39. ročník matematické olympiády na středních školách

Kategorie P

In: Leo Boček (editor); Jiří Binder (editor); Vladimír Burjan (editor); Karel Horák (editor); Pavel Töpfer (editor): 39. ročník matematické olympiády na středních školách. Zpráva o řešení úloh ze soutěže konané ve školním roce 1989/90. 31. mezinárodní matematická olympiáda. (Czech). Praha: Státní pedagogické nakladatelství, 1992. pp. 106–174.

Kategorie P

Texty úloh

P – I – 1

Napište co nejlepší algoritmus, který vytiskne všechny různé rozklady zadaného přirozeného čísla N na součty přirozených čísel. Rozklady nepovažujeme za různé, jestliže se liší pouze pořadím sčítanců. Zdůvodněte správnost algoritmu.

Např. pro $N = 5$ algoritmus vytiskne tyto rozklady (v libovolném pořadí a s libovolným pořadím sčítanců v jednotlivých rozkladech):

$$5 = 1 + 1 + 1 + 1 + 1$$
$$5 = 2 + 1 + 1 + 1$$
$$5 = 2 + 2 + 1$$
$$5 = 3 + 1 + 1$$
$$5 = 3 + 2$$
$$5 = 4 + 1$$
$$5 = 5$$

P – I – 2

V rovině je pevně rozmístěno n bodů označených čísly 1, 2, ..., n. V zadaném poli $D[1..n, 1..n]$ jsou uloženy jejich vzájemné vzdálenosti. Hodnota $D[i, j]$ udává vzdálenost bodu i od bodu j. Dále je dáno číslo t. Zapište algoritmus, který zjišťuje, zda lze uvažovaných n bodů rozdělit do dvou skupin tak, aby vzájemné vzdálenosti všech bodů patřících do stejné skupiny byly menší než t. Pokud takové rozdělení bodů je možné, algoritmus vypíše jedno libovolné přípustné rozdělení. Zdůvodněte správnost algoritmu.

P – I – 3

Je dán následující program:

PASCAL	BASIC
var A, B, C, D, E, F: integer;	
I, N: integer;	
begin	
read(N);	10 INPUT N
$A := -1$;	20 LET A = −1
$B := 0$;	30 LET B = 0
$D := 0$;	40 LET D = 0
$E := 0$;	50 LET E = 0
for $I := 1$ **to** N **do**	60 FOR I = 1 TO N
begin	
$A := A + 2$;	70 LET A = A + 2
$B := B + A$;	80 LET B = B + A
$C := B * I$;	90 LET C = B * I
$D := D + C$;	100 LET D = D + C

```
    E := E + I;                    110 LET E = E + I
    F := E * E;                    120 LET F = E * E
    end;                           130 NEXT I
  writeln(D);                      140 PRINT D
  writeln(F);                      150 PRINT F
end.
```

Vstupem programu je jedno celé číslo (hodnota proměnné N).

a) Odvoďte a dokažte vztah mezi hodnotami proměnných D a F po ukončení výpočtu.

b) Napište program, který bude počítat výsledné hodnoty proměnných D a F v závislosti na vstupní hodnotě proměnné N co nejrychleji. Rychlost výpočtu oběma způsoby (tj. zde uvedeným způsobem a vaším vlastním programem) porovnejte z hlediska počtu provedených aritmetických operací.

Poznámka. Proměnné A, B, C, E a I jsou pomocné, na jejich hodnotách po ukončení výpočtu nezáleží.

P – I – 4

Zásobníkový počítač

Nejprve se seznámíme se zásobníkovým počítačem a s jeho programovacím jazykem. Zásobníkový počítač pracuje výhradně s celými čísly. Oproti běžným počítačům má značná omezení v možnostech práce s paměťovými buňkami. Má sice paměť dostatečné velikosti, ale celá jeho paměť je organizována zásobníkovým způsobem. Zásobník má neomezenou hloubku, ukládají se do něj celá čísla. Čísla se v zásobníku uchovávají v tom pořadí, v jakém do něj byla vložena. To

číslo, které bylo do zásobníku vloženo jako poslední, je vždy na vrcholu zásobníku. Pouze číslo nacházející se na vrcholu zásobníku je v danou chvíli dostupné pro čtení nebo smazání. Po smazání čísla z vrcholu zásobníku se novým vrcholem zásobníku stává číslo uložené pod ním. Na začátku práce je zásobník prázdný. Kromě zásobníkové paměti má počítač k dispozici ještě jeden pracovní registr, do něhož je možné uložit jedno celé číslo. V registru lze provádět základní celočíselné aritmetické operace, prostřednictvím registru může počítač číst čísla ze vstupu a zapisovat čísla na výstup.

Jednoduché příkazy

Základním operacím zásobníkového počítače odpovídají jednoduché příkazy jeho programovacího jazyka:

IN	ze vstupu je přečteno jedno číslo a je uloženo do pracovního registru (pokud při provedení příkazu IN již na vstupu žádné číslo není, dojde k běhové chybě)
OUT	číslo z pracovního registru je zapsáno na výstup, obsah registru zůstane nezměněn
CONST *n*	do pracovního registru se uloží číslo „*n*“, které je v instrukci CONST uvedeno jako přímý operand (jedná se o jediný příkaz s přímým operandem)
PUSH	číslo z pracovního registru je uloženo na zásobník (stane se novým vrcholem zásobníku), obsah registru zůstane nezměněn
POP	odstraní se jedno číslo ze zásobníku (číslo z vrcholu zásobníku); je-li při provedení příkazu POP zásobník prázdný, dojde k běhové chybě

TOP	číslo nacházející se na vrcholu zásobníku je vloženo do pracovního registru, obsah zásobníku zůstane nezměněn (je-li při provedení příkazu TOP zásobník prázdný, dojde k běhové chybě)
EXCH	provede se vzájemná výměna čísel uložených v pracovním registru a na vrcholu zásobníku (je-li při provedení příkazu EXCH zásobník prázdný, dojde k běhové chybě)
ADD	aritmetická operace sčítání: k číslu uloženému v pracovním registru se přičte číslo z vrcholu zásobníku, výsledný součet se uloží do registru, obsah zásobníku zůstane nezměněn
SUB	aritmetická operace odčítání: od čísla uloženého v pracovním registru se odečte číslo z vrcholu zásobníku, výsledný rozdíl se uloží do registru, obsah zásobníku zůstane nezměněn
MUL	aritmetická operace násobení: číslo uložené v pracovním registru se vynásobí číslem z vrcholu zásobníku, výsledný součin se uloží do registru, obsah zásobníku zůstane nezměněn
DIV	aritmetická operace celočíselné dělení: číslo uložené v pracovním registru se celočíselně vydělí číslem z vrcholu zásobníku, výsledný celočíselný podíl (tj. celá část z podílu obou čísel) se uloží do registru, obsah zásobníku zůstane nezměněn

Podmínky

Zásobníkový počítač dále může provádět testy, kterým odpovídají jednoduché podmínky jeho programovacího jazyka a jejich negace. Tyto podmínky lze použít pouze na

místě podmínky v některém složeném příkazu. Existují následující jednoduché podmínky:

ZERO	číslo uložené v pracovním registru je rovno nule
POS	číslo uložené v pracovním registru je kladné
NEG	číslo uložené v pracovním registru je záporné
EMPTY	zásobník je prázdný
EOF	na vstupu již není žádné číslo

Negaci jednoduché podmínky zapisujeme klíčovým slovem **not** před jménem jednoduché podmínky. Tedy např. **not** ZERO znamená, že číslo v registru je nenulové.

Složené příkazy

Program v programovacím jazyce zásobníkového počítače je tvořen posloupností příkazů oddělených mezerami. Každý příkaz je buď jednoduchý (jedenáct výše uvedených základních operací), nebo složený. Složené příkazy se vytvářejí pomocí těchto řídicích konstrukcí:

if ⟨podmínka⟩ **then** ⟨příkaz⟩	neúplný podmíněný příkaz
if ⟨podmínka⟩ **then** ⟨příkaz 1⟩	
else ⟨příkaz 2⟩	úplný podmíněný příkaz
while ⟨podmínka⟩ **do** ⟨příkaz⟩	while-cyklus
begin	
⟨příkaz 1⟩	
⟨příkaz 2⟩	
...	
end	příkaz složený z posloupnosti příkazů

Význam podmíněného příkazu, cyklu i příkazu složeného z posloupnosti příkazů je obdobný jako význam stejnojmen-

ných příkazů v Pascalu. Na místě příkazu může být uveden jednoduchý příkaz nebo opět některý složený příkaz.

Poznámka. Řešitelé, kteří neznají řídicí konstrukce programovacího jazyka Pascal a nemají možnost se s nimi seznámit, mohou místo uvedených složených příkazů použít číslování řádků programu a příkazů

GOTO ⟨číslo řádku⟩

a

IF ⟨podmínka⟩ THEN ⟨příkaz⟩

kde na místě příkazu může stát jednoduchý příkaz nebo příkaz GOTO ⟨číslo řádku⟩ (obdobně jako v jazyce Basic).

Příklad

Způsob programování zásobníkového počítače ukážeme na následujícím příkladu. Naším úkolem bude napsat program, který ze vstupu přečte posloupnost čísel a určí, zda tato posloupnost obsahuje sudý počet kladných čísel menších než 10. Pokud ano, program vytiskne číslo 1, jinak vytiskne 0. Uvedeme zde program, který není nejrychlejším možným programem řešícím zadanou úlohu, ale který dobře ukazuje, jak se zásobníkový počítač programuje. Do závorek budeme zapisovat komentáře vysvětlující způsob práce programu.

```
while not EOF do
  begin
  IN                    (přečtení dalšího čísla)
  if POS then
    begin               (číslo je kladné)
    PUSH                (číslo se uloží do zásobníku)
    CONST 10
```

```
    SUB                 (rozdíl „10 – číslo ze vstupu“)
    if not POS then POP
                        (není-li číslo < 10, smaže se)
    end
  end
```
(nyní jsou ze vstupu přečtena všechna čísla a v zásobníku jsou uložena ta z nich, která jsou kladná a zároveň menší než 10)
```
CONST 1
while not EMPTY do
                        (čísla jsou vždy po dvou vybírána
                        ze zásobníku)
  begin
  POP
  if EMPTY then CONST 0
                        (lichý počet čísel v zásobníku)
            else POP
  end
```
(zásobník je vyprázdněn, v pracovním registru je připravena správná výsledná hodnota)
```
OUT                     (tisk výsledku)
```

Soutěžní úloha

A. Je zadána posloupnost navzájem různých celých čísel tvořená alespoň třemi čísly. Rozhodněte, zda je možné pomocí zásobníkového počítače nalézt a vytisknout

a) největší ze zadaných čísel,

b) dvě největší ze zadaných čísel,

c) tři největší ze zadaných čísel.

Pro každou z úloh a), b), c): Je-li to možné, napište pro-

gram v programovacím jazyce zásobníkového počítače, který tuto úlohu řeší a který pracuje co možná nejrychleji. Domníváte-li se, že to není možné, své stanovisko vysvětlete a zdůvodněte.

B. Je zadána posloupnost obsahující $2n + 1$ celých čísel pro nějaké n celé, nezáporné. Jediným nulovým prvkem této posloupnosti je číslo s pořadím $n + 1$ (tzn. prostřední prvek posloupnosti). Zjistěte pomocí zásobníkového počítače, zda je zadaná posloupnost symetrická, tzn. stejná při čtení zepředu i odzadu. Například posloupnost 2 8 −1 0 −1 8 2 je symetrická, zatímco posloupnost 3 0 5 symetrická není. Odpověď vytiskněte ve tvaru: 1 — je symetrická
0 — není symetrická.

P – II – 1

Je dáno pole $A[1 .. N, 1 .. N]$ obsahující N^2 navzájem různých celých kladných čísel. Navrhněte co nejrychlejší algoritmus, který vytiskne N největších čísel uložených v poli A. Původní obsah pole A nemusí být po ukončení výpočtu zachován. Zdůvodněte správnost navrženého algoritmu.

Poznámka. Optimální algoritmus řeší zadanou úlohu provedením nejvýše $p \cdot N^2$ operací pro nějaké pevné číslo p (tzn. při vyjádření pomocí parametru N má kvadratickou časovou složitost).

P – II – 2

Je dána konečná posloupnost celých čísel délky N. Prvky této posloupnosti označíme po řadě $X(1), X(2), \ldots, X(N)$.

Podposloupností délky k vybranou ze zadané posloupnosti budeme rozumět libovolnou konečnou posloupnost tvaru $X(i_1), X(i_2), \ldots, X(i_k)$, kde $1 \leqq i_1 < i_2 < \ldots < i_k \leqq N$ (tzn. ze zadané posloupnosti je vybráno libovolných k čísel, přičemž je zachováno jejich pořadí).

Navrhněte co nejlepší algoritmus, který určí délku nejdelší rostoucí podposloupnosti vybrané ze zadané posloupnosti. To znamená, že určí maximální k takové, že $X(i_1) < X(i_2) < \ldots < X(i_k)$ pro $1 \leqq i_1 < i_2 < \ldots < i_k \leqq N$. Zdůvodněte správnost algoritmu.

Příklad

Pro posloupnost 4 2 7 6 4 5 3 9 8 5 9 je $k = 5$, neboť maximální vybraná rostoucí podposloupnost 2 4 5 8 9 má délku 5.

P – II – 3

Je dán následující program:

PASCAL

```
const N = 100;
var A: array [1..N] of integer;
    J, K, L, R, X: integer;
begin
  for J := 1 to N do read(A[J]);
  L := 2;
  R := N;
  K := N;
  repeat
    for J := R downto L do
```

```
        if A[J − 1] < A[J] then
          begin
          X := A[J − 1]; A[J − 1] := A[J]; A[J] := X;
          K := J
          end;
      L := K + 1
      for J := L to R do
        if A[J − 1] < A[J] then
          begin
          X := A[J − 1]; A[J − 1] := A[J]; A[J] := X;
          K := J
          end;
      R := K − 1
    until L > R;
    for J := 1 to N do writeln(A[J])
end.
```

```
   BASIC
  10 LET N = 100
  20 DIM A(N)
  30 FOR J = 1 TO N
  40 INPUT A(J)
  50 NEXT J
  60 LET L = 2
  70 LET R = N
  80 LET K = N
  90 FOR J = R TO L STEP −1
 100 IF A(J−1) >= A(J) THEN GOTO 150
 110 LET X = A(J−1)
 120 LET A(J−1) = A(J)
```

```
130 LET A(J) = X
140 LET K = J
150 NEXT J
160 LET L = K+1
170 FOR J = L TO R
180 IF A(J-1) >= A(J) THEN GOTO 230
190 LET X = A(J-1)
200 LET A(J-1) = A(J)
210 LET A(J) = X
220 LET K = J
230 NEXT J
240 LET R = K-1
250 IF L <= R THEN GOTO 90
260 FOR J = 1 TO N
270 PRINT A(J)
280 NEXT J
```

Vstupem programu je 100 celých čísel.

a) Zjistěte a zdůvodněte, co je výsledkem práce programu.

b) Proveďte alespoň přibližný horní odhad rychlosti výpočtu uvedeného programu. Uvažujte přitom pouze operace porovnání dvou čísel. To znamená, že vaším úkolem je zjistit, kolik porovnání bude při výpočtu maximálně provedeno. Úlohu řešte obecně a výsledek vyjádřete v závislosti na hodnotě konstanty N.

P – II – 4

Studijní text k této úloze je shodný s textem pro soutěžní úlohu P–I–4 v domácím kole.

Soutěžní úloha

Napište co nejlepší program v programovacím jazyce zásobníkového počítače, který přečte zadanou posloupnost čísel a určí počet kladných sudých čísel a jejich součet.

P – III – 1

Je dáno pole celých čísel $S[1\,..\,N]$ a funkce F, která každému celému číslu přiřazuje celočíselnou hodnotu z intervalu od 1 do K (včetně obou mezí). Přitom hodnota K je podstatně menší než počet prvků N uložených v poli S. Navrhněte algoritmus, který přerovná prvky pole S tak, aby byly uspořádány vzestupně podle hodnot, které jim přiřazuje funkce F. Po přerovnání tedy musí platit pro každou dvojici prvků $S[i]$, $S[j]$, kde $1 \leqq i < j \leqq N$, že $F(S[i]) \leqq F(S[j])$. V algoritmu nesmíte použít žádnou další datovou strukturu o rozsahu úměrném velikosti N, je povoleno užít nanejvýš takový počet pomocných paměťových buněk, který je úměrný hodnotě K. Existuje algoritmus s lineární časovou složitostí vzhledem k velikosti N zadaného pole, tzn. algoritmus, který vykoná požadované přerovnání provedením nejvýše $p \cdot N$ operací pro nějaké pevné číslo p. Zdůvodněte správnost navrženého algoritmu.

P – III – 2

Funkce V dvou proměnných je definována pro všechna celá nezáporná čísla následujícím rekurzivním předpisem:

$$V(0,y) = y+1 \qquad \text{pro } y \geqq 0$$
$$V(x,0) = V(x-1,x-1) \qquad \text{pro } x > 0$$
$$V(x,y) = V(x-1,y+1) + V(x,y-1) \qquad \text{pro } x > 0,\ y > 0$$

a) Určete minimální počet různých funkčních hodnot funkce V, které je třeba vypočítat, máme-li určit hodnotu $V(m, n)$ pro daná čísla m, n. Výsledek vyjádřete v závislosti na hodnotách m, n.

b) Napište co nejlepší algoritmus pro výpočet hodnoty $V(m, n)$ a zdůvodněte jeho správnost. Předpokládejte, že nezáporná celá čísla m, n zadaná na vstupu jsou dostatečně malá a že tedy nenastane situace, že by hodnota $V(m, n)$ překročila maximální hodnotu zobrazitelnou v počítači při běžné práci s celočíselnou aritmetikou.

c) Jak se změní výsledek úlohy a), pokud pro $x > 0, y > 0$ bude hodnota funkce V definována předpisem

$$V(x, y) = V(x - 1, y + 1) + V(x - 1, y - 1)?$$

P – III – 3

Nejprve si zavedeme několik základních pojmů. Řekneme, že N-úhelník $A(1)A(2)\ldots A(N)$ je konvexní, jestliže všechny jeho vnitřní úhly jsou menší než 180 stupňů. Diagonálou konvexního N-úhelníku budeme rozumět každou úsečku spojující dva různé vrcholy N-úhelníku, které spolu nesousedí na obvodu (tj. nejsou spojené hranou N-úhelníku). Z každého vrcholu N-úhelníku tedy vychází celkem $N - 3$ diagonál. Triangulací konvexního N-úhelníku nazveme každý takový soubor jeho navzájem se neprotínajících diagonál, které rozdělují plochu N-úhelníku na samé trojúhelníky. O součtu délek všech diagonál, které tvoří triangulaci konvexního N-úhelníku, budeme hovořit jako o velikosti triangulace.

Zadání úlohy

Konvexní N-úhelník $A(1)A(2)\ldots A(N)$ je zadán kartézskými souřadnicemi svých vrcholů v rovině. Navrhněte co nejlepší algoritmus, který určí minimální velikost jeho triangulace. Zdůvodněte správnost navrženého algoritmu.

P – III – 4

Studijní text k této úloze je shodný s textem pro soutěžní úlohu P–I–4 v domácím kole.

Soutěžní úloha

a) Je zadána posloupnost celých čísel následujícího speciálního tvaru. Nejprve obsahuje několik čísel 10, potom několik čísel 20 a nakonec několik čísel 30. Přitom od každé z uvedených tří hodnot obsahuje alespoň jedno číslo.

Napište co nejlepší program v programovacím jazyce zásobníkového počítače, který zjistí, zda počet výskytů čísla 10 v zadané posloupnosti je roven počtu výskytů čísla 20 nebo počtu výskytů čísla 30 nebo zda počet výskytů čísla 20 je stejný jako počet výskytů čísla 30. Jestliže některá z uvedených tří rovností platí, tzn. obsahuje-li zadaná posloupnost shodný počet čísel alespoň dvou hodnot, program vypíše číslo 1, jinak je výsledkem 0.

Můžete předpokládat, že vstupní údaje jsou uvedeny správně podle zadání úlohy. Program tedy nemusí kontrolovat správnost zadané vstupní posloupnosti čísel.

b) Řešte tutéž úlohu bez použití příkazů SUB, MUL, DIV.

P – I – 1

Úlohu je možné řešit různými způsoby. Ukážeme si zde, jak lze řešení elegantně vysvětlit a zapsat pomocí rekurze.

Rozklady zadaného čísla N budeme vytvářet postupně po krocích. V každém kroku prodloužíme již vytvořenou část rozkladu o jednoho dalšího sčítance. Abychom zbytečně nevytvářeli rozklady lišící se pouze pořadím sčítanců (takové rozklady podle zadání úlohy nepovažujeme za různé), zvolíme si pevné uspořádání sčítanců v rozkladu podle velikosti. Všechny rozklady budeme vytvářet tak, aby posloupnost sčítanců byla neklesající. Pro usnadnění výkladu budeme nadále předpokládat, že jednotlivé sčítance vytvářeného rozkladu ukládáme do pole $A[1\,..\,N]$ a že P udává počet členů již vytvořené části rozkladu. Neustále tedy platí nerovnost $A[1] \leqq A[2] \leqq \ldots \leqq A[P]$.

Popíšeme nyní jeden krok našeho algoritmu. Předpokládejme, že již máme prvních P členů rozkladu zadaného čísla N, $P \geqq 0$, $A[1] \leqq A[2] \leqq \ldots \leqq A[P]$. K rozložení ještě zbývá taková hodnota M, že

$$A[1] + A[2] + \ldots + A[P] + M = N.$$

Chceme prodloužit vytvářený rozklad o jednoho sčítance, tj. o člen $A[P+1]$. Uvažujme, jakou hodnotu K může mít tento další sčítanec. Vzhledem k zavedenému uspořádání sčítanců v rozkladu podle velikosti musí být $K \geqq A[P]$. Položíme nejprve $K = A[P]$. Musíme rozlišit dva případy. Jestliže zbytek M již není možné rozložit ani na dva sčítance velké

alespoň jako K, celá zbývající hodnota M se musí použít jako další sčítanec a rozklad je tím vytvořen. V opačném případě se další výpočet rozkladu rozvětví do dvou cest podle toho, zda se ve vytvářeném rozkladu použije nebo nepoužije další sčítanec o hodnotě K. Obě tyto varianty musíme zpracovat. Pokud se použije, bude $A[P+1] = K$ a v dalším kroku budeme stejným způsobem vyšetřovat situaci s již vytvořeným úsekem rozkladu délky $P + 1$. Jestliže se další sčítanec hodnoty K v rozkladu nepoužije, budeme stejným způsobem vyšetřovat situaci s nezměněným již vytvořeným úsekem rozkladu délky P, ale s hodnotou K zvětšenou o 1.

Uvedený postup zapíšeme v programovacím jazyce Pascal snadno s použitím rekurzivní procedury R. Užití rekurze je zde vhodné, neboť odpovídá charakteru našeho algoritmu. Není ovšem nezbytné, algoritmus by bylo možné naprogramovat bez užití rekurze s jedním pomocným polem.

```
program ROZKLAD (input, output);

const MAX = 50;          {maximální vstupní hodnota N}

var A: array[1..MAX] of integer;
                         {ukládání rozkladů}
    N: integer;          {rozkládané číslo}

procedure R (P, K, M: integer);
  {P — počet členů rozkladu uložených v poli A,
   K — uvažovaná hodnota dalšího sčítance,
   M — hodnota, kterou je ještě třeba rozložit;
   invariant: A[1] + A[2] + ... + A[P] + M = N
         &  A[1] ≦ A[2] ≦ ... ≦ A[P] ≦ K
```

```
    procedura doplní rozklad délky P uložený v A na delší
    rozložením hodnoty M, přičemž A[P+1] ≥ K; dalším
    rekurzivním voláním sama sebe vytvoří všechna možná
    doplnění a vypíše je}

var I: integer;

begin {R}
if M < 2 * K then
  begin                          {zbytek M již nelze dále rozložit}
  A[P+1] := M;
  for I := 1 to P+1 do write(A[I] : 3);
  writeln
  end
else
  begin
  A[P+1] := K;                   {další sčítanec velikosti K se ...}
  R(P+1, K, M-K);                { ... buď uplatní v rozkladu ...}
  R(P, K+1, M)                   { ... nebo neuplatní v rozkladu}
  end
end; {R}

begin
read(N);                         {hodnota rozkládaného čísla}
R(0, 1, N)  {na začátku ještě nemáme žádný sčítanec,
             každý sčítanec musí mít hodnotu alespoň 1
             a zbývá ještě rozložit celé číslo N}
end.
```

P – I – 2

Algoritmus řeší úlohu postupným „obarvováním“ zadaných bodů. Na začátku výpočtu mají všechny body barvu 0 (nejsou obarveny). Během výpočtu každému z nich přiřadíme barvu 1 nebo barvu 2 podle toho, do které skupiny bude patřit. Existuje-li nějaké rozdělení bodů do dvou skupin podle zadání úlohy, budou po ukončení výpočtu všechny body obarveny a vzájemná vzdálenost libovolné dvojice bodů stejné barvy bude menší než t. Obarvení tedy bude udávat požadované rozdělení bodů do dvou skupin (jedno z možných, pokud existuje více různých takových rozdělení).

Popíšeme nyní postup obarvování. Začneme tím, že zvolíme jeden libovolný bod B a obarvíme ho na 1. Nyní projdeme všechny body, jejichž vzdálenost od B je větší nebo rovna t. Tyto body nesmějí být ve stejné skupině s bodem B a musí tedy mít opačnou barvu. Obarvíme je proto všechny na 2 a k bodu B si poznamenáme, že je „vyřešen“ (tj. je zajištěno, že nebude ve stejné skupině s nějakým bodem, který má od něj vzdálenost větší nebo rovnu t).

Obecný krok algoritmu vypadá následovně. Část bodů je již obarvena, z nich některé jsou vyřešeny. Vybereme libovolný obarvený bod, který dosud není vyřešen (opět ho pracovně označíme B). Pokud jsou všechny obarvené body vyřešeny, zvolíme za B libovolný dosud neobarvený bod a obarvíme ho třeba na 1 (zde na volbě barvy nezáleží, již vyřešené body si obarvení žádného dalšího bodu nevynucují a vlastně začínáme řešit úlohu od začátku se zbylými body). Neexistuje-li již žádný neobarvený bod, algoritmus

končí. Po zvolení bodu B projdeme všechny body, které mají od bodu B vzdálenost větší nebo rovnu t, a kontrolujeme jejich obarvení. Pokud takový bod není dosud obarven, přiřadíme mu opačnou barvu, než má B (nesmí být s bodem B ve stejné skupině). Jestliže již takový bod má opačnou barvu než B, je vše v pořádku a pokračujeme ve výpočtu. Pokud ovšem má barvu stejnou jako bod B, došlo k neřešitelnému konfliktu, stanoveným podmínkám obarvení nelze vyhovět. Požadované rozdělení bodů tedy neexistuje a algoritmus předčasně ukončí svoji práci. Nedošlo-li k tomuto předčasnému ukončení, označíme bod B za vyřešený. Celý postup se opakuje tak dlouho, dokud existuje nějaký nevyřešený bod.

Pro programovou realizaci algoritmu zavedeme dvě pracovní pole. V poli $B[1\,..\,n]$ je uložena barva každého z bodů. Evidenci vyřešených a nevyřešených bodů provedeme jiným způsobem, abychom činnost algoritmu urychlili (abychom si ušetřili práci s prohledáváním pole při výběru nějakého dosud nevyřešeného bodu). Pole $S[1\,..\,n]$ budeme proto používat jako zásobník, v němž budou uložena čísla těch vrcholů, které jsou obarveny a nejsou dosud vyřešeny.

```
program SKUPINY (input, output);

const MAX = 100; {maximální počet bodů}

var D: array[1..MAX, 1..MAX] of real;
                          {vzdálenosti bodů}
    B: array[1..MAX] of 0..2; {obarvení bodů}
    S: array[1..MAX] of 1..MAX;
                          {zásobník nevyřešených}
```

```
    SP: integer; {ukazatel do zásobníku}
    N: integer; {počet bodů}
    T: real; {mezní vzdálenost bodů}
    POC: integer; {počet obarvených bodů}
    I, J: integer; {pomocné proměnné}
begin

{Načtení vstupních údajů v pořadí N, D, T:}
read(N);
for I := 1 to N do
  for J := 1 to N do read(D[I, J]);
read(T);

{Inicializace proměnných — obarvení prvního bodu:}
SP := 1;
S[1] := 1;
B[1] := 1;
for I := 2 to N do B[I] := 0;
POC := 1;

{Vlastní výpočet:}
repeat
  while SP <> 0 do
    begin          {existuje obarvený nevyřešený bod}
    I := S[SP];    {vezmi jeden takový bod → I}
    SP := SP - 1;
    for J := 1 to N do  {vyřešení situace v bodě I:}
      if D[I, J] >= T then
                   {J musí mít jinou barvu než I!}
        if B[J] = 0 then
          begin
```

```
            B[J] := 3 - B[I]; {obarvení bodu J}
            POC := POC + 1;
            SP := SP + 1;
            S[SP] := J
            end
          else if B[J] = B[I] then
            begin
            SP := 0;        {konflikt v obarvení bodů I,
                             J → umělé ukončení výpočtu}
            POC := N + 1 {zvláštní nastavení pro označení
                          konfliktu — využívá se v závěru}
            end
        end;
      if POC < N then
        begin               {existuje neobarvený bod}
        I := 1;
        while B[I] > 0 do I := I + 1;
        B[I] := 1;      {vybereme jeden takový a obarvíme}
        POC := POC + 1;
        SP := 1;
        S[1] := I
        end
until (POC >= N) and (SP = 0);

{Výpis výsledků algoritmu:}
if POC = N then
  begin
  writeln('Rozdeleni bodu do skupin:');
  write('1. skupina:');
  for I := 1 to N do
```

```
    if B[I] = 1 then write(I : 4);
  writeln;
  write('2. skupina:');
  for I := 1 to N do
    if B[I] = 2 then write(I : 4);
  writeln
  end
else {tzn. POC = N + 1}
  writeln('Rozdeleni bodu neni mozne!')
end.
```

Správnost našeho algoritmu vyplývá z výše uvedeného rozboru. Postupným obarvováním bodů je zajištěno, že body mající vzájemnou vzdálenost větší než t se nemohou dostat do stejné skupiny. Přitom bod je obarven (a tím zařazen do jedné z vytvářených skupin) jedině tehdy, je-li to vynuceno jeho vzdáleností od nějakého jiného bodu již zařazeného do některé ze skupin, popř. tehdy, jsou-li všechny požadavky uspokojeny a přitom ještě zbývají neobarvené body (potom barvu jednoho dalšího bodu lze zvolit libovolně).

Výpočet podle uvedeného algoritmu je konečný, neboť v každém kroku je právě jeden z bodů označen jako vyřešený (v programu je odstraněn ze zásobníku S) a celý výpočet končí nejpozději po vyřešení všech bodů. Vykoná se tedy nejvýše N kroků výpočtu, kde N je pevně zadaný počet bodů.

Algoritmus má kvadratickou časovou složitost. V každém z N kroků výpočtu je vyřešen jeden bod. Přitom vyřešení každého bodu vyžaduje vyhodnotit jeho vzdálenost od

všech ostatních bodů (s případnými dalšími akcemi konstantní složitosti, jako je obarvování bodů apod.). V každém kroku se dále provádí výběr dalšího bodu ke zpracování. Tento výběr má ovšem také nejvýše lineární časovou složitost (v programu: hledání prvního neobarveného bodu, je-li zásobník prázdný). Celkově je tedy třeba provést maximálně počet operací úměrný hodnotě N^2.

P – I – 3

a) Nejprve ukážeme, jakých hodnot nabývají během výpočtu jednotlivé proměnné A, B, C, D, E, F. Vyjádříme hodnoty těchto proměnných po K průchodech **for**-cyklem v programu.

– Hodnota proměnné A se zvyšuje od -1 po 2. Proměnná A tedy nabývá postupně hodnot 1, 3, 5, ..., a po K průchodech cyklu má hodnotu $2K - 1$.

– V proměnné B se sčítají všechny dosud vypočtené hodnoty proměnné A. Po K průchodech má tedy hodnotu

$$\sum_{j=1}^{K}(2j-1) = 2*\sum_{j=1}^{K} j - \sum_{j=1}^{K} 1 = 2*\frac{K*(K+1)}{2} - K = K^2.$$

– Hodnota proměnné C se v každém kroku vytváří nově (bez ohledu na svou předchozí hodnotu) jako součin hodnot proměnné B a parametru cyklu I. Tedy po K-tém průchodu bude mít C hodnotu $K^2 * K = K^3$.

– V proměnné D se sčítají všechny dosud vypočtené hodnoty proměnné C. Po K průchodech má tedy hodnotu $\sum_{j=1}^{K} j^3$.

– V proměnné E se počítá součet všech hodnot parametru cyklu I. Po K průchodech cyklem má proto E hodnotu

$$\sum_{j=1}^{K} j = \frac{K * (K+1)}{2}.$$

– Hodnota proměnné F se v každém kroku výpočtu počítá nově jako kvadrát hodnoty proměnné E. Po K průchodech nabude F hodnoty

$$\frac{K^2 * (K+1)^2}{4}.$$

Po ukončení výpočtu, tzn. po N průchodech **for**-cyklem v programu, tedy proměnná D bude mít hodnotu $\sum_{j=1}^{N} j^3$ a proměnná F hodnotu

$$\frac{N^2 * (N+1)^2}{4}$$

Ukážeme, že tyto hodnoty se sobě rovnají. Důkaz rovnosti provedeme matematickou indukcí:

1. pro $N = 1$ rovnost zřejmě platí,

2. nechť platí dokazovaná rovnost pro $N = x$; ukážeme, že platí i pro $N = x + 1$:

$$\sum_{j=1}^{x+1} j^3 = \sum_{j=1}^{x} j^3 + (x+1)^3 = \begin{matrix}\text{podle} \\ \text{indukčního} \\ \text{předpokladu}\end{matrix} =$$

$$= \frac{x^2 * (x+1)^2}{4} + (x+1)^3 =$$

$$= \frac{(x+1)^2 * (x^2 + 4x + 4)}{4} = \frac{(x+1)^2 * (x+2)^2}{4}$$

Tedy i pro $N = x + 1$ dokazovaná rovnost skutečně platí. Tím je celé tvrzení dokázáno.

Závěr: Po ukončení výpočtu mají obě proměnné D a F stejnou výslednou hodnotu, a to

$$\frac{N^2 * (N+1)^2}{4}.$$

b) Program počítající výsledné hodnoty proměnných D a F co nejrychlejším způsobem využívá výsledku úlohy a):

```
var D, F, N: integer;
begin
  read(N);
  D := N * (N + 1);
  D := (D * D) div 4;
  F := D;
  writeln(D);
  writeln(F)
end.
```

Zbývá porovnat rychlost výpočtu programu uvedeného v zadání úlohy a našeho zrychleného. Podle původního programu se provádělo N průchodů **for**-cyklem a při každém průchodu 6 aritmetických operací. Program tedy pracoval v lineárním čase (měřeno v závislosti na vstupní hodnotě proměnné N) a prováděl celkem $6N$ operací. Náš nový program dosáhne zcela stejných výsledků výrazně rychleji. Pracuje totiž v konstantním čase bez ohledu na vstupní hodnotu N a provádí celkem pouze 4 aritmetické operace.

P – I – 4

A. a) Nalezení největšího ze zadaných čísel je snadné. Úlohu řeší následující jednoduchý program. Během výpočtu je v zásobníku uchováváno pouze jediné číslo, a sice hodnota dosud nalezeného maxima. Zadaná čísla ze vstupu není třeba vůbec ukládat do zásobníku.

```
IN
PUSH                    (uložení prvního čísla)
while not EOF do
  begin
  IN
  SUB                   (porovnání dalšího čísla ...)
  if POS then           (... s dosud nalezeným max.)
    begin
    ADD
    EXCH                (uložení nové hodnoty maxima)
    end
  end
TOP
OUT                     (tisk výsledného maxima)
```

b) Nalezení dvou největších ze zadaných čísel je možné provést v principu dvěma odlišnými způsoby. Ukážeme si je postupně oba. V první variantě řešení jsou všechna čísla čtená ze vstupu ukládána do zásobníku, přičemž na vrcholu zásobníku se stále udržuje největší z nich. Nalezené maximum se po přečtení všech čísel vytiskne a odstraní se ze zásobníku. V další fázi výpočtu se hledá maximum ze zbylých čísel (tzn. celkově druhé největší číslo) při postupném

vybírání čísel ze zásobníku. Hodnota tohoto maxima se přitom udržuje a průběžně aktualizuje v pracovním registru počítače.

```
IN
PUSH                    (uložení prvního čísla)
while not EOF do
  begin
  IN
  SUB                   (porovnání dalšího čísla ...)
  if POS then           (s dosud nalezeným maximem)
    begin
    ADD
    PUSH                (nová hodnota maxima na zás.)
    end
  else
    begin               (přečtené číslo není maximem)
    ADD
    EXCH                (uložení do zásobníku pod ...)
    PUSH                (... dosud nalezené maximum)
    end
  end
OUT                     (vytisknutí největšího čísla)
POP                     (smazání největšího čísla)
TOP
POP
while not EMPTY do      (hledání maxima ze zbylých č.)
  begin
  SUB                   (porovnání)
  if NEG  then TOP      (nová hodnota maxima do reg.)
```

```
        else ADD        (obnovení původní hodnoty max.)
  POP
  end
OUT                     (vypsání druhého nejv. čísla)
```

Ve druhém způsobu řešení se čísla zadaná na vstupu do zásobníku neukládají. V zásobníku jsou stále uložena (a průběžně aktualizována) pouze dvě čísla: největší z dosud přečtených čísel a druhé největší (to je na vrcholu).

```
IN
PUSH                    (uložení prvního čísla)
IN
SUB                     (porovnání prvních dvou čísel)
if POS then
  begin
  ADD
  EXCH
  end
else
  ADD
PUSH                    (připravena první dvě čísla
                         v zásobníku, menší navrchu)
while not EOF do
  begin
  IN
  SUB                   (porovnání dalšího čísla ...)
  if POS then           (... s druhým největším)
    begin
    ADD                 (je-li nové číslo větší, ...)
```

```
    POP                 (... dosavadní vrchol se smaže)
    SUB
    if POS then         (porovnání s dosud max. číslem)
      begin
      ADD
      EXCH
      end
    else
      ADD
    PUSH                (uložení ve správném pořadí ..)
    end                 (... do zásobníku)
  end
TOP
POP
EXCH                    (nejdříve vypíšeme největší ..)
OUT                     (... ze všech čísel)
TOP
OUT                     (vypsání druhého nejv. čísla)
```

c) Jestliže budeme do zásobníku našeho zásobníkového počítače ukládat pouze čísla čtená ze vstupu, popř. jiná čísla srovnatelné velikosti, úlohu nalézt tři největší ze zadaných čísel není možné řešit. Pro nalezení k největších čísel při jednom průchodu zadanou posloupností čísel je třeba mít k dispozici stále přístupných k paměťových míst. V případě zásobníkového počítače máme přímo přístupný pouze pracovní registr a vrchol zásobníku. Při čtení čísel ze vstupu je pracovní registr využíván pro čtení, takže je možné zároveň se čtením čísel ze vstupu a jejich ukládáním do zásobníku vyhledat pouze jedno největší ze zadaných čísel.

Při vybírání čísel ze zásobníku se využívá místo na vrcholu zásobníku, přes které se uskutečňuje přístup do zásobníku. Jako pracovní paměť zbývá k dispozici pouze registr, takže je možné vyhledávat opět pouze jedno, např. druhé největší ze zadaných čísel. Pokud bychom čtená čísla nechtěli do zásobníku vůbec ukládat, můžeme opět udržovat v zásobníku hodnoty pouze dvou největších čísel. Při třech bychom již nedokázali obnovovat potřebný stav zásobníku. Kdybychom totiž přečetli ze vstupu hodnotu nového maxima, nemáme kam odložit druhé a třetí největší číslo, abychom mohli uložit nové maximum na dno zásobníku.

Existuje ovšem řešení úlohy založené na možnosti zakódovat vhodným způsobem více čísel do jediného čísla, které pak uložíme do zásobníku. Na vrcholu zásobníku si tedy můžeme udržovat jediný záznam obsahující informaci o třech největších dosud nalezených číslech. Po přečtení každého dalšího čísla ze vstupu tento záznam rozkódujeme a po porovnání s nově přečtenou hodnotou opět zakódujeme (s případnými změnami) a uložíme. Možných kódů je celá řada, například lze použít součin příslušných mocnin tří pevně zvolených prvočísel. Trojici čísel a, b, c lze tedy uložit ve tvaru $2^a \cdot 3^b \cdot 5^c$.

Toto řešení je sice zcela správné, ale má dvě nevýhody. Je velmi pracné a komplikované, příslušný program je značně rozsáhlý, a proto ho zde ani neuvádíme. Druhou, významnější a důležitější problematickou otázkou je velikost čísel ukládaných do zásobníku. I to nejúspornější zakódování tří čísel běžné velikosti do jediného vede k velmi velké výsledné hodnotě. V definici zásobníkového počítače nebylo stanoveno žádné omezení na velikost čísel, která lze ukládat do

zásobníku, takže ve smyslu této definice je uvedené řešení správné. Je ovšem dobré uvědomit si, že takovéto řešení by bylo zcela nereálné v případě, že bychom programovali nějaký skutečně existující zásobníkový počítač, neboť u něj by velikost čísel, s nimiž se pracuje, byla jistě omezena.

B. Čísla ze vstupu postupně čteme a ukládáme do zásobníku tak dlouho, dokud nepřečteme prostřední prvek zadané posloupnosti (jednoznačně identifikovatelný tím, že jako jediný má hodnotu 0). Při čtení dalších čísel ze vstupu odebíráme čísla ze zásobníku a kontrolujeme, zda je splněna podmínka symetrie podle zadání.

```
IN
while not ZERO do
  begin
  PUSH
  IN
  end                          (první polovina posloupnosti
                                je uložena v zásobníku)
while not EOF do
  begin
  IN                           (přečtení dalšího čísla)
  SUB
  if ZERO then                 (při shodě s vrcholem zás. ...)
    POP                        (... snížíme úroveň zásobníku)
  else
    while not EOF do           (jinak dočteme čísla ze vstupu)
      IN
  end
if ZERO then CONST 1           (nastavení výsledné hodnoty)
```

else *CONST* 0

OUT (tisk výsledku)

P – II – 1

Zavedeme pomocné pole $SLMAX[1\,..\,N]$, které bude obsahovat informace o poloze maximálních hodnot v jednotlivých sloupcích pole A. Bude tedy platit $SLMAX[J] = I$ právě tehdy, jestliže $A[I, J]$ je největší ze všech čísel uložených v J-tém sloupci pole A.

Nejprve provedeme počáteční zaplnění pole $SLMAX$ odpovídajícími hodnotami. Výběr N největších čísel uložených v poli A potom proběhne v N krocích následujícího výpočtu:

– pomocí pole $SLMAX$ nalezneme největší hodnotu ze sloupcových maxim; tuto hodnotu získáme jako maximum z čísel $A[SLMAX[J], J]$ pro J od 1 do N; nechť je to číslo $A[I, K]$

– číslo $A[I, K]$ je tedy největším z čísel uložených v poli A; vytiskneme ho a vypustíme ho z pole A dosazením nuly za $A[I, K]$

– obnovíme informaci o poloze sloupcového maxima ve sloupci, v němž došlo ke změně, tzn. spočteme novou hodnotu $SLMAX[K]$.

program *MAXIMA* (input, output);

const $M = 100$; {maximální přípustná hodnota N}

var N: integer; {velikost zadané matice}
  A: **array** $[1\,..\,M, 1\,..\,M]$ **of** integer; {zadaná matice čísel}
  $SLMAX$: **array** $[1\,..\,M]$ **of** integer;

```
                                {polohy sloup. maxim}
  MAX: integer; {pro výběr max. hodnoty}
  I, J, K: integer; {pomocné proměnné}

procedure SLOUPEC(J:integer);
  {počítá polohu maximálního čísla v J-tém sloupci
  pole A a ukládá ji jako hodnotu SLMAX[J]
  do pom. pole SLMAX}
var MX, I: integer;
begin {SLOUPEC}
MX := 0;
for I := 1 to N do
  if MX < A[I, J] then
    begin
    MX := A[I, J];
    SLMAX[J] := I
    end
end; {SLOUPEC}

begin
read(N);
for I := 1 to N do
  for J := 1 to N do read(A[I, J]);
                                {čtení zadané matice A}
for J := 1 to N do SLOUPEC(J);
                                {nastavení hodnot SLMAX}
for I := 1 to N do {N kroků výpočtu:}
  begin
  MAX := 0;
  for J := 1 to N do {výběr největšího čísla}
    if A[SLMAX[J], J] > MAX then
```

```
      begin
      MAX := A[SLMAX[J], J];
                          {MAX je největší číslo}
      K := J {K je sloupec, odkud bylo
                            vybráno číslo MAX}
      end;
  writeln(MAX); {vypsání největšího čísla}
  A[SLMAX[K], K] := 0; {... a jeho smazání z A}
  SLOUPEC(K) {nová hodnota největšího
                          čísla ve sloupci K}
  end
end.
```

Správnost algoritmu přímo plyne z úvodního rozboru. V každém kroku výpočtu je nalezena a vytištěna největší hodnota z maxim v jednotlivých sloupcích, což je jistě největší číslo momentálně se nacházející v poli A. Přepsání tohoto čísla nulou je vytisknuté číslo z pole A vynecháno (všechna čísla v poli A jsou podle zadání kladná!) a v dalším kroku se tedy bude vyhledávat největší ze všech zbývajících čísel. Celkem program po N krocích výpočtu vytiskne skutečně N největších čísel uložených původně v poli A. Výpočet je konečný, má předem omezený počet kroků hodnotou N.

Popsaný algoritmus má kvadratickou časovou složitost. Přečtení N^2 čísel ze vstupu i počáteční zaplnění pole $SLMAX$ jistě vyžadují řádově N^2 operací. Vlastní výpočet je pak tvořen N kroky, přičemž v každém z nich je nejprve pomocí pole $SLMAX$ vybráno maximum z N čísel a po jeho vypsání a smazání je opět výběrem maxima z N čísel

obnoveno správné zaplnění pole $SLMAX$. Celkem se tedy provede počet operací úměrný hodnotě N^2.

Rychlejší algoritmus řešící zadanou úlohu není možný, neboť již jenom počet hodnot, které je třeba zpracovat a z nichž každá může ovlivnit výsledek, je N^2.

Poznámka. Zkuste sami modifikovat zde uvedené řešení úlohy tak, aby původní obsah pole A zůstal zachován.

P – II – 2

Postupně budeme procházet zadanou posloupnost čísel X. V i-tém kroku výpočtu budeme sledovat, jak mohou vypadat rostoucí podposloupnosti vybrané z počátečního úseku posloupnosti X délky i, tzn. z posloupnosti $X(1)$, $X(2)$, ..., $X(i)$. Pro dosažení co nejúspornějšího a nejrychlejšího řešení úlohy si zavedeme pomocné pole $M[1\,..\,N]$, do něhož si budeme průběžně ukládat následující informaci: prvek $M[j]$ je v každém okamžiku roven minimální dosud známé hodnotě posledního prvku vybrané rostoucí podposloupnosti délky j. Další průběžně aktualizovaná proměnná K udává délku maximální (tzn. nejdelší) dosud nalezené rostoucí podposloupnosti. V poli M jsou tedy definovány hodnoty $M[1]$, $M[2]$, ..., $M[K]$. Po provedení i-tého kroku výpočtu budou tudíž splněny následující podmínky:

1. $1 \leqq i \leqq N$
2. K je délka maximální rostoucí podposloupnosti vybrané z posloupnosti $X(1)$, $X(2)$, ..., $X(i)$
3. $M[j] = \min\{X(i_j)$; existují indexy $i_1 < i_2 < \ldots < i_j \leqq \leqq i$ takové, že $X(i_1) < X(i_2) < \ldots < X(i_j)\}$ ... pro $j = 1, \ldots, K$

Z poslední uvedené podmínky zřejmě plyne platnost nerovnosti $M[1] < \ldots < M[K]$. Pokud totiž rostoucí vybraná podposloupnost délky j může končit číslem $M[j]$, pak existuje také vybraná podposloupnost délky $j-1$, která vznikne z předchozí uvažované podposloupnosti vynecháním posledního členu. Její poslední člen bude ovšem jistě menší než $M[j]$, a tedy skutečně platí $M[j-1] < M[j]$.

Po provedení všech N kroků výpočtu bude proměnná K obsahovat délku maximální rostoucí podposloupnosti vybrané z celé zadané posloupnosti $X(1), \ldots, X(N)$, a právě to je požadovaný výsledek úlohy.

Zbývá ukázat, jakým způsobem provedeme aktualizaci hodnot proměnné K a údajů uložených v poli M při jednom kroku výpočtu. Uvažujme i-tý krok výpočtu a zpracování čísla $X(i)$ ze zadané posloupnosti. Je-li $X(i)$ větší než $M[K]$, je možné prodloužit dosud nejdelší nalezenou vybranou rostoucí podposloupnost o číslo $X(i)$. Zvětšíme tedy hodnotu proměnné K o jedničku a pro nové K definujeme údaj $M[K]$ jako hodnotu čísla $X(i)$. V opačném případě není možné dosud maximální vybranou podposloupnost prodloužit a hodnota K se tedy nezmění. Může se ovšem stát, že číslo $X(i)$ nám umožní snížit některou z dříve stanovených hodnot $M[j]$. Jak jsme již uvedli, platí stále nerovnost $M[1] < \ldots < M[K]$. Je tedy možné najít takový index j, že buď

$$j = 1 \qquad \text{a} \qquad X(i) \leqq M[1],$$

nebo

$$1 < j \leqq K \qquad \text{a} \qquad M[j-1] < X(i) \leqq M[j].$$

Nastane-li ostrá nerovnost $X(i) < M[j]$, můžeme nyní snížit hodnotu $M[j]$ tím, že za ni dosadíme číslo $X(i)$. Existuje totiž rostoucí vybraná podposloupnost délky $j-1$ končící číslem $M[j-1]$, a protože $X(i) > M[j-1]$, číslo $X(i)$ tuto podposloupnost prodlužuje na rostoucí vybranou podposloupnost délky j. Jejím posledním prvkem je právě číslo $X(i)$, které tedy sníží údaj $M[j]$, je-li to možné.

Uvedený rozbor je zároveň zdůvodněním správnosti navrženého algoritmu. Výpočet je jistě konečný, neboť je tvořen přesně N kroky, kde N je délka zpracovávané posloupnosti čísel. Časová složitost algoritmu je v optimálním případě $N * \log_2(N)$. Provádí se totiž N kroků výpočtu a v každém z kroků se kromě jednoduchých akcí s konstantní časovou složitostí musí vyhledávat v poli M index j určující, kterou hodnotu $M[j]$ budeme modifikovat. Vzhledem k uspořádání pole M podle velikosti je možné určit index j binárním prohledáváním (půlením intervalů), a tedy s časovou složitostí $\log_2(N)$. Odtud plyne složitost celého algoritmu $N * \log_2(N)$.

```
program PODPOSLOUPNOST (input, output);

const MAX = 100; {maximální délka posloupnosti}

var X: array[1..MAX] of integer;
                        {zadaná posloupnost čísel}
    M: array[1..MAX] of integer;
                        {pomocné pole dle rozboru}
    N: integer; {počet čísel v posloupnosti X}
    K: integer; {výsledná délka podposloupnosti}
    D, H: integer; {meze pro binární prohledávání}
```

```
    I, J: integer; {pomocné proměnné}

begin
read(N);
for I := 1 to N do read(X[I]); {přečtení posloupnosti}
I := 1;
K := 1;
M[1] := X[1];
while I < N do {N kroků výpočtu:}
  begin
  I := I + 1;
  if X[I] > M[K] then {zpracování čísla X[I]}
    begin
    K := K + 1; {prodloužení max. podposloupnosti}
    M[K] := X[I]
    end
  else
    begin {nalezení indexu J v poli M}
    if X[I] <= M[1] then
      J := 1
    else
      begin  {chceme 1 < J ≦ K
                & M[J - 1] < X[I] ≦ M[J]}
      D := 1;
      H := K;
      while H - D > 1 do
              {binární prohledávání v úseku D - H}
        begin {stále platí: M[D] < X[I] ≦ M[H]}
        J := (H + D) div 2;
        if X[I] > M[J] then D := J
```

```
                else H := J
            end;
        J := H {výsledná hodnota indexu J}
        end;
    M[J] := X[I] {nová hodnota M[J] pro nalezené J}
    end
  end;
write('Nejdelší vybraná rostoucí podposloupnost');
writeln(' má délku ', K, '.');
writeln
end.
```

P – II – 3

a) Výsledkem práce uvedeného programu je setřídění N čísel zadaných na vstupu podle velikosti od největšího k nejmenšímu a vytisknutí všech čísel v tomto sestupném pořadí. Jedná se o programovou realizaci třídicího algoritmu zvaného shake-sort, což je vylepšená varianta známého bublinkového třídění.

Proměnné L a R slouží k vyznačení úseku pole A, který je ještě třeba setřídit. V každém okamžiku platí, že musíme ještě třídit čísla uložená v úseku $A[L-1], \ldots, A[R]$, zatímco ostatní čísla v poli A jsou již na svých místech (tj. na místech, kam budou patřit i po setřídění celého pole A). Tomu odpovídá počáteční nastavení hodnot proměnných L a R, neboť na začátku výpočtu je třeba setřídit celé pole A. Třídění končí, jestliže $L > R$, tzn. jsou-li již všechna čísla v poli A na svých místech.

Celé třídění probíhá po krocích. Každý krok je v uvedeném programu představován provedením jednoho **for**-cyklu a následným nastavením některé z hodnot L, R. Pravidelně se střídají kroky výpočtu, v nichž je dosud neutříděný úsek pole A procházen pomocí proměnné J od nižších indexů k vyšším a naopak. Je-li při takovém průchodu nalezena dvojice sousedních čísel $A[J-1]$, $A[J]$, pro kterou platí $A[J-1] < A[J]$, jsou tato dvě čísla mezi sebou prohozena (vymění si místa v poli A). Do proměnné K je zároveň zaznamenáno místo poslední výměny dvou sousedních čísel během jednoho průchodu polem A. Je tudíž jisté, že po ukončení průchodu dosud nesetříděným úsekem pole A se dostane na svoje správné místo v poli A nejméně jedno další číslo. Při průchodu ve směru od nižších indexů k vyšším je to nejmenší z čísel nacházejících se v tomto úseku (dostane se na konec úseku), při opačném směru je to největší z čísel (dostane se na začátek). Je tedy možné změnit hodnotu příslušné proměnné L nebo R a tím zmenšit úsek pole A, který je ještě třeba setřídit. V některých případech je možné zmenšit sledovaný úsek pole A i o více čísel najednou. Na svých místech jsou totiž jistě všechna čísla od místa poslední provedené výměny dvou sousedních čísel až k odpovídajícímu konci úseku. Správné změny hodnoty proměnné L nebo R proto dosáhneme přiřazovacím příkazem za **for**-cyklem využívajícím hodnoty proměnné K.

Výpočet programu je pro libovolná vstupní data jistě konečný, neboť v každém kroku výpočtu se dostane na své správné místo v poli A nejméně jedno číslo a úsek, který je ještě třeba setřídit, se tedy zmenší. Počet kroků výpočtu je proto předem omezen hodnotou konstanty N.

b) Program třídí celkem N čísel. V každém kroku výpočtu se dostane na své správné místo v poli A minimálně jedno číslo, takže k setřídění celého pole je třeba provést maximálně $N - 1$ kroků (bude-li na svých místech $N - 1$ čísel, pak poslední N-té číslo již také). V každém kroku výpočtu se úsek pole A, který je ještě třeba setřídit, zmenšuje alespoň o jedno číslo. V prvním kroku má délku N, v nejhorším případě má výpočet plných $N - 1$ kroků a potom v posledním kroku má tříděný úsek délku 2. Při průchodu úsekem délky D v jednom kroku výpočtu se provede $D - 1$ porovnání dvou sousedních čísel. Celkově proto bude při výpočtu provedeno maximálně

$$(N-1)+(N-2)+\ldots+3+2+1=\frac{N(N-1)}{2}$$

porovnání. Pro konstantu $N = 100$ zvolenou v našem programu představuje tento výraz celkem 4 950 porovnání. Třídicí algoritmus shake-sort má tedy kvadratickou časovou složitost.

P – II – 4

Program bude číst ze vstupu postupně zadaná čísla a bude z nich vybírat ta, která jsou kladná a sudá. Otestovat, zda je číslo kladné, můžeme snadno pomocí jednoduché podmínky POS. Ke zjištění, jestli je číslo sudé, musíme naprogramovat test dělitelnosti dvěma. Vzhledem k tomu, že zásobníkový počítač pracuje v celočíselné aritmetice, stačí vstupní číslo vydělit dvěma, zpětně vynásobit dvěma a porovnat s jeho původní hodnotou (kterou jsme si předem uschovali na zásobníku).

S vybranými kladnými sudými čísly je třeba provádět dvě operace – určovat jejich počet a jejich součet. To je možné provádět v zásadě dvěma různými způsoby, podobně jako u úlohy P–I–4 Ab). První možností je ukládat vybraná čísla průběžně do zásobníku. Přitom musíme jednu z výsledných hodnot (např. počet čísel) počítat zároveň s jejich ukládáním do zásobníku a druhou hodnotu (součet čísel) potom spočteme během vyprazdňování zásobníku. Druhá metoda řešení úlohy spočívá v tom, že se čísla čtená ze vstupu do zásobníku vůbec neukládají. Během celého výpočtu se v zásobníku udržují pouze dvě hodnoty, a to součet a počet již zpracovaných kladných sudých čísel. Po přečtení všech čísel ze vstupu pak máme v zásobníku oba hledané údaje.

Ukážeme si nejprve podrobné řešení naší úlohy využívající první z uvedených postupů. V první etapě výpočtu tedy čteme čísla ze vstupu, provedením potřebných testů z nich vybíráme kladná sudá a tato čísla ukládáme do zásobníku. Na vrcholu zásobníku zároveň udržujeme hodnotu udávající počet čísel uložených do zásobníku. Po přečtení všech čísel ze vstupu tuto hodnotu vytiskneme a odstraníme ze zásobníku. Ve druhé etapě výpočtu potom čísla ze zásobníku postupně odebíráme a v pracovním registru počítáme jejich součet. Ten pak vytiskneme jako další výsledek našeho programu.

```
CONST 0
PUSH                    (v zásobníku zatím není žádné
                         ze vstupních čísel)
while not EOF do
  begin
```

```
  IN                    (přečtení dalšího čísla)
  if POS then
    begin               (číslo je kladné)
    PUSH
    PUSH
    CONST 2
    EXCH
    DIV                 (test dělitelnosti dvěma)
    MUL
    POP
    SUB
    if ZERO then        (číslo je sudé)
      begin
      TOP               (záměna vrchních dvou čísel ..)
      POP               (.. v zásobníku, na vrchol ..)
      EXCH              (.. se dostane počet čísel)
      PUSH
      CONST 1
      ADD               (zvětšení počtu čísel o 1)
      EXCH              (uložení nového počtu čísel)
      end
    else
      POP               (číslo není sudé — smažeme ho)
    end
  end
TOP
OUT                     (vypsání počtu čísel)
POP
CONST 0
while not EMPTY do
```

```
  begin
  ADD                    (součet hodnot čísel ze zás.)
  POP
  end
OUT                      (vypsání součtu čísel)
```

Při druhém z možných postupů si budeme na vrcholu zásobníku uchovávat součet již přečtených kladných sudých čísel a pod ním jejich počet. Po nalezení dalšího kladného sudého čísla tyto dva údaje zaktualizujeme. Na závěr výpočtu program vypíše obě čísla uložená v zásobníku.

```
CONST 0
PUSH                     (zatím počet = 0)
PUSH                     (zatím součet = 0)
while not EOF do
  begin
  IN                     (přečtení dalšího čísla)
  if POS then
    begin                (číslo je kladné)
    PUSH
    PUSH
    CONST 2
    EXCH
    DIV                  (test dělitelnosti dvěma)
    MUL
    POP
    SUB
    if ZERO then         (číslo je sudé)
      begin
```

```
      TOP
      POP          (přesun čísla do registru)
      ADD          (přičtení čísla k součtu)
      POP          (odstranění dosavadního součtu)
      EXCH         (počet čísel do registru)
      PUSH         (počet čísel do zásobníku)
      CONST 1
      ADD          (zvýšení počtu čísel o 1)
      POP          (odstranění dosavadního počtu)
      EXCH
      PUSH         (obnovení situace v zásobníku)
      end
    else
      POP          (číslo není sudé — smažeme ho)
    end
  end
TOP
POP
OUT                (vypsání součtu čísel)
TOP
POP
OUT                (vypsání počtu čísel)
```

P – III – 1

Algoritmus řešící zadanou úlohu bude založen na následujícím postupu. Nejprve si rozdělíme celé pole S na K „přihrádek", přičemž každá přihrádka je určena vždy pro čísla, jimž funkce F přiřazuje jednu z hodnot od 1 do K. Velikosti těchto přihrádek není těžké předem zjistit při jednom

průchodu polem S. Pro uložení velikostí přihrádek budeme používat pomocné pole $C[1..K]$. Ve druhé fázi výpočtu pak budeme přihrádky zaplňovat těmi prvky pole S, které do nich patří. Přitom je třeba zaznamenávat si, kam až je která přihrádka zaplněna. K tomuto účelu bude sloužit druhé pomocné pole $A[1..K]$. Obě pomocná pole C a A mají v souladu se zadáním úlohy velikost úměrnou hodnotě K. Zbývá vyřešit technickou otázku, jak celý proces zařazování čísel do přihrádek organizovat, aby třídění probíhalo „na místě" bez nutnosti pracovat s dalším polem velikosti N. Tímto problémem se budeme podrobněji zabývat v dalším detailním popisu algoritmu.

V první fázi výpočtu provedeme dosazení počátečních hodnot do polí C a A. Toto dosazení se snadno uskuteční při jednom sekvenčním průchodu polem S. Po inicializaci polí C a A bude $C[i]$ určovat počet všech čísel uložených v S, kterým funkce F přiřazuje hodnotu i. Údaj $A[i]$ bude mít význam indexu, od kterého budou po uspořádání čísel v poli S uložena ta z nich, jimž funkce F přiřazuje hodnotu i (tzn. jsou to začátky jednotlivých „přihrádek" v poli S).

Druhou fází výpočtu je pak vlastní uspořádání čísel. Přerovnávání čísel uložených v poli S probíhá po krocích. V každém kroku je jedno z N čísel přemístěno na své správné místo (tj. na první volné místo v té „přihrádce", do které toto číslo patří). Přitom se průběžně mění hodnoty uložené v polích C a A. V průběhu této druhé fáze výpočtu bude hodnota $C[i]$ udávat (pro každý index i z rozmezí od 1 do K) počet čísel uložených v poli S takových, že jim funkce F přiřazuje právě hodnotu i a že tato čísla dosud nebyla zařazena na své výsledné místo v poli S. Hodnota $A[i]$ bu-

de indexem v poli S, kam se má umístit další přemisťované číslo, jemuž funkce F přiřazuje hodnotu i.

Začneme umísťováním čísla $S[j]$ s indexem $j = 1$. Toto číslo patří do pole S na místo s indexem $A[F(S[j])]$. Umístíme ho tam proto a zvýšíme hodnotu $A[F(S[j])]$ o 1, aby se stala indexem volného místa v poli S, kam lze uložit další číslo, jemuž funkce F přiřazuje stejnou hodnotu $F(S[j])$. Zároveň snížíme hodnotu $C[F(S[j])]$ o 1, neboť jedno číslo s hodnotou $F(S[j])$ je již umístěno. Jestliže je $A[F(S[j])]$ různé od j, nesmíme ztratit číslo, které bylo dosud uloženo v poli S na místě s indexem $A[F(S[j])]$. Toto číslo proto přesuneme výměnou na uvolněnou pozici v poli S s indexem j. Tím jsme zároveň, aniž bychom změnili hodnotu j, získali nové číslo $S[j]$, které budeme umísťovat v dalším kroku výpočtu. Pokud bylo náhodou číslo $S[j]$ již na svém místě, tj. pokud bylo $j = A[F(S[j])]$, žádný přesun čísel v poli S v tomto kroku neprovádíme a pouze změníme hodnoty uložené v polích A a C výše uvedeným způsobem. V tomto případě ale ještě musíme změnit hodnotu indexu j a určit tak nové číslo $S[j]$, které se bude umisťovat na své místo v dalším kroku výpočtu. Vyhledáme proto takové m, že $C[m]$ je nenulové, tzn. ještě je třeba umístit nějaké číslo, jemuž funkce F přiřazuje hodnotu m. Polohu takového čísla v poli S neznáme, ale víme, že po uspořádání patří na místo s indexem $A[m]$. Tedy prvek s indexem $j = A[m]$ není ještě umístěn a zvolíme ho proto pro další krok výpočtu. V každém kroku výpočtu se jedno z čísel dostane na své správné místo v poli S, takže po provedení N kroků bude na svých místech všech N čísel uložených v poli S.

Z uvedeného rozboru přímo plyne správnost popsaného

algoritmu. Konečnost výpočtu je dána provedením předem známého počtu N kroků výpočtu. Algoritmus má lineární časovou složitost z hlediska parametru N, jak požaduje zadání úlohy. Počáteční obsazení polí C a A vyžaduje provést jeden sekvenční průchod polem S, tedy řádově N operací. Vlastní výpočet má přesně N kroků, přičemž počet operací prováděných v každém kroku je omezen konstantou. Za konstantu zde považujeme i hodnotu K, která je podle předpokladu uvedeného v zadání úlohy podstatně menší než počet prvků N uložených v poli S.

```
program PREROVNANI (input, output);
const N = 100; {velikost pole S}
      K = 10; {počet různých hodnot funkce F}

var  S: array[1 .. N] of integer; {zadané pole čísel}
     A, C: array[1 .. K] of integer;
                                  {prac. pole podle rozboru}
     I, J, D, M, H, P, T: integer; {pomocné proměnné}

function F (X: integer): integer; {zadaná funkce F}
begin
if X >= 0 then {... může vypadat třeba takto}
  F := X mod K + 1
else
  F := 1
end;

begin
{Přečtení čísel do pole S a inicializace polí C a A:}
for I := 1 to N do read(S[I]);
for I := 1 to K do C[I] := 0;
```

```
for I := 1 to N do C[F(S[I])] := C[F(S[I])] + 1;
D := 1; {pro výpočet počátečních hodnot A}
for I := 1 to K do
  begin
  A[I] := D;
  D := D + C[I]
  end;

{Přerovnání čísel v poli S:}
J := 1; {index umísťovaného prvku v poli S}
M := 1; {minimální index v poli C, že C[M] > 0}
for I := 1 to N do
  begin
  H := F(S[J]); {hodnota funkce F umísťovaného čísla}
  P := A[H]; {nová pozice v S pro umísť. číslo}
  A[H] := A[H] + 1;
  C[H] := C[H] - 1;
  if P <> J then
    begin {výměna čísel na pozicích J a P v poli S}
    T := S[J];
    S[J] := S[P];
    S[P] := T
    end
  else
    begin {číslo S[j] je již na svém místě}
    while (C[M] = 0) and (M < K) do M := M + 1;
    J := A[M] {vybráno další číslo, které je třeba
                        umístit v následujícím kroku}
    end
  end;
```

```
{Výpis setříděného pole S:}
writeln('Setříděné pole čísel:');
for I := 1 to N do write(S[I],' ');
writeln
end.
```

P – III – 2

a) Z definice funkce V je patrné, že k vyjádření hodnoty $V(m, n)$ není třeba znát žádnou hodnotu $V(x, y)$ pro $x > m$. Budeme proto postupně zjišťovat počet potřebných hodnot $V(x, y)$ pro $x = m$, $x = m - 1$, $x = m - 2$, ..., $x = 0$. Uvažujme nejprve vzájemné závislosti různých hodnot funkce V v případě, že oba argumenty funkce jsou kladné (tzn. podle třetího řádku definice funkce V).

Ke stanovení hodnoty $V(m, n)$ je třeba znát $V(m, n-1)$, k určení $V(m, n-1)$ potřebujeme $V(m, n-2)$, ... atd., až ke stanovení $V(m, 1)$ potřebujeme znát $V(m, 0)$. Tedy pro $x = m$ je nutné spočítat celkem $n + 1$ hodnot funkce V, a sice hodnoty $V(m, y)$ pro všechna y od 0 do n. K výpočtu hodnoty $V(m, n)$ dále podle definice funkce V musíme znát hodnotu $V(m-1, n+1)$. Určení této hodnoty ze stejných důvodů jako v případě $x = m$ vyžaduje znát všechny hodnoty $V(m-1, y)$ pro y od 0 do $n+1$, tedy celkem $n+2$ hodnot funkce V. Mezi těmito $n + 2$ hodnotami jsou již obsaženy také všechny hodnoty, které potřebujeme znát pro výpočet $V(m, n-1)$, $V(m, n-2)$, ..., $V(m, 1)$. Zbývá vyřešit případ hodnoty $V(m, 0)$, k jejímuž výpočtu potřebujeme znát $V(m-1, m-1)$. Pro stanovení poslední uvedené hodnoty musíme opět ze stejných důvodů znát všechny hodnoty

$V(m-1, y)$ pro y od 0 do $m-1$. Zavedeme-li označení $d = \max(n+1, m-1)$, lze tedy počet potřebných hodnot funkce V pro $x = m-1$ vyjádřit jako $d+1$. Jsou to hodnoty $V(m-1, y)$ pro y od 0 do d.

S postupně klesající hodnotou prvního argumentu funkce V nyní počet hodnot této funkce, které je třeba spočítat, stále roste po jedné. Pro výpočet čísla $V(m-1, d)$ totiž potřebujeme znát $V(m-2, d+1)$, a tedy také všechny $V(m-2, y)$ pro y od 0 do $d+1$ atd. Problémy nyní nedělá ani výpočet hodnot $V(x, 0)$ pro $x < m$. Hodnota $V(m-1, 0)$ je stanovena definicí funkce V jako $V(m-2, m-2)$ a jistě $m-2 < d+1$. Pro $x = 0$ budeme tedy potřebovat znát hodnoty $V(0, y)$ pro y od 0 do $d+m-1$, tzn. celkem $d+m$ hodnot. Tyto hodnoty jsou již podle definice funkce V přímo dány a neodkazují se na žádné další hodnoty.

Počet všech potřebných hodnot funkce V pro výpočet $V(m, n)$ jsme tedy stanovili takto:

pro $x = m$ ... $n+1$ hodnot funkce V
pro $x = m-1$ ... $d+1$ hodnot funkce V
pro $x = m-2$ ... $d+2$ hodnot funkce V
.....
pro $x = 1$ ... $d+m-1$ hodnot funkce V
pro $x = 0$ ... $d+m$ hodnot funkce V

Zbývá provést jednu malou ne příliš významnou korekci. V případě $x = 0$ není třeba počítat hodnotu $V(0, 1)$, neboť k vyjádření $V(1, 0)$ potřebujeme znát $V(0, 0)$ a k vyjádření hodnot $V(1, 1)$, $V(1, 2)$, ..., $V(1, d+m-2)$ potřebujeme po řadě $V(0, 2)$, $V(0, 3)$, ..., $V(0, d+m-1)$. Od výsledného počtu potřebných hodnot V tedy můžeme odečíst 1.

Celkový počet potřebných hodnot funkce V nyní získáme součtem:

$$\begin{aligned}(n+1)+(d+1)+(d+2)+\ldots+\\ +(d+m-1)+(d+m)-1=\\ =n+md+\tfrac{1}{2}m(m+1),\end{aligned}$$

kde $d=\max(n+1,m-1)$.

Uvedený vzorec platí pro výpočet libovolné hodnoty $V(m,n)$, pokud $m>0$. V případě $m=0$ stačí samozřejmě spočítat pouze jedinou (požadovanou) hodnotu funkce V.

b) Pro výpočet hodnoty $V(m,n)$ musí algoritmus postupně spočítat všechny potřebné hodnoty funkce V, jejichž celkový počet jsme stanovili v řešení úlohy a). Je třeba zvolit takové pořadí výpočtu hodnot funkce V, abychom v každém kroku používali pouze již spočtené hodnoty. Z definice funkce V je zřejmé, že toto vhodné pořadí je následující: počítáme hodnoty $V(x,y)$ pro x rostoucí od 0 do m a pro každé takové x postupně pro y rostoucí od 0 do $d+m-x$ (až v případě $x=m$ stačí pouze pro y od 0 do n).

Je třeba ještě uvážit, které předchozí hodnoty funkce V musí algoritmus uchovávat a jakou datovou strukturu tedy bude vhodné použít. Podle definice funkce V potřebujeme znát při výpočtu $V(x,y)$ pro $x>0$ pouze některou z hodnot $V(x-1,z)$, kde $z\geqq y$ (viz 2. a 3. řádek definice), a případně ještě údaj $V(x,y-1)$. Pro uložení starších hodnot funkce V nám proto bude stačit jednorozměrné pole $P[0\,..\,d+m]$, ve kterém budeme spočtené hodnoty funkce V postupně přepisovat přes sebe. V každém okamžiku, tj. při výpočtu čísla

$V(x, y)$, mají uložené hodnoty $P[j]$ následující význam:

pro $j < y$ ... $P[j]$ má hodnotu $V(x, j)$
pro $j \geqq y$ ... $P[j]$ má hodnotu $V(x - 1, j)$.

Algoritmus výpočtu $V(m, n)$ nyní zapíšeme v programovacím jazyce Pascal. Pro zjednodušení zápisu algoritmu budeme i pro $x = m$ počítat všechny hodnoty $V(x, y)$ pro y od 0 až do d.

```
program FUNKCE V (input, output);

const MAX = 100;          {maximální velikost pole P,
                           tzn. maximální přípustná hodnota
                           pro m + n + 1 a pro 2m − 1}

var P: array[0 .. MAX] of integer;
                          {zákl. datová struktura}
    M, N: integer; {argumenty ze vstupu}
    D: integer; {hodnota dle rozboru}
    I, J: integer; {pomocné proměnné}

begin
read(M, N);
if M = 0 then
  writeln(N + 1)
else
  begin
  if N + 1 > M − 1 then D := N + 1
          else D := M − 1; {vypočtená hodnota d}
  for J := 0 to D + M do P[J] := J + 1;
                          {výpočet čísel V(1, y)}
```

```
  for I := 1 to M do {zbývající hodnoty V}
    begin
    P[0] := P[I - 1];
    for J := 1 to D + M - I do
                              P[J] := P[J - 1] + P[J + 1]
    end;
  writeln(P[N])
  end
end.
```

c) Při uvedené změně v definici funkce V se počet hodnot funkce V potřebný k výpočtu hodnoty $V(m, n)$ sníží. Přesné stanovení tohoto počtu ovšem bude o něco obtížnější. Rozlišíme tři základní případy:

1. $m \leqq n$

Pro výpočet hodnoty $V(m, n)$ potřebujeme znát $V(m - 1, n - 1)$ a $V(m - 1, n + 1)$, k jejich vyjádření musíme zase použít hodnoty $V(m - 2, n - 2)$, $V(m - 2, n)$, $V(m - 2, n+2)$, ... atd. Vzhledem k předpokladu $m \leqq n$ se nikdy nedostaneme k hodnotě $V(x, 0)$ pro $x > 0$ a nikdy se proto neuplatní druhý řádek v definici funkce V v zadání úlohy. Bude tudíž třeba spočítat

pro $x = m$ ... 1 hodnotu funkce V
pro $x = m - 1$ ... 2 hodnoty funkce V
pro $x = m - 2$ ... 3 hodnoty funkce V
.....
pro $x = 0$ ... $m + 1$ hodnot funkce V

tedy celkem $1 + 2 + \ldots + (m + 1) = \frac{1}{2}(m + 1)(m + 2)$ hodnot funkce V.

2. $m > n$ a čísla m, n mají různou paritu (tzn. jedno je sudé a druhé liché).

Při výpočtu hodnoty $V(m,n)$ zjistíme, že potřebujeme znát $V(m-n,0)$. Uvažujme nejprve potřebný počet hodnot funkce V bez určování počtu hodnot nutných k vyjádření $V(x,0)$ pro $x > 0$ (podle 2. řádku definice). Až do hodnot s $x = m - n$ je situace stejná jako v případě 1. Je třeba spočítat

$$\begin{aligned}&\text{pro } x = m \quad \ldots \text{ 1 hodnotu funkce } V\\ &\text{pro } x = m-1 \ldots \text{ 2 hodnoty funkce } V\\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\ &\text{pro } x = m-n \ldots \ n+1 \text{ hodnot funkce } V.\end{aligned}$$

Počet potřebných hodnot funkce V se dále nebude zvyšovat o 1 s každým snížením prvního argumentu funkce V, ale pouze při každém druhém snížení. Budeme totiž počítat hodnoty

$$\begin{aligned}V(m-n,y) &\text{ pro } y = 0, 2, 4, \ldots, 2n\\ V(m-n-1,y) &\text{ pro } y = 1, 3, 5, \ldots, 2n+1\\ V(m-n-2,y) &\text{ pro } y = 0, 2, 4, \ldots, 2n+2\\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\ V(0,y) &\text{ pro } y = 1, 3, 5, \ldots, 2n+(m-n) = m+n,\end{aligned}$$

neboť m, n mají různou paritu. Celkem je tedy třeba určit

$$\begin{aligned}&1+2+\ldots+n+\\ &+2[(n+1)+(n+2)+\ldots+\tfrac{1}{2}(m+n+1)] =\\ &= \tfrac{1}{2}n(n+1)+\tfrac{1}{4}(m+3n+3)(m-n+1)\end{aligned}$$

hodnot funkce V.

Zbývá vyřešit výpočet hodnot $V(x,0)$ pro $x > 0$. Je třeba spočítat $V(m-n,0)$, $V(m-n-2,0)$, ..., $V(1,0)$. Podle definice je třeba k určení $V(m-n,0)$ spočítat $V(m-n-1, m-n-1)$. Zatímco všechny dosud vyjadřované hodnoty funkce V měly argumenty různé parity, v tomto případě mají oba argumenty stejnou paritu a také k vyjádření hodnoty $V(m-n-1, m-n-1)$ budeme potřebovat samé další hodnoty se stejnou paritou obou argumentů funkce V. Jedná se tedy o rozdílné hodnoty, než jaké jsme dosud počítali. Jejich počet určíme snadno podle již vyřešeného 1. případu. Bude jich třeba $\frac{1}{2}(m-n)(m-n+1)$. Mezi těmito hodnotami jsou již obsaženy také všechny hodnoty potřebné k vyjádření čísel $V(m-n-2,0)$, ..., $V(1,0)$. Celkově je tedy nutné při výpočtu hodnoty $V(m,n)$ určit

$$\frac{1}{2}n(n+1)+\frac{1}{4}(m+3n+3)(m-n+1)+\frac{1}{2}(m-n)(m-n+1)$$

různých hodnot funkce V.

3. $m > n$ a čísla m, n mají stejnou paritu (tzn. jsou obě sudá nebo obě lichá).

Tento případ je velmi podobný předchozímu. Opět bude třeba spočítat pro $x = m, m-1, \ldots, m-n$ po řadě $1, 2, \ldots, n+1$ hodnot funkce V. Konkrétně pro $x = m-n$ jsou to opět hodnoty $V(m-n, y)$ pro $y = 0, 2, \ldots, 2n$. Výpočet bude tentokrát končit hodnotami $V(0,y)$ pro $y = 0, 2, \ldots, m+n$, neboť m, n mají nyní stejnou paritu. Bude tedy třeba určit

$$1+2+\ldots+n+$$
$$+2[(n+1)+(n+2)+\ldots+\tfrac{1}{2}(m+n)]+\tfrac{1}{2}(m+n)+1=$$

$= \frac{1}{2}n(n+1) + \frac{1}{4}(m+3n+2)(m-n) + \frac{1}{2}(m+n) + 1$

hodnot funkce V.

Opět zbývá dořešit výpočet hodnot $V(x,0)$ pro $x > 0$. Budou se počítat čísla $V(m-n,0)$, $V(m-n-2,0)$, ..., $V(2,0)$. Stejně jako ve 2. případě je třeba spočítat hodnotu $V(m-n-1, m-n-1)$. Mezi hodnotami k tomu potřebnými budou již hodnoty funkce V nezbytné k vyjádření všech ostatních čísel $V(m-n,0)$, ..., $V(2,0)$. Počet těchto hodnot můžeme opět vyjádřit pomocí výsledku 1. řešeného případu výrazem $\frac{1}{2}(m-n)(m-n+1)$.

Situace je nyní ovšem trochu komplikovanější tím, že jak tyto přidávané, tak i dříve spočítané hodnoty funkce V mají argumenty stejné parity, takže některé z hodnot jsou započítány v obou počtech. Zjistíme, kolik takových případů nastalo, a od celkového součtu je jednou odečteme. Z geometrického vyjádření obou množin hodnot v rovině snadno odvodíme, že se jedná o hodnoty nezbytné k vyjádření čísla $V(\frac{1}{2}(m+n), \frac{1}{2}(m+n))$, což je celkem $\frac{1}{2}[\frac{1}{2}(m+n)+1][\frac{1}{2}(m+n)+2]$ různých hodnot.

Pro vyjádření $V(m,n)$ je tedy v tomto případě nutné spočítat

$$\frac{1}{2}n(n+1) + \frac{1}{4}(m+3n+2)(m-n) + \frac{1}{2}(m+n) + 1 + \\ + \frac{1}{2}(m-n)(m-n+1) - \frac{1}{4}(m+n+2)(m+n+4)$$

různých hodnot funkce V.

P – III – 3

Předpokládejme, že proměnná N udává počet vrcholů zadaného N-úhelníku a že v polích AX, AY jsou uloženy

x-ové a y-ové souřadnice vrcholů $A(1), \ldots, A(N)$. Indexy vrcholů $1, 2, \ldots, N$ budeme uvažovat uspořádané v tomto pořadí, jak za sebou následují na obvodu N-úhelníku. Po vrcholu $A(N)$ následuje opět vrchol $A(1)$, uspořádání je tedy cyklické. V našem algoritmu budeme chtít pracovat s indexem vrcholu, který následuje jako d-tý za vrcholem s indexem i. Jeho pořadové číslo je $i+d$, pokud ovšem hodnota tohoto součtu nepřekročí N. Jinak je třeba vzhledem k cyklickému uspořádání index výsledného vrcholu počítat pomocí operace modulo N. Tento přepočet nám bude provádět pomocná funkce R, která je definována předpisem $R(k) = (k-1) \bmod N + 1$. Tedy v pořadí d-tým vrcholem následujícím za $A(i)$ je vrchol $A(R(i+d))$.

Vrcholy N-úhelníku jsou zadány svými kartézskými souřadnicemi. Při hledání minimální triangulace budeme potřebovat znát velikosti různých diagonál N-úhelníku. Délku úsečky spojující vrcholy $A(k)$ a $A(l)$ označme $DIST(k,l)$. Pomocná funkce $DIST$ spočte vzdálenost vrcholů $A(k)$ a $A(l)$ snadno pomocí Pythagorovy věty. Výsledek je roven druhé odmocnině ze součtu kvadrátů rozdílů x-ových a y-ových souřadnic bodů $A(k)$ a $A(l)$.

Hlavní datovou strukturou našeho algoritmu bude dvourozměrná tabulka reálných čísel $T[1\,..\,N, 1\,..\,N-2]$. Tabulku T budeme postupně zaplňovat tak, aby hodnotou $T[i,d]$ byla minimální možná velikost triangulace mnohoúhelníku $A(i)A(R(i+1))\ldots A(R(i+d))$, zvětšená ještě o délku úsečky $A(i)A(R(i+d))$. Budeme tedy uvažovat mnohoúhelníky tvořené $d+1$ sousedními vrcholy původního zadaného N-úhelníku. Pro každé pevně zvolené d z rozmezí od 2 do $N-2$ je takových mnohoúhelníků přesně N různých, neboť

je N možných voleb indexu i. Pro $d = N - 1$ je již takový mnohoúhelník jediný a je jím celý původní N-úhelník.

Pro $d = 1$ definujeme $T[i, d] = 0$ pro všechna i od 1 do N. Výpočet dalších hodnot $T[i, d]$ bude probíhat postupně po krocích pro d rostoucí od 2 do $N - 2$ a pro každé takové d vždy pro všechna i od 1 do N. Každou jednotlivou hodnotu $T[i, d]$ udávající minimální triangulaci mnohoúhelníku $A(i)A(R(i+1))\ldots A(R(i+d))$ zvětšenou o délku úsečky $A(i)A(R(i + d))$ spočteme následujícím způsobem: Každá triangulace uvedeného mnohoúhelníku obsahuje dvojici úseček $A(i)A(R(i+j))$, $A(R(i+j))A(R(i+d))$ pro nějaké číslo j z rozmezí 1 až $d - 1$. Triangulace totiž musí dělit plochu mnohoúhelníku na samé trojúhelníky, jeden z takto vzniklých trojúhelníků musí mít jako jednu svoji stranu úsečku $A(i)A(R(i+d))$ a jeho třetím vrcholem pak musí být právě nějaký bod $A(R(i + j))$. Při výpočtu hodnoty $T[i, d]$ proto budeme postupně zkoumat triangulace obsahující dvojici úseček $A(i)A(R(i + j))$, $A(R(i + j))A(R(i + d))$ pro všechna j od 1 do $d - 1$ (což jsou všechny existující triangulace, jak jsme právě vysvětlili). Velikost každé takové triangulace snadno spočteme jako součet hodnot $T[i, j]$ a $T[R(i+j), d-$ $- j]$, které již známe z předchozích kroků výpočtu, neboť tabulku T zaplňujeme postupně od nejmenších hodnot proměnné d k největším a platí jak $j < d$, tak i $d - j < d$. Za hodnotu $T[i, d]$ nyní stačí vzít minimální hodnotu ze všech velikostí takových triangulací pro j od 1 do $d - 1$ a přičíst k ní ještě délku úsečky $A(i)A(R(i + d))$.

Po zaplnění celé tabulky T lze nalézt výsledek úlohy mezi hodnotami $T[i, N - 2]$. Pro každou hodnotu indexu i od 1 do N udává $T[i, N - 2]$ minimální velikost triangulace

mnohoúhelníku $A(i)\ldots A(R(i+N-2))$ zvětšenou o délku úsečky $A(i)A(R(i+N-2))$. Čísla $T[i, N-2]$ jsou tedy všechna velikostmi jistých triangulací zadaného N-úhelníku a nejmenší z nich je hledanou minimální možnou velikostí triangulace.

Z uvedeného rozboru vyplývá správnost navrženého algoritmu. Výpočet podle algoritmu je jistě konečný, neboť počet všech opakování v cyklech je předem omezen hodnotou proměnné N. Základem algoritmu je výpočet hodnot uložených v tabulce T. Počet těchto počítaných hodnot $T[i, d]$ je úměrný N^2 (= velikost tabulky T). Přitom výpočet jednoho čísla $T[i, d]$ představuje řádově N operací (pro jednotlivé možné volby indexu j). Celý algoritmus má tedy časovou složitost N^3.

```
program TRIANGULACE (input, output);

const MAX = 100; {maximální počet vrcholů}

var AX, AY: array[1..MAX] of real;
                          {souřadnice vrcholů}
    N: integer; {počet vrcholů}
    T: array[1..MAX, 1..MAX] of real;
                          {tabulka dle rozboru}
    I, J, D: integer; {pomocné proměnné}
    M, Q: real; {pro výpočet minim}

function R (K: integer): integer;
  {přepočet indexů vrcholů modulo N}
begin
R := (K - 1) mod N + 1
end; {R}
```

```
function DIST (K, L: integer): real;
  {vzdálenost bodů v rovině}
begin
DIST := sqrt(sqr(AX[K] - AX[L]) +
                          + sqr(AY[K] - AY[L]))
end; {DIST}

begin
{Načtení vstupních dat:}
read(N);
for I := 1 to N do
  begin
  read(AX[I], AY[I]);
  T[I, 1] := 0
  end;

{Zaplnění tabulky T:}
for D := 2 to N - 2 do
  for I := 1 to N do
    begin
    M := 0; {libovolná inicializace proměnné M}
    for J := 1 to D - 1 do
      begin
      Q := T[I, J] + T[R(I + J), D - J];
      if (Q < M) or (J = 1) then M := Q
          {výběr minimální triangulace}
      end;
    T[I, D] := M + DIST(I, R(I + D))
    end;

{Stanovení výsledné hodnoty jako minimum z T[I, N - 2]:}
M := T[1, N - 2];
```

```
for I := 2 to N do
  if T[I, N - 2] < M then M := T[I, N - 2];
writeln('Minimální velikost triangulace');
writeln('zadaného mnohoúhelníku: ', M)
end.
```

P – III – 4

Budeme řešit pouze úlohu b), neboť tato úloha je obecnější a její řešení v sobě zahrnuje i řešení úlohy a). Samostatné řešení úlohy a) s použitím aritmetických operací zakázaných v b) se příliš neliší, základní myšlenka řešení i efektivita výsledného programu zůstává zachována, pouze zápis algoritmu se mírně zjednoduší.

V programu je nutné porovnávat mezi sebou různé dvojice hodnot. Porovnání jsme dosud prováděli pomocí příkazu SUB. Stačí si ovšem uvědomit, že odečíst od čísla jistou konstantu znamená totéž jako přičíst k němu stejnou konstantu, ovšem s opačným znaménkem. Tímto způsobem nahradíme operace SUB příkazy ADD, jejichž použití není zakázáno.

Úlohu je možné řešit mnoha různými způsoby. Ukážeme si zde dvě řešení, z nichž každé jiným způsobem využívá zásobníku k uchování potřebných údajů. V prvním řešení se do zásobníku ukládají pouze čítače obsahující vhodné informace o počtech čísel na vstupu. Hodnoty těchto čítačů se v průběhu výpočtu mění pomocí aritmetických operací. Druhé řešení úlohy používá zásobník přímo k uložení čísel ze vstupu. Počty uložených čísel tak vlastně nahrazují hodnoty čítačů z předchozího řešení.

První varianta řešení úlohy spočívá v provedení následujících kroků výpočtu:

1. Ze vstupu se čtou čísla „10". V zásobníku se přitom udržuje čítač, jehož hodnota se průběžně zvyšuje o 1, takže po přečtení všech „10" ze vstupu čítač udává jejich počet.

2. Ze vstupu se čtou čísla „20". V zásobníku se přitom udržují dva čítače, jejichž hodnota se průběžně mění. Na dně zásobníku se od dříve stanoveného počtu „10" vždy odečte 1 a na vrcholu zásobníku se počítá počet všech „20" přičítáním 1 k druhému čítači. Po přečtení všech čísel „20" ze vstupu jsou v zásobníku uloženy dvě hodnoty: na dně zásobníku je rozdíl počet „10" — počet „20" a na vrcholu počet „20".

3. Je-li hodnota udávající rozdíl počtu „10" a „20" nulová, vytiskne se „1" jako výsledek práce algoritmu a ukončí se výpočet. Jinak se tato hodnota ponechá na dně zásobníku pro další použití, nad ní na vrcholu zásobníku zůstává uložen počet „20" a pokračuje se ve výpočtu následujícím krokem.

4. Ze vstupu se čtou čísla „30". Přitom se průběžně snižuje o 1 hodnota čítače uloženého na vrcholu zásobníku, takže po přečtení všech čísel ze vstupu bude tento čítač udávat rozdíl počet „20" — počet „30" .

5. Je-li hodnota udávající rozdíl počtu „20" a „30" nulová, vytiskne se „1" jako výsledek práce algoritmu a ukončí se výpočet. Jinak se tato hodnota ponechá na vrcholu zásobníku pro další použití, pod ní na dně zásobníku zůstává nadále uložen rozdíl počtu „10" a „20" a pokračuje se ve výpočtu následujícím krokem.

6. Sečtou se obě hodnoty uložené v zásobníku. Tyto hod-

noty představují dříve zjištěné rozdíly (počet „10“ — počet „20“) a (počet „20“ — počet „30“). Výsledkem tohoto součtu je hodnota rozdílu (počet „10“ — počet „30“).

7. Je-li výsledný rozdíl nulový, algoritmus vytiskne výsledek „1“, jinak vytiskne „0“. Tím je úloha vyřešena a algoritmus ukončí svoji práci.

```
CONST −1                (počet „10“ je zatím nulový ..)
PUSH                    (... z techn. důvodů dáme −1)
IN
PUSH                    (na vrcholu je přečtené číslo)
CONST −10
ADD
while ZERO do
    begin               (přečtené číslo je „10“)
    POP                 (zrušení „10“ ze zásobníku)
    CONST 1
    ADD                 (zvýšíme hodnotu čítače o 1)
    EXCH                (uložíme novou hodnotu čítače)
    IN
    PUSH                (další číslo ze vstupu do zás.)
    CONST −10
    ADD                 (otestujeme, zda je to „10“)
    end
CONST 1                 (máme přečtenu první „20“)
EXCH                    (založení druhého čítače)
```

(za toto první číslo „20“ nemusíme snižovat hodnotu čítače na dně zásobníku díky počáteční inicializaci na −1 místo na 0 v prvním řádku programu; na

dně zásobníku je nyní počet „10“ zmenšený o 1 a na vrcholu hodnota 1 — zatím bylo přečteno jedno číslo „20“)

```
IN
PUSH                    (přečtené číslo na vrcholu)
CONST -20
ADD                     (otestujeme, zda je to „20“)
while ZERO do
    begin               (přečtené číslo je „20“)
    POP                 (zrušení „20“ ze zásobníku)
    CONST 1
    ADD                 (nová hodnota horního čítače)
    POP
    EXCH                (záměna uložení čítačů)
    PUSH
    CONST -1
    ADD                 (nová hodnota druhého čítače)
    POP
    EXCH                (obnova uložení čítačů)
    PUSH
    IN
    PUSH                (další číslo ze vstupu do zás.)
    CONST -20
    ADD                 (otestujeme, zda je to „20“)
    end
```

(nyní je v zásobníku uložen rozdíl počtu „10“ a „20“, nad ním je počet „20“, na vrcholu zásobníku první číslo „30“ přečtené ze vstupu)

```
POP
TOP                    (horní čítač do registru)
POP
EXCH                   (záměna čítačů)
if ZERO then
    CONST 1            (počet „10“ = počet „20“)
else
    begin
    EXCH               (obnova uložení čítačů)
    PUSH
    CONST -1
    ADD                (již jsme přečetli první „30“)
    EXCH               (nová hodnota horního čítače)
    while not EOF do
        begin
        IN             (přečtení čísla „30“ ze vstupu)
        CONST -1
        ADD            (snížení hodnoty čítače)
        EXCH
        end
```

(nyní jsou přečtena všechna čísla ze vstupu; v zásobníku jsou uloženy rozdíly počtu „10“ a „20“ a počtu „20“ a „30“)

```
    TOP                (horní čítač do registru)
    if ZERO then
        CONST 1        (počet „20“ = počet „30“)
    else
        begin
        POP
```

```
        ADD              (součet obou čítačů)
        if ZERO then
          CONST 1        (počet „10“ = počet „30“)
        else
          CONST 0        (záporný výsledek)
        end
    end
OUT                      (tisk výsledku)
```

Druhá varianta řešení zadané úlohy se skládá z postupného provedení následujících kroků výpočtu:

1. Ze vstupu se přečtou všechna čísla „10“. Průběžně se ukládají do zásobníku a v čítači udržovaném na vrcholu zásobníku se zároveň spočítá jejich počet.

2. Ze vstupu se přečtou všechna čísla „20“. Ukládají se do zásobníku nad čísla „10“, která byla do zásobníku vložena v předcházejícím kroku. Odčítáním jedničky za každé uložené číslo „20“ od čítače udržovaného na vrcholu zásobníku se určí rozdíl počtu čísel „10“ a „20“.

3. Je-li tento rozdíl nulový, vytiskne se „1“ jako výsledek práce algoritmu a ukončí se výpočet. Jinak se čítač z vrcholu zásobníku zruší a pokračuje se ve výpočtu následujícím krokem.

4. Ze vstupu se přečtou všechna čísla „30“. Při jejich čtení se průběžně vypouštějí čísla „20“ ze zásobníku, dokud je to možné (za každé přečtené číslo „30“ se vypustí jedno číslo „20“ ze zásobníku). Tím se určí rozdíl počtu „30“ a „20“. Zároveň se v čítači udržovaném na vrcholu zásobníku spočítá počet všech přečtených čísel „30“.

5. Je-li na vstupu stejný počet čísel „30“ a „20“, je možné vypustit ze zásobníku za každé čtené číslo „30“ jedno číslo „20“ (tj. nestane se, že by se během čtení „30“ ze vstupu narazilo v zásobníku na číslo „10“ uložené pod všemi „20“) a po přečtení všech čísel ze vstupu už naopak v zásobníku žádné číslo „20“ nezbude. V takovém případě vytiskne algoritmus výsledek „1“ a skončí. Jinak se ze zásobníku vypustí všechna zbylá čísla „20“ (jsou-li tam nějaká) a pokračuje se ve výpočtu následujícím krokem.

6. Porovná se počet čísel „30“ zaznamenaný v čítači na vrcholu zásobníku s počtem čísel „10“ uložených v zásobníku pod tímto čítačem. Toho dosáhneme postupným vypouštěním čísel „10“ a současným snižováním hodnoty čítače.

7. Jsou-li tyto počty stejné, algoritmus vytiskne výsledek „1“, jinak vytiskne „0“. Tím je úloha vyřešena a algoritmus ukončí svoji práci.

Naprogramování uvedeného postupu řešení je opět pouze technickou záležitostí. Kroky 1 až 3 jsou snadné. V kroku 4 je výhodné zavést zvláštní pomocnou hodnotu ukládanou do zásobníku jako příznak v případě, že při vybírání čísel „20“ ze zásobníku zjistíme, že jich je méně, než kolik je „30“ na vstupu. Tento příznak nám umožní rozhodnout o dalším postupu v kroku 5. Můžeme použít například číslo „40“, kterým nahradíme číslo „10“ ležící v zásobníku nejblíže k vrcholu hned pod čísly „20“. Zbývající kroky výpočtu jsou již opět snadné.