P-I-1 (Ne)tančící starosta

Slovem *tanečníci* budeme označovat všech $n-1$ lidí kromě starosty. O dvou tanečnících řekneme, že jsou *sousední*, pokud buď stojí přímo vedle sebe, nebo mezi nimi je jen starosta.

Víme, že na konci musí být všichni tanečníci uspořádaní podle výšky. Můžeme je tedy zkusit uspořádat. Následně zkontrolujeme, zda starosta se svou výškou na svém místě do tohoto pořadí pasuje. Pokud ne (je příliš malý nebo velký), máme jistotu, že řešení neexistuje.

Všimněme si nyní, že libovolné dva sousední tanečníky umíme prohodit pomocí jedné taneční figury, která obsahuje jen je dva a případně starostu. Speciálně tedy všechny tanečníky umíme pomocí figur uspořádat podle výšky (přičemž starosta zůstane na svém místě).

Toto stačí na zisk čtyř bodů: Algoritmem BubbleSort, kterému stačí výměny sousedních prvků,* můžeme uspořádat všechny tanečníky, zkontrolovat, zda starosta uspořádání nekazí, a podle toho potom buď vypsat NE, anebo posloupnost figur, které BubbleSort provedl.

Řešení bez starosty

BubbleSort potřebuje na uspořádání x-prvkové posloupnosti v nejhorším případě až $x(x-1)/2$ výměn, což je mnohem více, než si můžeme dovolit my. Lepší řešení bude muset vhodně využívat i figury s větším množstvím tanečníků.

V této části řešení si ukážeme, že libovolný úsek x tanečníků, který neobsahuje starostu, lze uspořádat pomocí nejvýše $x-1$ figur. Hlavní myšlenka je jednoduchá: První figurou dostaneme nejmenšího tanečníka na začátek úseku, druhou figurou druhého nejmenšího tanečníka na druhou pozici a tak dále. Když správně umístíme předposledního tanečníka, poslední už jistě bude na svém místě.

Konkrétně bude každá iterace vypadat následovně: Projdeme celý neuspořádaný zbytek posloupnosti (tj. všechny tanečníky kromě těch, které jsme již dříve umístili na správné místo) a najdeme v něm minimum. Následně provedeme taneční figuru od začátku neuspořádaného zbytku až po pozici, na níž jsme toto minimum našli, čímž jej přesuneme na místo, kam patří.

Řešení za 8 bodů

Výše uvedeným postupem uspořádáme zvlášť všechny tanečníky nalevo od starosty a zvlášť všechny tanečníky napravo od starosty. Na to potřebujeme dohromady nejvýše $n-3$ figur.

Těsně nalevo od starosty je nyní několik tanečníků, kteří svou výškou patří napravo, a těsně napravo od něj jsou zase tanečníci, kteří jsou příliš malí na to,

* Viz kuchařka KSP o třídění – *https://ksp.mff.cuni.cz/kucharky/trideni/*

aby byli v pravé části. Tento úsek najdeme a jednou figurou se starostou uprostřed přesuneme všechny tanečníky na správnou stranu.

Na závěr znovu uspořádáme zvlášť všechny tanečníky nalevo od starosty a zvlášť všechny tanečníky napravo od starosty. Toto řešení si tak vystačí s nejvýše $2n - 5$ figurami.

Řešení za plný počet bodů

Podobně jako v předchozím řešení budeme rozlišovat *levou část* (pozice nalevo od starosty, kam patří tanečníci menší než starosta) a *pravou část* (kam patří ti vyšší).

Na plný počet bodů potřebujeme řešení, které nikdy neudělá více než $3n/2$ figur. Níže si jedno takové řešení ukážeme. Začneme tím, že se podíváme, která část je kratší, a tu vyřešíme jako první tak, že na každého tanečníka použijeme nejvýše dvě figury. Delší část potom vyřešíme již známým postupem, který na každého tanečníka potřebuje nejvýše jednu figuru.

Pozice $s - i$ a $s + i$ budeme nazývat *zrcadlové*. Pokud provedeme figuru na úseku mezi dvěma zrcadlovými pozicemi (včetně nich), vyměníme tím jejich obsah (a také obsah všech dvojic zrcadlových pozic mezi nimi, ale to nám bude jedno).

Bez újmy na obecnosti předpokládejme, že levá část je kratší. Postupně pro každou pozici od 1 po $s - 1$ provedeme následující:

- Pokud je prvek, který na tuto pozici patří, v pravé části, tak nejprve provedeme figuru v pravé části, kterou jej dostaneme na zrcadlovou pozici k té správné. Následně provedeme figuru od správné pozice po její zrcadlovou, čímž dostaneme aktuální prvek na správné místo.
- Jinak v levé části provedeme figuru, kterou aktuální prvek rovnou dostaneme na správnou pozici.

Níže je znázorněný příklad, jak dostáváme na správné místo tanečníky 1, 2 a 3 představující tři nejmenší tanečníky. Písmeno S představuje starostu, tečky jsou ostatní tanečníci, pomlčky znázorňují úsek, který provádí taneční figuru. Všimněte si, že jakmile někoho záměrně umístíme na správné místo, už s ním zaručeně nebudeme nikdy hýbat.

```
..3..2S.........1.
            ------    1 na zrcadlovou pozici
..3..2S.....1......
-------------         1 na správné místo
1.....S2..3........
       -----          2 na zrcadlovou pozici
1.....S.3..2.......
 -----------          2 na správné místo
12..3.S............
  ---                 3 rovnou na správné místo
123...S............
```

Při implementaci je důležité si uvědomit, že nejvyšší n je malé ($n \leq 1000$). To zaprvé znamená, že si můžeme dovolit přímočaře odsimulovat všechny potřeb-

né figury. A zadruhé to znamená, že i operace, které bychom mohli implementovat efektivněji, můžeme ve skutečnosti implementovat přímočaře. Například krok, kdy každému tanečníkovi chceme přiřadit jeho cílovou pozici, bychom sice mohli implementovat s časovou složitostí $\mathcal{O}(n \log n)$, ale mnohem pohodlnější je udělat to kvadraticky.

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

int main() {
    // Načítáme vstup
    int N, S, L;
    cin >> N >> S >> L;
    --S; // V programu indexujeme od nuly
    vector<int> tanecnici(N);
    for (int &x : tanecnici) cin >> x;

    // Ověříme, zda existuje řešení
    vector<int> zaver = tanecnici;
    sort( zaver.begin(), zaver.end() );
    if (zaver[S] != tanecnici[S]) { cout << "NE\n"; return 0; }

    // Explicitně přiřadíme každému tanečníkovi místo, kam patří
    vector<int> stav(N, -1);
    stav[S] = S;
    for (int n=0; n<N; ++n) if (n != S) {
        for (int i=0; i<N; ++i) if (tanecnici[i] == zaver[n] && stav[i] == -1) {
            stav[i] = n;
            break;
        }
    }

    // Zjistíme, která strana je menší, a podle toho si zvolíme,
    // od kterého konce budeme prvky umísťovat na správné místo
    int start = 0, step = 1, end = N;
    if (S > N-1-S) { start = N-1; step = -1; end = -1; }

    // Postupně pro každé n umístíme správného tanečníka na místo n
    vector< pair<int,int> > kroky;
    for (int n=start; n!=end; n+=step) {
        if (stav[n] == n) continue; // Už je tam
        int where = 0;
        while (stav[where] != n) ++where;
        if ( (where-S)*(n-S) < 0 ) {
            // Tanečníka máme na špatné straně starosty
            int mirror = 2*S - n;

            int i = min(mirror,where), j=max(mirror,where)+1;
            kroky.push_back( {i, j} );
            reverse( stav.begin()+i, stav.begin()+j );

            i = min(mirror,n); j = max(mirror,n)+1;
            kroky.push_back( {i, j} );
            reverse( stav.begin()+i, stav.begin()+j );
        } else {
            // Tanečníka máme na správné straně starosty
            int i = min(n,where), j = max(n,where)+1;
            kroky.push_back( {i, j} );
```

```
            reverse( stav.begin()+i, stav.begin()+j );
        }
    }

    // Vypíšeme výstup
    cout << "ANO\n";
    cout << kroky.size() << "\n";
    for (const auto &krok : kroky)
        cout << (krok.first+1) << " " << krok.second << "\n";
}
```

P-I-2 To je ale rozhled!

Začneme tím, že už během načítání vstupu zahodíme po sobě jdoucí opakující se hodnoty. Tím dostaneme nové pohoří, které má přesně stejný tvar jako to na vstupu, ale navíc každá hora je jen bod, nikoliv úsek.

Zjistit, které hory jsou nejvyšší na světě, je lehké. Abychom určili prominenci ostatních hor, potřebujeme pro každou z nich umět efektivně určit dvě věci: Jednak potřebujeme vědět, kde je směrem doleva i doprava nejbližší ostře vyšší bod terénu. A když to už víme, tak potřebujeme určit, jak nejhlouběji mezi nimi klesneme. Jinými slovy, na našem poli nadmořských výšek potřebujeme umět efektivně provádět operaci „najdi nejbližší větší hodnotu v daném směru“ a zároveň určit minimum v úseku k této nejbližší vyšší hodnotě.

Když chceme o každém prvku pole vědět, kde nalevo od něj je nejbližší větší, stačí pole jednou projít zleva a průběžně si udržovat nějaké vhodné údaje v zásobníku. Přesněji to celé bude fungovat následovně: Postupně budeme výšky jednotlivých bodů pohoří zpracovávat zleva doprava. Představte si, že se na už zpracovanou část tohoto pohoří díváme zprava. Některé body vidíme, některé jsou už zakryté jinými body, které jsme zpracovali později. Ty, které vidíme, mají společnou vlastnost: Napravo od každého z nich jsou zatím jen samé menší hodnoty. Právě tyto body si budeme pamatovat v zásobníku. Když na pravém konci pohoří přibude další bod, nejbližší nalevo, který je od něj vyšší, musí nutně být jeden z těch bodů, které si pamatujeme.

Zpracování každého dalšího bodu bude vypadat následovně: Dokud máme na vrchu zásobníku bod, jehož výška je menší či rovna výšce zpracovávaného, vyhodíme ho (tyto body při pohledu zprava přestaly být viditelné.) Bod, který nám zůstal na vrchu zásobníku, je pro právě zpracovávaný nejbližším větším vlevo. Na závěr už jen přidáme právě zpracovaný bod na vrch zásobníku.

Zpracování konkrétního bodu sice může trvat dlouho (pokud toho ze zásobníku musíme zrovna hodně vyhodit), jednoduše však nahlédneme, že dohromady celý algoritmus zabere pouze lineární čas, jelikož každý prvek do zásobníku právě jednou vložíme a nejvýše jednou ho ze zásobníku odebereme.

Potřebujeme ještě umět určit, jaká je nejmenší hodnota mezi aktuálním a nejbližším vyšším prvkem. V zásobníku si tedy kromě výšek prvků budeme také u každého z nich (kromě posledního) ukládat minimum z úseku mezi tímto prvkem a následujícím prvkem v zásobníku. Tato minima snadno zvládneme počítat se stej-

nou časovou složitostí: Kdykoliv vyhazujeme prvek ze zásobníku, dva takové úseky se nám spojí a minimum z výsledného úseku je menší z těchto dvou.

Dohromady tedy strávíme čas $\mathcal{O}(n)$ tím, že pro každý bod najdeme nejbližší vyšší vlevo (a potom i vpravo spuštěním tohoto algoritmu na obrácené pole) a minimum z přislušného úseku. Paměťová složitost $\mathcal{O}(n)$.

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

#define NEKONECNO 1000000007

static vector<int>
prominence_vlevo (const vector<int> &H)
{
  int N = H.size();
  vector<int> prom(N,-1), vysky (1, NEKONECNO), minima;

  for (int n = 0; n < N; ++n)
    {
      int aktmin = H[n];
      while (vysky.back() <= H[n])
        {
          aktmin = min (aktmin, minima.back ());
          vysky.pop_back();
          minima.pop_back ();
        }
      prom[n] = vysky.size () > 1 ? H[n] - aktmin : NEKONECNO;
      vysky.push_back (H[n]);
      minima.push_back (aktmin);
  }

  return prom;
}

static vector<int>
prominence_vpravo (const vector<int> &H)
{
  vector<int> Hcopy (H);
  reverse(Hcopy.begin(), Hcopy.end());
  vector<int> prom = prominence_vlevo (Hcopy);
  reverse(prom.begin(), prom.end ());
  return prom;
}

int main()
{
  // Načteme vstup a odstraníme po sobě jdoucí duplicity
  int N;
  cin >> N;
  vector<int> pohori;
  for (int n = 0; n < N; ++n)
    {
      int vyska;
      cin >> vyska;
      if (n == 0 || vyska != pohori.back())
        pohori.push_back(vyska);
    }
```

```
  N = pohori.size();
  int everest = *max_element(pohori.begin(), pohori.end());

  // Pro každý bod najdeme nejbližší větší vlevo i vpravo
  vector<int> BL = prominence_vlevo (pohori), BR = prominence_vpravo (pohori);

  // Pro každou horu najdeme a vypíšeme její prominenci
  for (int i = 0; i < N; ++i)
    {
      if (i-1 >= 0 && pohori[i-1] >= pohori[i])
        continue;
      if (i+1 <  N && pohori[i+1] >= pohori[i])
        continue;
      if (pohori[i] == everest)
        cout << everest << "\n";
      else
        cout << min(BL[i],BR[i]) << "\n";
    }

  return 0;
}
```

P-I-3 Střelec

Úlohu budeme řešit pomocí dynamického programování.* Postupně pro každý počet tahů i a každou souřadnici (j, k) vypočítáme hodnotu $P[i, j, k]$, totiž počet způsobů, kterými se umíme dostat na políčko (j, k) s tím, že jsme zatím navštívili ve správném pořadí prvních i písmen slova w, modulo $10^9 + 7$. Dále v řešení již pro jednodušší čitelnost nebudeme zmiňovat, že u veškerých výpočtů nás ve skutečnosti zajímá jen zbytek po dělení $10^9 + 7$.

Některé tyto hodnoty jsou zřejmé. Označme jako $S[j, k]$ písmeno na políčku (j, k) na šachovnici. Pokud $S[j, k] \neq w[i]$, tak zřejmě $P[i, j, k] = 0$, jelikož po přeskákání prvních i písmen slova w na nějakém políčku můžeme skončit jedině tehdy, když obsahuje i-té písmeno slova w. Také je zřejmé, že pokud $S[j, k] = w[1]$, tak $P[1, j, k] = 1$, jelikož jediný způsob, jak navštívit první písmeno slova w, je začít na něm.

Pro výpočet zbylých hodnot $P[i, j, k]$ si uvědomme, že skončit po i tazích na políčku (j, k) lze tak, že provedeme prvních $i-1$ tahů, kterými se dostaneme na nějaké políčko, a z něj následně i-tým tahem přejdeme na políčko (j, k). Posloupnosti skoků, které nás tam dovedou z různých políček, jsou zjevně navzájem různé, $P[i, j, k]$ tedy dostaneme jako součet $P[i - 1, j', k']$ přes všechna políčka (j', k'), z nichž můžeme i-tým krokem skočit na (j, k). Řešením úlohy je potom součet všech hodnot $P[n, j, k]$ přes všechna možná j a k, kde n je délka slova w.

Jelikož se střelec pohybuje jen šikmo, na šachovnici s rozměry $r \times s$ je nejvýše $2\min(r, s) - 2$ možností, odkud přijít na konkrétní políčko. Takto přímočará implementace dynamickým programováním zajistí, že každou z hodnot $P[i, j, k]$ umíme vypočítat v čase $\mathcal{O}(\min(r, s))$, a tedy celková časová složitost tohoto řešení je $\mathcal{O}(nrs \min(r, s))$.

---

* Viz kuchařka KSP – *https://ksp.mff.cuni.cz/kucharky/dynamika/*

Rychlejší řešení

Výše popsané řešení můžeme zefektivnit tak, že budeme šikovněji sčítat možnosti, jak se na políčko dostat, a tím časovou složitost zlepšíme na $\mathcal{O}(nrs)$. Políčkem se souřadnicemi (j, k) procházejí dvě úhlopříčky: Jednu z nich tvoří ta políčka, která mají součet souřadnic $j + k$, druhou tvoří ta, která mají rozdíl souřadnic $j - k$. Označme si jako $A[i, x]$ celkový počet způsobů, jak se pomocí i tahů dostat na nějaké políčko úhlopříčky se součtem souřadnic x, a jako $B[i, y]$ celkový počet způsobů, jak se pomocí i tahů dostat na nějaké políčko úhlopříčky s rozdílem souřadnic y.

Když spočítáme hodnoty $P[i, j, k]$ pro pevné i a pro všechna j a k, můžeme $A[i, x]$ a $B[i, y]$ pro všechna x a y spočítat v čase $\mathcal{O}(rs)$, stačí postupně projít všechna políčka (j, k) a hodnotu $P[i, j, k]$ přičíst k $A[i, j+k]$ a k $B[i, j-k]$. Když nyní známe hodnoty $A[i, x]$ a $B[i, y]$ pro všechna x a y, můžeme pomocí nich v čase $\mathcal{O}(rs)$ spočítat hodnoty $P[i + 1, j, k]$ pro všechna j a k, totiž $P[i + 1, j, k] = A[i, j + k] + B[i, j - k] - 2P[i, j, k]$ (jelikož nesmíme zůstávat na místě, musíme odečíst dvakrát hodnotu $P[i, j, k]$, jednou za $A[i, j + k]$ a podruhé za $B[i, j - k]$).

Časová složitost řešení je tedy $\mathcal{O}(nrs)$. Paměťová složitost popsaného řešení je také $\mathcal{O}(nrs)$, ale můžeme si všimnout, že v každém okamžiku stačí znát hodnoty P, A a B nejvýše pro i a $i + 1$ (jak tomu je v ukázkové implementaci), a tudíž lze paměťovou složitost snížit na $\mathcal{O}(rs)$.

```
from collections import defaultdict

R, C = [ int(_) for _ in input().split() ]
W = input()
board = [ input() for r in range(R) ]

# Zjistíme počet způsobů pro první písmeno (0 nebo 1)
P = [ [ int( board[r][c] == W[0] ) for c in range(C) ] for r in range(R) ]

# Postupně pro každé další písmeno sčítáme úhlopříčky
# a pomocí nich spočítáme nové hodnoty P
for n in range(1,len(W)):
    A, B = defaultdict(int), defaultdict(int)
    for r in range(R):
        for c in range(C):
            A[r+c] += P[r][c]
            B[r-c] += P[r][c]

    newP = [ [ 0 for c in range(C) ] for r in range(R) ]
    for r in range(R):
        for c in range(C):
            if W[n] != board[r][c]: continue
            newP[r][c] = A[r+c] + B[r-c] - 2*P[r][c]

    P = newP

print( sum( sum(row) for row in P ) )
```

V implementaci jsme pro jednodušší čitelnost vynechali počítání modulo 10^9+7 a použili jsme asociativní pole (`defaultdict`), abychom nemuseli ošetřovat záporné indexy v poli B. Samozřejmě bychom místo $B[j - k]$ mohli také používat $B[(j - k) + (s - 1)]$ a tehdy by A i B mohly být obyčejná pole.

P-I-4 Vozidlo na Marsu

Řešení podúloh uvádíme ve stejném pořadí jako v zadání.

Podúloha a: Vynásob jámu čtyřmi

Počet kamínků v lokalitě `jáma` můžeme zdvojnásobit tak, že z kamenolomu do jámy přeneseme tolik kamenů, kolik jich právě v jámě je. Když toto dvakrát zopakujeme, máme kýžený výsledek. V programu to vypadá následovně:

```
přenes K jáma jáma -
přenes K jáma jáma -
```

Podúloha b: Porovnej počty kamenů

V lokalitě `A` je a kamenů, v lokalitě `B` jich je b. Otestovat, zda $a \geq b$, můžeme následovně: Z lokality `A` zkusíme do kamenolomu odnést tolik kamenů, kolik jich je v lokalitě `B`. Pokud se to nepodaří, víme, že $a < b$. Pokud se to podaří, víme, že $a \geq b$, a v tom případě můžeme do `A` přidat zpět tolik kamenů, kolik jich je v `B`, po čemž jich tam budeme mít stejné množství jako na začátku.

Test na rovnost můžeme udělat tak, že nejdříve otestujeme, zda $a \geq b$, a pokud ano, tak otestujeme i zda $b \geq a$.

```
přenes A B K konec
přenes K B A -
přenes B A K konec
přenes K A B -
přenes K J C -
```

To stejné řešení lze zapsat i stručněji: Místo do kamenolomu můžeme přenášet kameny rovnou zpět do té lokality, odkud je bereme. Tím je zaručené, že se počet kamenů nezmění, vozidlo však přesto vyhodnotí, zda se operace podařila, a podle toho skočí anebo neskočí na příslušné návěstí.

```
přenes A B A konec
přenes B A B konec
přenes K J C -
```

Podúloha c: Makro pro dělení se zbytkem

V lokalitách `A` a `B` máme vstup, tedy hodnoty a a b, přičemž $b > 0$. Do lokality `P` chceme uložit celou část podílu a/b a do lokality `Z` zbytek po tomto dělení. Postup bude jednoduchý. Na začátku zkopírujeme obsah `A` do lokality `Z`. Následně budeme v cyklu z lokality `Z` odebírat vždy b kamenů najednou a za každé úspěšné odebrání přidáme jeden kamen do lokality `P`.

Je zřejmé, že tento algoritmus časem skončí (počet kamenů v `Z` se snižuje), a když se tak stane, v lokalitě `P` máme počet úspěšných odebrání (tedy přesně celou část podílu a/b) a v lokalitě `Z` nám zbyly přesně kameny odpovídající zbytku po tomto dělení.

V níže uvedeném programu používáme makra `přidej`, `skoč` a `čekej` definovaná ve studijním textu.

```
MAKRO vyděl A B P Z
            přidej A Z
    cyklus: přenes Z B K konec
            přenes K J P -
            skoč cyklus
    konec:  čekej
KONEC
```

Podúloha d: Největší společný dělitel

V lokalitách `A` a `B` máme nějaké neznámé počty kamenů a a $b > 0$. Chceme vypočítat a do prázdné lokality `C` uložit jejich největšího společného dělitele.

Použijeme Euklidův algoritmus. Ten je založený na dvou pozorováních. První je zjevné: Pro $x > 0$ platí $\mathrm{nsd}(x, 0) = x$. Druhé pozorování, totiž že $\mathrm{nsd}(x, y) = \mathrm{nsd}(y, x \bmod y)$ si zdůvodníme.

Pokud $x < y$, tak tvrdí, že $\mathrm{nsd}(x, y) = \mathrm{nsd}(y, x)$, což zjevně platí. Nechť nyní $x \geq y$. Hodnotu x můžeme zapsat jako $x = py + z$, kde p je celá část podílu x/y a z je zbytek po tomto dělení. Potom naše pozorování říká, že $\mathrm{nsd}(x, y) = \mathrm{nsd}(y, z)$.

Proč tento vztah platí? Každé d, které dělí x i y, musí dělit i $x - py$, čili z. No a naopak, každé d, které dělí y i z, musí dělit i $py + z$, čili x.

Celý Euklidův algoritmus je už velmi jednoduchý: Dokud jsou obě čísla kladná, používáme druhé pozorování, a když jednou klesneme na nulu, použijeme první pozorování a máme hotovo.

Všimněte si, že pro $x \geq y$ se každým použitím druhého pozorování součet aktuálních dvou hodnot zmenší, takže tento algoritmus určitě časem skončí. V této soutěžní úloze nás přesná časová složitost netrápí.

```
# vynuluje X
MAKRO vynuluj X
      přenes X X K -
KONEC

# pokud X obsahuje nulu, skočí na návěstí N
MAKRO nulové X N
      přenes X J X N
KONEC

# hodnotu X zapíšeme do lokality Y, čímž původní obsah Y přepíšeme
MAKRO zapiš X Y
      vynuluj Y
      přidej X Y
KONEC

# hlavní program: pokud B=0, končíme, jinak změníme (A,B) na (B, A mod B)
cyklus: nulové B konec
        vynuluj Z
        vyděl A B odpad Z
        zapiš B A
        zapiš Z B
        skoč cyklus
konec:  zapiš A C
```

V programu používáme makro `vyděl` z podúlohy c), přičemž se dopouštíme jedné nepřesnosti, proměnnou `odpad`, kam se ukládá celá část podílu, na nic nepotřebujeme, a tak se ji ani neobtěžujeme mezi jednotlivými voláními `vyděl` vyprázdnit.

Podúloha d podruhé

Připomeňme si na závěr ještě jednou, že v této soutěžní úloze nás netrápí časová složitost programů, jen jejich správnost. Největšího společného dělitele můžeme vyřešit i tak, že si uvědomíme, že $\mathrm{nsd}(a, b) \leq b$, a postupně pro každé i od b až po 1 (sestupně) otestujeme, zda i dělí a i b. První takovéto i je zřejmě rovné $\mathrm{nsd}(a, b)$.

```
# pokud X není dělitelné Y, skočí na návěstí chyba
MAKRO nedělitelné X Y chyba
          vynuluj [zbytek]
          vyděl X Y [podíl] [zbytek]
          nulové [zbytek] dělilo
          skoč chyba
  dělilo: čekej
KONEC


# hlavní program
        přidej B I
cyklus: nedělitelné A I zmenši
        nedělitelné B I zmenši
        # pokud jsme se dostali sem, I dělí A i B, tedy je to nsd(A,B)
        přidej I C
        skoč konec
        # pokud jsme skončili na návěstí zmenši,
        # zmenšíme I o jedna a hledáme dále
zmenši: přenes I J K -
        skoč cyklus
```