ta původní. Snadno spočítáme, že opakovaným použitím tohoto algoritmu (dokud nám nezbude jediný prvek) dostaneme odpověď v lineárním čase.

```
label kolo
get a ; jempty konec
put a ; put 0
jump even

label even
  get a ; jz a kolo
  get b ; jz b posledni
  jeq a b pis_odd
jump even
```

```
label odd
  get a ; jz a kolo
  get b ; jz b kolo
  jeq a b pis_even
jump odd

label pis_odd  ; put a ; jump odd
label pis_even ; put a ; jump even
label posledni ; put a ; jump kolo
label konec   ; put a ; print ; stop
```