38. ročník matematické olympiády na středních školách

Kategorie P

In: Leo Boček (editor); Jiří Binder (editor); Tomáš Hecht (editor); Karel Horák (editor); Pavel Töpfer (editor): 38. ročník matematické olympiády na středních školách. Zpráva o řešení úloh ze soutěže konané ve školním roce 1988/89. 30. mezinárodní matematická olympiáda. (Czech). Praha: Státní pedagogické nakladatelství, 1991. pp. 113–177.

Kategorie P

Texty úloh

P - I - 1

Ve státě K je N měst $A(1), \ldots, A(N)$. Velké povodně zničily řadu mostů. Z tohoto důvodu bylo mezi mnoha městy přerušeno spojení. Je dáno pole $B[1..N, 1..N]$, jehož prvky mají hodnotu 0 nebo 1. Prvek $B[i,j]$ bude mít hodnotu *1*, jestliže vede přímá cesta z města $A(i)$ do města $A(j)$, a hodnotu 0, pokud se z města $A(i)$ do města $A(j)$ přímo nedá dostat. Uvědomte si, že $B[i,j] = B[j,i]$. Nalezněte a zdůvodněte algoritmus, který pro daná i, j zjistí, zda je možné dostat se z města $A(i)$ do města $A(j)$.

P - I - 2

Na kružnici leží N různých bodů $A(1), A(2), \ldots, A(N)$ $(N \geqq 4)$ v tomto pořadí ve směru hodinových ručiček. Pro každé dva body X a Y na kružnici označíme $/XY/$ délku oblouku z X do Y ve směru hodinových ručiček.

Je dáno pole $D[1], D[2], \ldots, D[N]$ přirozených čísel, ve kterém

$D[I] = /A(I)A(I+1)/$ pro $1 \leqq I < N$,
$D[N] = /A(N)A(1)/$.

Nalezněte algoritmus, který zjistí, zda existují indexy p, q, r, s takové, že

$$p < q < r < s$$

a současně

$/A(p)A(q)/ = /A(q)A(r)/ = /A(r)A(s)/ = /A(s)A(p)/$.

Při výpočtech, které budete v algoritmu provádět, musí být všechny výsledky i mezivýsledky celá čísla.

P-I-3

Je dán následující program

V PASCALU	V BASICU
var *I*, *J*: integer;	
A: **array** [1..100]	10 DIM A(100)
of integer;	
begin	
for *I*:=1 **to** 100 **do**	20 FOR I=1 TO 100
read (*A*[*I*]);	30 INPUT A(I)
	40 NEXT I
I:=1;	50 LET I=1
J:=1;	60 LET J=1

```
while I+J ≦ 100 do
  if A[I] = A[I+J]
    then J:=J+1
    else I:=I+1;
write(J)
end
```

```
70 IF I+J>100 THEN GOTO 100
80 IF A(I)=A(I+J) THEN
       LET J=J+1 : GOTO 70
90 LET I=I+1 : GOTO 70
100 PRINT J
```

Vstupem programu je uspořádané pole (ne nutně různých) přirozených čísel. Určete a zdůvodněte, co je výsledkem práce programu (tj. jak souvisí výsledná hodnota J s hodnotami pole A a proč).

P - I - 4

Nejprve zavedeme některé pojmy, které budeme potřebovat ve čtvrté úloze. Úlohy podobného charakteru budou i v krajském a celostátním kole.

Mnohoúhelník je část roviny ohraničená jednou lomenou čárou, která sama sebe v žádném bodě neprotíná. Body této lomené čáry patří také do mnohoúhelníku. Mnohoúhelník je zadán posloupností svých vrcholů uspořádaných proti směru hodinových ručiček.

Mnohoúhelník je *konvexní,* jestliže s každými dvěma svými body obsahuje také úsečku, která je spojuje.

Konvexním obalem N bodů v rovině budeme rozumět nejmenší konvexní mnohoúhelník, který je všechny obsahuje. (Nejmenší ve smyslu množinové inkluze.)

Při řešení úloh budete moci kromě příkazů programovacího jazyka (Pascalu nebo Basicu) využívat také tři funkce: VPRAVO, VNITR a UHEL:

VPRAVO (*X1*, *Y1*, *X2*, *Y2*, *X3*, *Y3*) — výsledkem bude logická hodnota »pravda«, jestliže body *P1* = (*X1*, *Y1*) a *P2* = (*X2*, *Y2*) jsou různé a bod *P3* = (*X3*, *Y3*) leží na přímce procházející body *P1* a *P2* nebo vpravo od ní při pohledu z bodu *P1* směrem k bodu *P2*. Jinak bude výsledkem »nepravda«.

VNITR (*X*, *Y*, *X1*, *Y1*, *X2*, *Y2*, *X3*, *Y3*) — výsledkem bude logická hodnota »pravda«, jestliže body *P1* = (*X1*, *Y1*), *P2* = (*X2*, *Y2*) a *P3* = (*X3*, *Y3*) neleží na jedné přímce a bod *P* = (*X*, *Y*) je bodem trojúhelníku *P1 P2 P3*. Jinak je výsledkem »nepravda«.

UHEL (*X1*, *Y1*, *X2*, *Y2*, *X3*, *Y3*) — jestliže *P1* = (*X1*, *Y1*), *P2* = (*X2*, *Y2*) a *P3* = (*X3*, *Y3*) jsou tři různé body, potom výsledkem bude velikost úhlu *P1 P2 P3* ve stupních od 0 stupňů do 360 stupňů měřená proti směru hodinových ručiček.

Soutěžní úloha

a) Nalezněte (co nejlepší) algoritmus, který zjistí, zda daný mnohoúhelník je konvexní, a zdůvodněte jeho správnost.

b) Nalezněte (co nejlepší) algoritmus, který pro *N* zadaných bodů (*N* > = 4) v rovině najde jejich konvexní obal, a zdůvodněte jeho správnost. Souřadnice konvexního obalu uložte do pole *KONOBALX* a *KONOBALY* a počet bodů obalu do proměnné *POCET*.

P - II - 1

Nalezněte a dokažte (co nejlepší) algoritmus, který pro nezáporné celé číslo *n* vypočítá celočíselnou hodnotu *f*(*n*), pro kterou platí:

$f(0) = 1, f(1) = 1, f(2n) = 2f(n), f(2n + 1) = 2f(n + 1) - f(n)$
Nepoužívejte rekurzi a minimalizujte spotřebu paměti!

P - II - 2

Je dáno dvojrozměrné pole A (matice) rozměru $N \times M$, jehož prvky obsahují pouze čísla 0 nebo 1. Nalezněte a dokažte co nejlepší algoritmus, který v daném poli A nalezne maximální »obdélník« obsahující samé jedničky (maximální ve smyslu »obsahující nejvíc jedniček«). Výsledkem práce algoritmu bude čtveřice čísel I, J, K, L takových, že $A[I, J]$ je prvek v levém horním rohu a $A[K, L]$ prvek v pravém dolním rohu nalezeného maximálního obdélníku.

P - II - 3

Je dán algoritmus

P: **for** $i := 1$ **to** $N - 1$ **do**
  if $A[i] > A[i + 1]$ **then**
    $A[i] := : A[i + 1]$; (výměna dvou sousedních prvků pole)

Napište a dokažte (co nejlepší) program, který pro dané celočíselné N-prvkové pole A zjistí, zda by výše uvedený algoritmus P toto pole vzestupně uspořádal. Výsledný program nesmí modifikovat pole A ani používat jiné pomocné pole.

P - II - 4

Nalezněte a dokažte (co nejlepší) algoritmus, který zjistí, zda se daný bod nachází uvnitř daného mnohoúhelníku.

Poznámka. Definice základních pojmů a pomocných funkcí, které je možné při řešení použít, jsou uvedeny u zadání úlohy P — I — 4.

P - III - 1

Nalezněte a dokažte (co nejlepší) algoritmus, který pro dva soubory konečných po dvou disjunktních množin přirozených čísel $P = \{P1, P2, \ldots, Pm\}$ a $Q = \{Q1, Q2, \ldots, \ldots, Qn\}$ takové, že $P1 \cup P2 \cup \ldots \cup Pm = Q1 \cup Q2 \cup \cup \ldots \cup Qn$, najde nejmenší počet K operací sjednocení a rozdělení množin, jimiž se P převede na Q. Operací sjednocení se nahradí dvě množiny v souboru jejich sjednocením. Operací rozdělení se jedna množina souboru rozdělí na dvě disjunktní části.

P - III - 2

Řekneme, že matice $M \times N$ s prvky 0 a 1 obsahuje »čárový vzorek«, jestliže každý jedničkový prvek (případně s výjimkou těch, které leží na okraji matice) má právě dva jedničkové sousední prvky (soused může být vlevo, vpravo, nahoru nebo dolů, ne ve směru úhlopříčky).

Nalezněte a dokažte (co nejlepší) algoritmus, který pro danou matici $M \times N$ obsahující čárový vzorek a její daný nulový prvek $A[I, J]$ »vybarví« (tj. přepíše nuly například na dvojky) plochu ohraničenou »čarou« z jedničkových prvků a okraji matice a obsahující prvek $A[I, J]$.

P - III - 3

Je dán algoritmus P:

for $\check{y} := 1$ **to** K **do**
  for $I := 1$ **to** $N - 1$ **do**
    if $A[I] > A[I + 1]$ **then**
      $A[I] := : A[I + 1]$; (výměna dvou sousedních prvků)

Napište a dokažte (co nejlepší) program, který pro dané celočíselné N-prvkové pole A najde nejmenší číslo K takové, aby výše uvedený algoritmus P vzestupně uspořádal toto pole. Výsledný program nesmí modifikovat pole A ani používat jiné pomocné pole.

P - III - 4

a) Nalezněte a dokažte (co nejlepší) algoritmus, který rozdělí daný (ne nutně konvexní) mnohoúhelník na neprotínající se trojúhelníky s vrcholy ve vrcholech mnohoúhelníku. Výstupem algoritmu je množina dvojic vrcholů, jejichž pospojováním dostaneme takové rozdělení. (Stačí najít jedno řešení.)

b) Nalezněte a dokažte (co nejlepší) algoritmus, který zjistí, zda lomená čára zadaná posloupností bodů $P(1), \ldots,$ $\ldots, P(N), P(1)$ tvoří mnohoúhelník.

Poznámka. Definice základních pojmů a pomocných funkcí, které je možné při řešení použít, jsou uvedeny u zadání úlohy P — I — 4.

P-I-1

Úlohu lze řešit více různými způsoby. Ukážeme jeden z těch možných algoritmů, které jsou časově nejefektivnější. Algoritmus postupně vytváří seznam čísel měst, do nichž se lze dostat z výchozího města $A(i)$. Na začátku práce algoritmu je v seznamu uloženo pouze číslo i. V každém kroku algoritmu vyjmeme ze seznamu jedno (libovolné) číslo K a místo něj do seznamu zařadíme čísla těch měst, která jsou přímo spojena s městem $A(K)$. Každé číslo města přitom bude do seznamu vloženo celkem nejvýše jednou. Vytváření seznamu ukončíme v okamžiku, kdy do něj zařadíme číslo j označující cílové město $A(j)$, nebo když již není možné zařadit do seznamu žádné další číslo města. Výsledek práce algoritmu je určen tím, zda bylo číslo cílového města zařazeno do seznamu.

Programová realizace uvedeného algoritmu vyžaduje především zvolit vhodnou datovou strukturu pro ukládání vytvářeného seznamu. Vzhledem k tomu, že počet měst N je předem znám a že do seznamu bude číslo každého města zařazeno nejvýše jednou, zvolíme pro reprezentaci seznamu jednorozměrné pole $S[1..N]$ a jednu pomocnou proměnnou P, která udává momentální délku seznamu ($S[P]$ je vždy poslední prvek seznamu). Na začátku práce algoritmu je tedy $P = 1$ a $S[1] = i$.

Nyní je třeba vyřešit způsob volby čísla K. Na jeho výběru výsledek práce algoritmu nezávisí, v principu je možné

zvolit libovolné z čísel uložených momentálně v seznamu. Z hlediska efektivní programové realizace jsou vhodné dva způsoby:

1. Za K zvolíme číslo naposledy zařazené do seznamu, tzn. číslo $S[P]$. Odstranění tohoto čísla ze seznamu se provede snadno snížením hodnoty proměnné P o jedničku. Se seznamem v tomto případě pracujeme jako se zásobníkem, což odpovídá prohledávání stromu všech možných cest z počátečního města $A(i)$ do cílového města $A(j)$ tzv. »do hloubky«.

2. Za K zvolíme číslo ze začátku seznamu. K tomu je vhodné zavést pomocnou proměnnou Z, která bude v každém okamžiku udávat momentální začátek platného vytvářeného seznamu v poli S, takže $S[Z]$ je vždy první prvek seznamu. Na začátku práce algoritmu je $Z = 1$. Za K potom vezmeme číslo $S[Z]$ a odstranění tohoto čísla ze seznamu dosáhneme zvýšením hodnoty proměnné Z o jedničku. Vybrané číslo tedy z pole S ve skutečnosti neodstraníme, pouze vyznačíme, že již nepatří do seznamu. Se seznamem v tomto případě pracujeme jako s frontou, což odpovídá prohledávání stromu všech možných cest z počátečního města $A(i)$ do cílového měst $A(j)$ tzv. »do šířky«.

Obě uvedené strategie jsou stejně dobré a řeší zadanou úlohu. V našem programu zvolíme například druhou z nich. Uvedeme ještě dvě poznámky k zavedení pomocné proměnné Z. Tato proměnná vyznačující začátek seznamu v poli S není nutná, bylo by možné vybrané číslo vždy skutečně odstranit z pole S a všechna ostatní čísla náležející do seznamu posunout v poli S o jedno místo. Takové řešení by ovšem bylo značně pracnější a pomalejší. Postup využívající pomocnou

proměnnou Z navíc udržuje v poli S čísla všech měst, o nichž již víme, že jsou dosažitelná z města $A(i)$.

Zbývá provést poslední akci — pro zvolené město s číslem K zařadit do seznamu čísla těch měst, která mají přímé spojení s městem $A(K)$ a která do seznamu dosud nebyla zařazena. Čísla měst s přímým spojením s městem $A(K)$ získáme snadno přímo ze zadané matice B. Stačí projít K-tý řádek této matice a vyhledat indexy těch sloupců, v nichž je na K-tém řádku uložena hodnota 1. Z nalezených čísel měst máme do seznamu zařadit pouze ta, která do něj dosud zařazena nebyla. Pro tento účel je výhodné zavést pomocné pole $R[1..N]$, ve kterém se bude o každém z měst evidovat informace, zda již bylo jeho číslo zařazeno do seznamu. Údaj $R[L] = 1$ znamená, že číslo L již bylo do vytvářeného seznamu zařazeno, $R[L] = 0$ znamená opak. Na začátku práce algoritmu budou všechny hodnoty pole R nastaveny na 0, pouze $R[i] = 1$. Můžeme opět poznamenat, že ve variantě řešení, kterou jsme zvolili pro naši realizaci (prohledávání »do šířky«, se seznamem se pracuje jako s frontou, užívá se pomocné proměnné Z k vyznačení začátku seznamu), není zavedení pole R nezbytné, neboť v poli S se udržují čísla všech měst zařazených někdy během výpočtu do seznamu. Čísla měst, která již byla ze seznamu vybrána, zůstala uložena na místech $S[1], \ldots, S[Z - 1]$. Prohledávání celého pole S před zařazením každého nového čísla do seznamu by ovšem bylo značně pracnější než pouhé testování příznaku v poli R.

Následující program přesně realizuje popsaný algoritmus. Pro jednoduchost je v něm počet měst N zadán přímo jako

konstanta (není obtížné změnit). Pole B vstupních hodnot je programem čteno po řádcích.

```
program MESTA (input, output);

const N = 10;                    (počet měst)

var B: array [1..N, 1..N] of integer; (zadaná matice cest)
    S, R: array [1..N] of integer;
        {S — vytvářený seznam měst, R — příznaky}
  Z, P: integer;                 {indexy seznamu v poli S}
  I, J: integer;                 {začátek a cíl cesty}
  K, U, V: integer;              {pomocné proměnné}

begin

{Načtení vstupních hodnot:}
for U := 1 to N do
  for V := 1 to N do
      read (B[U, V]);
read (I, J);
{Inicializace proměnných:}
P := 1;
Z := 1;
S[1] := I;
for U := 1 to N do
  R[U] := 0;
R[I] := 1;
```

```
{Vlastní výpočet:}
repeat
K : = S[Z];                    {vybráno číslo K ze seznamu}
Z : = Z + 1;
for U : = 1 to N do
  if (B[K, U] = 1) and (R[U] = 0) then
      begin                    {zařazení čísla U do seznamu}
      P : = P + 1;
      S[P] : = U;
      R[U] : = 1
      end
until (Z > P) or (R[ℱ] = 1);

if R[ℱ] = 1 then
  writeln ('Cesta je možná.')
else
  writeln ('Cesta není možná.')

end.
```

Výpočet podle uvedeného algoritmu je konečný, neboť do vytvářeného seznamu je každé z N čísel měst zařazeno nejvýše jednou a při každém kroku algoritmu je ze seznamu vyřazeno právě jedno číslo. Výpočet skončí nejpozději po vyprázdnění celého seznamu, tzn. nejvýše po N krocích.

Popsaný algoritmus má kvadratickou časovou složitost. Již jsme ukázali, že výpočet vyžaduje provedení nejvýše N kroků, tj. N průchodů cyklu repeat-until v programu. V každém kroku se přitom provádí jeden for-cyklus o N průchodech. Celkově tedy výpočet vyžaduje provedení řádově $N * N$ operací. Z hlediska časové složitosti lepší algoritmus

než kvadratický není možný, neboť již jenom vstupní data mají velikost úměrnou hodnotě $N * N$ (velikost zadané matice B) a pro výsledek úlohy jsou všechny vstupní údaje významné.

Správnost uvedeného řešení vyplývá přímo z popisu algoritmu. Do vytvářeného seznamu jsou postupně zařazována čísla měst, která jsou dosažitelná z výchozího města i. Pokud je tedy do seznamu zařazeno také číslo cílového města j a výpočet skončí s hodnotou $R[j] = 1$, je správně ohlášeno, že cesta z města $A(i)$ do města $A(j)$ je možná. Jestliže výpočet skončí v situaci, že se celý seznam vyprázdnil, byla již do seznamu postupně zařazena (a opět z něj odstraněna) čísla všech měst, která mají spojení s městem $A(i)$. Je-li tedy $R[j] = 0$, je správně ohlášeno, že cesta z města $A(i)$ do města $A(j)$ není možná.

P-1-2

Úlohu je možné řešit více různými způsoby. Ukážeme zde algoritmus, který je z hlediska časové efektivity optimální, neboť je lineární.

Součet zadaných vzdáleností $D[1], \ldots, D[N]$ sousedních bodů na kružnici určuje délku kružnice. Mají-li existovat body dělící kružnici na čtyři stejně velké úseky celočíselné velikosti, musí být délka kružnice násobkem čtyř. Nejprve proto zjistíme, zda platí tato základní nutná podmínka. Jestliže ano, spočítáme délku čtvrtkružnice a označíme ji *CTVRT*. V opačném případě indexy požadovaných vlastností neexistují a úloha nemá řešení.

Zavedeme pomocné pole $E[0..N]$, které bude obsahovat $N + 1$ hodnot splňujících následující podmínky:

1. $E[0] = 0$
2. pro $i = 1, \ldots, N$:
   $E[i] = j$ jestliže existuje index j takový, že $j > i$ a zároveň $/A(i)\ A(j)/ = CTVRT$
   $E[i] = 0$ jinak

S využitím tohoto pole E je již řešení zadané úlohy snadné. Stačí ověřit, zda pro nějakou hodnotu indexu $p = 1, \ldots, N$ platí $E[E[E[p]]] <> 0$. Platnost této nerovnosti pro nějakou hodnotu p je nutnou i postačující podmínkou existence požadovaného dělení kružnice na čtyři stejné části. Hledanými indexy p, q, r, s jsou potom po řadě hodnoty p, $E[p]$, $E[E[p]]$, $E[E[E[p]]]$.

Platnost uvedeného tvrzení ihned dokážeme:

a) Jestliže existují indexy p, q, r, s takové, že platí $p < q < r < s$ a zároveň $/A(p)\ A(q)/ = /A(q)\ A(r)/ = /A(r)\ A(s)/ = /A(s)\ A(p)/$, pak podle definice pole E bude $E[p] = q$, $E[q] = r$, $E[r] = s$, a tedy $E[E[E[p]]] = s$. Pro hodnotu indexu p tudíž platí nerovnost $E[E[E[p]]] <> 0$.

b) Nechť naopak pro nějakou hodnotu indexu p platí nerovnost $E[E[E[p]]] <> 0$. Označme $q = E[p]$, $r = E[E[p]]$, $s = E[E[E[p]]]$. Z předpokladu $s <> 0$ vyplývá i $q <> 0$, $r <> 0$, neboť $E[0] = 0$. Přímo z definice pole E nyní dostáváme:

$p < q$ $/A(p)\ A(q)/ = CTVRT$
$q < r$ $/A(q)\ A(r)/ = CTVRT$
$r < s$ $/A(r)\ A(s)/ = CTVRT$

Jistě je také $/A(s)\ A(p)/ = CTVRT$, neboť

$/A(p)\ A(q)/ + /A(q)\ A(r)/ + /A(r)\ A(s)/ + /A(s)\ A(p)/ = 4 * CTVRT.$

Indexy p, q, r, s mají tudíž všechny požadované vlastnosti ze zadání úlohy.

Dokázali jsme, že algoritmus řešící zadanou úlohu pomocí hodnot pole E je správný. Výpočet podle tohoto algoritmu je jistě konečný, nevyžaduje provést více než N porovnání. Algoritmus je zřejmě lineární, jeho časové nároky jsou úměrné hodnotě N.

Při ověřování nerovnosti $E[E[E[p]]] <> 0$ v algoritmu navíc není nutné procházet indexem p všechny hodnoty od 1 do N. Stačí prověřovat hodnoty p od 1 do takového čísla M, pro které $/A(1)\ A(M)/ < CTVRT$ a $/A(1)\ A(M+1)/ >= CTVRT$. Pro $p > M$ by totiž nemohla být splněna podmínka $p < q < r < s$ ze zadání úlohy a vzhledem k definici pole E by jistě platilo $E[E[E[p]]] = 0$.

Zbývá navrhnout algoritmus na vytvoření pole E. Snadno bychom sestrojili kvadratický algoritmus, který postupně pro všechny hodnoty $i = 1, \ldots, N$ připočítáváním délek jednotlivých elementárních úseků zjišťuje, zda existuje odpovídající index j, pro nějž platí podmínka z definice pole E. Ukážeme zde lepší, lineární algoritmus.

Budeme používat tři pomocné proměnné charakterizující v každém okamžiku jistý sledovaný úsek kružnice (úsek je pro nás potenciální čtvrtkružnicí): proměnná *DELKA* udává délku úseku, *DOLNI* je index počátečního bodu a *HORNI* index koncového bodu tohoto úseku. Stále tedy platí: */A(DOLNI) A(HORNI)/ = DELKA*. Vždy bude *DOLNI* $<=$ *HORNI*. Jestliže najdeme dvojici hodnot *DOLNI*, *HORNI*, pro kterou je *DELKA* = *CTVRT*, uložíme do pole E hodnotu $E[DOLNI] = HORNI$. Pokud pro zvolené

DOLNI takové *HORNI* neexistuje, bude $E[DOLNI] = 0$. Toto přesně odpovídá definici pole *E*.

Výpočet začíná pro $DOLNI = 1, HORNI = 1, DELKA = 0$. Dokud je $DELKA < CTVRT$, postupně prodlužujeme sledovaný úsek zvyšováním hodnoty proměnné *HORNI* (a zároveň počítáme délku úseku v proměnné *DELKA*). Poté uložíme do pole *E* správnou hodnotu $E[DOLNI]$. Optimalizace výpočtu spočívá v tom, že pro další hodnotu indexu *DOLNI* nebudeme počítat délku úseku opět od nuly, ale využijeme předchozí hodnoty. Zvětšíme hodnotu proměnné *DOLNI* o 1 a upravíme hodnotu proměnné *DELKA* odečtením délky příslušného elementárního úseku, který jsme ze sledovaného úseku tímto vynechali. Výpočet se nyní bude opakovat. Celý výpočet skončí, jakmile hodnota proměnné *HORNI* překročí hodnotu *N*. Zbývající dosud nespočtené hodnoty pole *E* budou rovny 0, jak vyplývá přímo z definice pole *E*.

Správnost uvedeného postupu pro výpočet hodnot pole *E* není třeba zvlášť zdůvodňovat, jedná se o konstrukci hodnot pole *E* přímo podle definice. Výpočet podle uvedeného algoritmu je jistě konečný, neboť v každém jeho kroku dochází buď ke zvýšení hodnoty proměnné *HORNI* nebo proměnné *DOLNI* o 1. Přitom na začátku výpočtu je $HORNI = DOLNI = 1$, stále platí $DOLNI <= HORNI$ a výpočet končí ihned, jakmile hodnota proměnné *HORNI* překročí *N*. Vykoná se tedy méně než $2 * N$ kroků algoritmu. Odtud také plyne, že i algoritmus na sestrojení pole *E* je lineární.

```
program KRUZNICE (input, output);

const N = 10;                    {počet zadaných bodů}
```

```
var  D: array [1..N] of integer;      {vstupní data}
     E: array [0..N] of integer;      {pole E podle rozboru}
     CTVRT, M, DELKA, DOLNI, HORNI:   integer;
                                      {proměnné z rozboru úlohy}
     I, J: integer;                   {pomocné proměnné}


begin
J := 0;
for I := 1 to N do
  begin
  read (D[I]);                        {načtení vstupních dat}
  J := J + D[I]                       {výpočet délky kružnice}
  end;
I := 0;                  {pro ukončení v případě J mod 4 <> 0}
if J mod 4 = 0 then {základní nutná podmínka ex. řešení}
  begin
  CTVRT := J div 4;   {velikost čtvrtkružnice}
  {Výpočet hodnot pole E:}
  E[0] := 0;
  DOLNI := 1;
  HORNI := 1;
  DELKA := 0;
  while HORNI <= N do
    if DELKA < CTVRT then
      begin            {prodloužit sledovaný úsek}
      DELKA := DELKA + D[HORNI];
      HORNI := HORNI + 1
      end
    else
```

```
      begin                {definovat hodnotu pole E ...}
      if DELKA = CTVRT then
          E[DOLNI] : = HORNI
      else
        E[DOLNI] : = 0;
      DELKA : = DELKA - D[DOLNI];
                           {... a zkrátit sledovaný úsek}
      DOLNI : = DOLNI + 1
      end;
for I : = DOLNI to N do
  E[I] : = 0;
{Výpočet maximálního indexu M:}
M : = 0;
ƒ : = 0;
while ƒ < = CTVRT do
  begin
  M : = M + 1;
  ƒ : = ƒ + D[M]
  end;
{Vlastní výpočet řešení úlohy:}
I : = 1;
while (E[E[E[I]]] = 0) and (I < M) do
  I : = I + 1
end;
{Výsledek výpočtu:}
if E[E[E[I]]] = 0 then
  writeln ('Indexy požadovaných vlastností neexistují.')
else
  begin
  writeln ('Indexy požadovaných vlastností existují.');
```

```
    writeln ('Vyhovuje například čtveřice indexů:');
    writeln (I : 10, E[I] : 10, E[E[I]] : 10, E[E[E[I]]] : 10)
  end
end.
```

P - I - 3

Výsledkem práce zadaného programu je hodnota proměnné *J*. Ukážeme, že na konci programu proměnná *J* udává délku maximálního úseku stejných čísel ve vstupních datech.

Vzhledem k uspořádání pole A musí shodné prvky tvořit souvislý úsek. Označme délku nejdelšího takového úseku jako D a výstupní hodnotu proměnné *J* symbolem V. Dokážeme, že výpočet programu je vždy konečný a že $V = D$.

1. Program obsahuje jediný cyklus typu while, a to s podmínkou $I + J <= 100$. Na začátku výpočtu má výraz $I + J$ hodnotu 2 a při každém průchodu cyklem jeho hodnota vzroste o 1. Cyklus se tedy provede 99krát a skončí, jakmile součet $I + J$ dosáhne hodnoty 101. Výpočet je tudíž konečný.

2. $V <= D$

Jistě $V >= 1$, neboť proměnná *J* má na začátku výpočtu hodnotu 1 a nikdy se nezmenšuje. Jestliže $V = 1$, pak dokazovaná nerovnost $V <= D$ zřejmě platí, neboť $D >= 1$. Pokud $V > 1$, musela proměnná *J* nabýt hodnoty V zvětšením o 1 z hodnoty $V - 1$ při splnění podmínky $A[I] = A[I + V - 1]$ pro nějaké I. Protože je pole A setříděné, musí se sobě rovnat také všechny prvky ležící mezi $A[I]$ a $A[I + V - 1]$, tzn. platí $A[I] = A[I + 1] = \ldots = A[I + V - 1]$. Existuje tedy úsek stejných čísel v poli A

délky V a tudíž $V <= D$. Nerovnost $V <= D$ je tak dokázána.

3. $V = D$

Nechť úsek stejných čísel délky D v poli A začíná prvkem s indexem M, tj. platí $A[M] = A[M + 1] = \ldots = A[M + D - 1]$ pro $M + D - 1 <= 100$. Proměnná I musí během výpočtu nabýt hodnoty M. Po ukončení výpočtu je totiž $I + J = 101$, a kdyby bylo $I < M$, muselo by platit $J > 101 - M$ neboli také $V > 101 - M$. Z nerovností $M + D - 1 <= 100$ a $V > 101 - M$ dostáváme $V > D$, což je ve sporu s již dokázanou nerovností $V <= D$. Jestliže tedy proměnná I nabude hodnoty M, bude se při dalších průchodech cyklem zvětšovat hodnota proměnné J, dokud bude platit $A[M] = A[M + J]$. Vzhledem k rovnosti $A[M] = A[M + D - 1]$ získá proměnná J hodnotu D. Již jsme dokázali, že $V <= D$, takže J nemůže nabýt hodnoty větší. Je tedy $V = D$, což jsme měli dokázat.

P-I-4

a) Při řešení úlohy využijeme skutečnosti, že mnohoúhelník je konvexní právě tehdy, jestliže všechny jeho vnitřní úhly jsou menší nebo rovny 180 stupňů. Toto tvrzení lze snadno dokázat jednoduchou geometrickou úvahou. V algoritmu proto stačí zkontrolovat velikost všech vnitřních úhlů zadaného mnohoúhelníku. To lze velice pohodlně provést pomocí předdefinované funkce *UHEL*.

Předpokládejme, že v proměnné N je uložen počet vrcholů zadaného mnohoúhelníku a v polích X, Y souřadnice jeho

vrcholů v pořadí $(X[1], Y[1]), (X[2], Y[2]), \ldots, (X[N], Y[N])$. Algoritmus řešící zadanou úlohu potom můžeme zapsat v Pascalu takto:

```
X[N + 1] : = X[1]; Y[N + 1] : = Y[1];
X[N + 2] : = X[2]; Y[N + 2] : = Y[2];
I : = 1;
while (I < = N)
  and (UHEL (X[I + 2], Y[I + 2], X[I + 1], Y[I + 1],
  X[I], Y[I]) < = 180) do
    I : = I + 1;
if I > N then writeln ('Mnohoúhelník je konvexní.')
         else writeln ('Mnohoúhelník není konvexní.');
```

Abychom nemuseli zvlášť řešit situaci pro první a poslední vrchol mnohoúhelníku (tj. vrcholy s indexy 1 a N), přiřadili jsme před zahájením vlastního výpočtu souřadnice vrcholů $(X[1], Y[1])$ a $(X[2], Y[2])$ do polí X, Y ještě jednou do položek s indexy $N + 1$, $N + 2$.

Správnost algoritmu přímo vyplývá z uvedeného rozboru. Je třeba si uvědomit, že podle definice mnohoúhelníku jsou jeho vrcholy zadány v pořadí proti směru hodinových ručiček a že funkce UHEL dává jako svůj výsledek velikost úhlu měřeného také proti směru hodinových ručiček. Pro vyjádření velikosti vnitřního úhlu mnohoúhelníku při vrcholu s indexem $I + 1$ je proto v zápisu algoritmu užito správného pořadí vrcholů ve volání funkce UHEL.

Algoritmus má lineární časovou složitost a výpočet podle něj je konečný, neboť každý vnitřní úhel je kontrolován nejvýše jednou. Vykoná se proto maximálně N kroků výpočtu,

popřípadě při nalezení nějakého vnitřního úhlu většího než 180 stupňů končí výpočet ještě dříve.

b) Popíšeme neformálně činnost algoritmu řešícího zadanou úlohu. Algoritmus nejprve zjistí jeden bod z konvexního obalu. Za tento bod zvolíme například ten ze zadaných bodů, který má největší x-ovou souřadnici. Je-li takových bodů více, vybereme z nich ten, který má největší y-ovou souřadnici. Takto získaný bod jistě náleží do konvexního obalu.

Známe-li již několik (třeba jen jeden) bodů konvexního obalu, další bod získáme následujícím výpočtem: Nechť P je bod naposledy zařazený do vytvářeného konvexního obalu a bod A libovolný jiný bod, který dosud do obalu nebyl zařazen (za »nezařazený« do konvexního obalu považujeme i počáteční bod s největší x-ovou souřadnicí). Pro všechny ostatní body nezařazené do konvexního obalu nyní budeme zjišťovat, zda některý z nich leží vpravo od přímky PA při pohledu z bodu P k bodu A. Pokud najdeme takový bod B, budeme nadále totéž zjišťovat pro bod B a přímku PB. Nemusíme ale již uvažovat body, které leží vlevo od přímky PA, neboť ty jistě leží vlevo i od přímky PB. Po otestování všech bodů nezařazených do konvexního obalu tedy získáme bod C takový, že žádný jiný z bodů nezařazených dosud do konvexního obalu neleží vpravo od přímky PC při pohledu z bodu P k bodu C. Tento bod C nyní zařadíme jako další bod do konvexního obalu. Je-li bod C shodný s počátečním bodem, algoritmus již nalezl celý konvexní obal. Jinak se výpočet dalšího bodu konvexního obalu opakuje.

Opět budeme předpokládat, že proměnná N udává počet zadaných bodů. Souřadnice těchto bodů jsou uloženy v polích X, Y v položkách s indexy $1, 2, \ldots, N$. Podle požadavků

úlohy budeme souřadnice bodů nalezeného konvexního obalu ukládat do polí *KONOBALX*, *KONOBALY* a počet bodů tohoto konvexního obalu do proměnné *POCET*.

Program bude využívat pomocné pole *OBAL*[1..*N*], ve kterém bude pro každý ze zadaných bodů zaznamenáno, zda byl zařazen do konvexního obalu. Pokud *OBAL*[*K*] = 0, bod (*X*[*K*], *Y*[*K*]) do konvexního obalu dosud nebyl zařazen, jestliže *OBAL*[*K*] = 1, uvedený bod do konvexního obalu zařazen byl. U počátečního (a tedy také koncového) bodu vytvářeného konvexního obalu je udržována v poli *OBAL* hodnota 0 a index tohoto bodu je zaznamenán ve zvláštní proměnné *PB*. Souřadnice tohoto bodu jsou vloženy do polí *KONOBALX*, *KONOBALY* až jako poslední.

```
program KONOBAL (input, output);
const MAX = 50;  {maximální přípustný počet všech bodů}

var X, Y: array [1..MAX] of real; {souřadnice zadaných
      bodů}
  KONOBALX, KONOBALY: array [1..MAX] of real;
                        {souřadnice bodů konvexního obalu}
  OBAL: array [1..MAX] of integer;
                        {indikace zařazení bodu do konvex. obalu}
  N, POCET, PB: integer;  {počet všech bodů, počet bodů
                           v konv. obalu, počáteční bod}
  I, K, B: integer;  {pomocné proměnné — indexy bodů}
begin
{Načtení vstupních dat:}
read (N);
for I := 1 to N do read (X[I], Y[I]);
```

```
{Hledání počátečního bodu:}
PB : = 1;
for I : = 2 to N do
  if X[I] > X[PB] then PB : = I
  else if (X[I] = X[PB] and (Y[I] > Y[PB]) then PB : = I


{Inicializace proměnných:}
for I : = 1 to N + 1 do OBAL[I] : = 0;
POCET : = 0;
K : = PB;


{Vlastní výpočet:}
repeat
  {K je zatím poslední bod konvexního obalu}
  B : = 1;
  while (OBAL[B] = 1) or (B = K) do B : = B + 1;
  {B označuje bod s nejmenším indexem nezařazený do obalu}
  I : = B + 1;
  while (OBAL[I] = 1) or (I = K) do I : = I + 1;
  {I je další adept na zařazení do konv. obalu}
  while I < = N do
    begin
    if VPRAVO (X[K], Y[K], X[B], Y[B], X[I], Y[I])
      then B : = I;
    repeat I : = I + 1
    until (OBAL[I] < > 1) and (I < > K);
    end;
  {B zde označuje další bod konvexního obalu}
  K : = B;
```

```
    POCET : = POCET + 1;
    KONOBALX [POCET] : = X[K];
    KONOBALY[POCET] : = Y[K];
    OBAL[K] : = 1
until K = PB;
{Vypsání výsledného konvexního obalu:}
writeln ('Konvexní obal je tvořen', POCET : 1,' body.');
writeln ('Souřadnice bodů konvexního obalu:');
for I : = 1 to POCET do
    writeln (KONOBALX[I] : 10, KONOBALY[I] : 10)
end.
```

Správnost popsaného algoritmu vyplývá přímo z výše uvedeného rozboru. Algoritmus je popsán induktivně, po krocích, a je ho také možné matematickou indukcí formálně dokázat. Z rozboru je zřejmá i konečnost výpočtu podle našeho algoritmu. V každém kroku výpočtu je přidán jeden bod do postupně vytvářeného konvexního obalu, konvexní obal N bodů je tvořen nejvýše těmito N body, takže výpočet skončí nejpozději po N krocích.

Popsaný algoritmus má kvadratickou časovou složitost. Vyžaduje provedení nejvýše N kroků, jak jsme právě ukázali. Přitom v každém kroku je každý z N zadaných bodů právě jednou testován, zda není prodloužením dosud nalezené části konvexního obalu. Celkový počet operací nezbytný k vyřešení úlohy je tedy úměrný hodnotě $N * N$.

Poznámka. V případě, že na hraně konvexního obalu leží tři body (nebo více bodů), zde uvedený algoritmus je všechny může ale nemusí zařadit do konvexního obalu. Protože oba

případy vyhovují definici konvexního obalu ze zadání úlohy, hlouběji se tímto případem nezabýváme.

P - II - 1

Hodnotu $f(k)$ pro nějaké pevně zvolené číslo $k > 1$ snadno vyjádříme pomocí dvou jiných hodnot funkce f. Přitom argumenty funkce f odpovídající těmto hodnotám jsou dvě po sobě následující čísla menší než k. Pokud je totiž k sudé, je $f(k) = 2.f(k/2)$, což můžeme zapsat také jako $f(k) = 2.f(k/2) + 0.f(k/2 + 1)$. Je-li k liché, potom platí rovnost $f(k) = 2.f((k + 1)/2) - f((k - 1)/2)$. Tyto vztahy jsme získali jednoduchou úpravou přímo z definice funkce f.

Nyní si všimneme, jak je možné vyjádřit hodnoty $f(k)$, $f(k + 1)$ pro dvě po sobě jdoucí celá čísla k, $k + 1$. Rozlišíme tři případy:

1. $k = 0 \ldots f(k) = f(k + 1) = 1$ přímo podle definice funkce f.
2. k sudé, $k > 0 \ldots f(k) = 2.f(k/2)$ podle definice funkce f, číslo $k + 1$ je liché, a proto $f(k + 1) = 2.f(k/2 + 1) - f(k/2)$.
3. k liché $\ldots f(k) = 2.f((k + 1)/2) - f((k - 1)/2)$ podle definice, číslo $k + 1$ je sudé, a proto $f(k + 1) = 2.f((k + 1)/2)$.

Tedy pro $k = 0$ jsou hodnoty $f(k), f(k + 1)$ známy a pro $k > 0$ je možné vyjádřit obě hodnoty $f(k), f(k + 1)$ opět jako vhodnou kombinaci hodnot funkce f odpovídajících dvěma menším po sobě jdoucím číslům v roli argumentů (a to konkrétně číslům k div 2, k div 2 + 1).

Z uvedeného rozboru vyplývá, že k výpočtu hodnoty $f(n)$ pro zadané číslo n nám budou stačit tři proměnné. V jedné proměnné K bude stále uložena postupně se zmenšující

hodnota argumentu — od počáteční vstupní hodnoty n až k nule, v každém kroku výpočtu celočíselně dělena dvěma. V dalších dvou proměnných A a B se budou počítat koeficienty, jimiž je třeba vynásobit hodnoty $f(K)$, $f(K + 1)$, abychom získali správný výsledek. Na začátku výpočtu bude $K = n, A = 1, B = 0$. Hodnota výrazu $A.f(K) + B.f(K + 1)$ bude během celého výpočtu udržována konstantní. Na začátku výpočtu má tento výraz hodnotu $1.f(n) + 0.f(n + 1) = f(n)$, po ukončení při $K = 0$ bude mít tvar $A.f(0) + B.f(1) = A + B$. Výslednou hodnotu $f(n)$ lze tedy získat jako součet hodnot proměnných A a B po ukončení výpočtu.

Zbývá ukázat, jak musí vypadat přepočet hodnot proměnných K, A, B v každém kroku výpočtu, aby se zachovala konstantní hodnota výrazu $A.f(K) + B.f(K + 1)$. Využijeme k tomu dříve odvozené vztahy pro vyjádření hodnot $f(k)$, $f(k + 1)$:

a) je-li hodnota proměnné K sudá kladná:

$A.f(K) + B.f(K + 1) = A.2.f(K/2) + B.2.f(K/2 + 1) -$
$- B.f(K/2) = (2A - B).f(K/2) + 2B.f(K/2 + 1)$

— tedy hodnota K se zmenší na polovinu, proměnná A získá hodnotu $2A - B$ a hodnota proměnné B se zdvojnásobí;

b) je-li hodnota proměnné K lichá:

$A.f(K) + B.f(K + 1) = A.2.f((K + 1)/2) -$
$- A.f((K - 1)/2) + B.2.f((K + 1)/2) =$
$= -A.f((K - 1)/2) + 2.(A + B).f((K + 1)/2)$

— tedy hodnota proměnné K se zmenší na $(K - 1)/2$ neboli K div 2, hodnota proměnné A změní znaménko a proměnná B získá hodnotu $2(A + B)$.

Na základě uvedeného rozboru snadno zapíšeme hledaný algoritmus na výpočet hodnoty funkce f pro daný argument n.

Algoritmus vyjádříme ve tvaru funkce v programovacím jazyce Pascal:

```
function F (K : integer): integer;

var A, B: integer;

begin
A : = 1;
B : = 0;
while K > 0 do
  begin
  if K mod 2 = 0 then
    begin
    A : = 2 * A — B;
    B : = 2 * B
    end
  else
    begin
    B : = 2*(A + B);
    A : = —A
    end;
  K : = K div 2
  end;
F : = A + B
end;
```

Správnost uvedeného algoritmu vyplývá z rozboru úlohy. Během celého výpočtu se udržuje neměnná hodnota výrazu $A.f(K) + B.f(K + 1)$, přičemž při vyvolání funkce s argumentem n vyjadřuje tento výraz hledanou hodnotu $f(n)$. Po

ukončení výpočtu while-cyklu v programu je $K = 0$ a podle definice funkce f je $f(0) = f(1) = 1$, takže hodnotu uvedeného výrazu vyjadřuje součet hodnot proměnných $A + B$. Tento součet je proto správným výsledkem algoritmu.

Výpočet probíhající podle našeho algoritmu je jistě konečný, neboť v každém kroku výpočtu se hodnota proměnné K celočíselně dělí dvěma a celý výpočet končí, jakmile K dosáhne nulové hodnoty.

Algoritmus splňuje požadavek ze zadání úlohy na minimální paměťovou náročnost. K celému výpočtu stačí použít pouze tři proměnné. Algoritmus má logaritmickou časovou složitost a konstantní paměťovou složitost.

P - II - 2

Navrhnout nějaký, byť pomalý a neefektivní algoritmus řešící zadanou úlohu je velmi snadné. Stačí zkoumat postupně všechny obdélníky v dané matici, zda jsou tvořeny samými jedničkami. Obdélník vždy vymezíme volbou jeho levého horního a pravého dolního rohu. Takovéto řešení má ovšem časovou složitost $N^3 \cdot M^3$, neboť pro volbu levého horního rohu máme $N.M$ možností, pro volbu pravého dolního rohu také (při již zvoleném levém horním rohu je zde možností o něco méně, ale z hlediska časové složitosti algoritmu to není podstatná úspora) a průchod zvoleným obdélníkem v matici o rozměrech $N \times M$ představuje také řádově $N.M$ operací.

Krátký program řešící úlohu tímto neefektivním způsobem může vypadat následovně:

program *OBDEL* (input, output);

```
const N = 20;              {počet řádků matice A}
      M = 15;              {počet sloupců matice A}

var A : array [1..N, 1..M] of integer;
    I, J, K, L, P, Q : integer; {pracovní indexy}
    MI, MJ, MK, ML : integer; {souřadnice max. obdél.}
    C: integer;          {pro kontrolu obdélníku}
    VEL: integer;        {velikost zkoumaného obdélníku}
    MAX: integer;        {maximální velikost obdélníku}

begin
MAX := 0;
for I := 1 to N do {načtení hodnot matice A}
  for J := 1 to M do read(A[I, J]);
for I := 1 to N do
  for J := 1 to M do {levý horní roh obdélníku A[I, J]}
    for K := I to N do
      for L := J to M do          {pravý dolní roh A[K, L]}
        begin
        VEL := (K - I + 1) * (L - J + 1);
        if VEL > MAX then
          begin          {obdélník je větší než maximální}
          C := 1;
          for P := I to K do
            for Q := J to L do
              C := C * A[P, Q];     {kontrola obdélníku}
          if C = 1 then
            begin          {obdélník je tvořen jedničkami}
            MAX := VEL;
```

```
            MI := I; Mj := j; MK := K; ML := L
          end
        end
      end;

if MAX = 0 then
  writeln ('Zadaná matice neobsahuje žádnou jedničku.')
else
  begin
  writeln ('Maximální obdélník tvořený jedničkami má');
  writeln ('souřadnice levého horního rohu:', MI:7, Mj:6);
  writeln ('souřadnice pravého dolního rohu:', MK:6, ML:6)
  end
end.
```

Ukažme si nyní jiný algoritmus, který řeší zadanou úlohu výrazně rychleji. V první fázi řešení provedeme pomocný výpočet, při kterém určíme délky souvislých sloupců jedniček v dané matici A. Výsledky tohoto výpočtu si uložíme přímo do pole A tak, že položíme $A[i,j] = k$, jestliže prvek $A[i,j]$ sám a dalších přesně $k - 1$ prvků pod ním mělo původně hodnotu 1, tzn. jestliže v původní matici A platilo $A[p,j] = 1$ pro $p = i, i + 1, \ldots, i + k - 1$ a navíc buď $i + k - 1 = N$ nebo $i + k - 1 < N$ a přitom $A[i + k, j] = 0$ (kde N je počet řádků matice A). Údaje v zadaném poli A tím pozměníme, ale pouze tak, že v případě potřeby by bylo snadné zrekonstruovat původní podobu pole A (neboť žádná nula v poli A neubyla ani nepřibyla, nenulová čísla jsou uložena

na místech původních jedniček). Výsledek první pomocné fáze výpočtu si ukážeme na příkladu:

ze zadané matice:					dostaneme upravenou matici:				
1	1	0	1	0	3	4	0	1	0
1	1	1	0	1	2	3	2	0	3
1	1	1	1	1	1	2	1	1	2
0	1	0	0	1	0	1	0	0	1

Ve druhé fázi výpočtu již budeme hledat v poli A maximální obdélník tvořený jedničkami (nyní po úpravě nenulovými čísly). Postupně budeme zkoumat všechny možné pozice levého horního rohu takového obdélníku. Pro zvolený levý horní roh $A[i,j] > 0$ musíme vyzkoušet všechny přípustné polohy pravého horního rohu $A[i,l]$. Prvek $A[i,l]$ může být pravým horním rohem obdélníku s levým horním rohem $A[i,j]$, jestliže všechna čísla $A[i,q]$ pro $q = j, j+1, \ldots, l$ jsou nenulová.

Velikost maximálního obdélníku, který je v původní matici A tvořen samými jedničkami a jehož levý a pravý horní roh mají souřadnice $[i,j]$, resp. $[i,l]$, nyní již snadno určíme pomocí hodnot, které jsme si předem připravili v první fázi výpočtu. Takový obdélník má totiž šířku $(l - j + 1)$ a jeho výška je rovna minimu z hodnot $A[i,q]$ pro $q = j, j+1, \ldots, \ldots, l-1, l$.

Uvedený výpočet je možné opakovat pro všechny možné volby levého horního rohu obdélníku a přitom si v pomocné proměnné udržovat velikost maximálního již nalezeného obdélníku tvořeného v zadané matici samými jedničkami.

V dalších čtyřech pomocných proměnných si musíme zaznamenávat souřadnice levého horního a pravého dolního rohu nalezeného maximálního obdélníku. Tyto proměnné budou po ukončení výpočtu udávat požadovaný výsledek úlohy.

```
program OBDELNIK (input, output);

const N = 20;     {počet řádků matice A}
      M = 15;     {počet sloupců matice A}
var   A: array [1..N, 1..M] of integer;
      I, J, L: integer;        {pracovní indexy v poli A}
      K: integer;              {výška zkoumaného obdélníku}
      MI, MJ, MK, ML: integer; {souřadnice rohů max.
                                obdélníku}
      MAX: integer;            {velikost maximálního obdélníku}
      VEL: integer;            {velikost zkoumaného obdélníku}

begin
{Načtení matice A:}
for I := 1 to N do
  for J := 1 to M do read (A[I,J]);

{První fáze výpočtu — modifikace pole A:}
for J := 1 to M do
  for I := N - 1 downto 1 do
    if A[I,J] = 1 then A[I,J] := A[I,J] + A[I + 1,J];

{Druhá fáze výpočtu — hledání maximálního obdélníku:}
MAX := 0;
```

```
for I := 1 to N do
  for J := 1 to M do
    begin                    {levý horní roh A[I,J]}
    L := J;                  {pravý horní roh A[I,L]}
    K := A[I,L];
    while K > 0 do
      begin
      VEL := (L - J + 1) * K;   {velikost zkoumaného
                                 obdélníku}
      if VEL > MAX then
        begin                {je větší než dosud maximální}
        MAX := VEL; {- > zaznamenat jeho velikost}
        MI := I; MJ := J; MK := I + K - 1; ML := L
                        {- > zaznamenat jeho souřadnice}
        end;
      L := L + 1;            {nová poloha prav. horního rohu}
      if L > M then
        K := 0               {už jsme vně matice A}
      else if A[I,L] < K then
        K := A[I,L]          {snížení výšky obdélníku z »1«}
      end
    end;


{Vypsání výsledku výpočtu:}
if MAX = 0 then
  writeln ('Zadaná matice neobsahuje žádnou jedničku.')
else
  begin
  writeln ('Maximální obdélník tvořený jedničkami má');
```

```
    writeln ('souřadnice levého horního rohu:', MI : 7, Mj : 6);
    writeln ('souřadnice pravého dolního rohu:',MK : 6, ML : 6)
  end
end.
```

Správnost algoritmu plyne z uvedeného rozboru. Pokud zadaná matice obsahuje samé nuly, zůstane ve druhé fázi výpočtu proměnná *MAX* s počáteční hodnotou 0 nezměněna a na základě toho je vypsáno příslušné hlášení. Jestliže matice obsahuje alespoň jednu jedničku, musí obsahovat také nějaký maximální obdélník tvořený jedničkami. Dvojice proměnných *I*, *J* během výpočtu nabude hodnot odpovídajících souřadnicím levého horního rohu tohoto maximálního obdélníku, neboť pomocí indexů *I*, *J* algoritmus postupně prochází všechny prvky pole *A*. Proměnná *L* potom jistě nabude také hodnoty sloupcového indexu pravého horního rohu maximálního obdélníku z jedniček a přitom bude proměnná *K* udávat výšku tohoto obdélníku. Za této situace získá proměnná *MAX* hodnotu udávající velikost maximálního obdélníku tvořeného jedničkami (pokud této hodnoty nenabyla již dříve při zkoumání jiného obdélníku stejné velikosti tvořeného samými jedničkami). Zároveň jsou do proměnných *MI*, *MJ*, *MK*, *ML* zaznamenány souřadnice levého horního a pravého dolního rohu nalezeného maximálního obdélníku. Vzhledem k maximalitě velikosti tohoto obdélníku již nemůže být nalezen žádný větší obdélník tvořený samými jedničkami a hodnoty proměnných *MAX*, *MI*, *MJ*, *MK*, *ML* proto zůstanou až do ukončení výpočtu nezměněny. Při ukončení výpočtu jsou proto [*MI*, *MJ*] a [*MK*, *ML*] souřadnice levého horního a pravého dolního rohu maximálního obdélníku

tvořeného samými jedničkami. Jestliže lze v zadané matici A nalézt více různých obdélníků ze samých jedniček této maximální velikosti, budou proměnné MI, MJ, MK, ML udávat souřadnice rohů jednoho z nich (toho, který byl nalezen jako první).

Výpočet podle uvedeného algoritmu je jistě konečný, neboť počet průchodů každým z cyklů v programu je předem omezen některým z rozměrů zadané matice. Načtení hodnot matice A ze vstupu a modifikace obsahu pole A v první fázi výpočtu vyžadují provedení $N.M$ operací. Ve druhé fázi výpočtu se $N.M$ způsoby volí levý horní roh zkoumaného obdélníku a pro každou takovou volbu se provádí nejvýše M průchodů vnitřním while-cyklem s vlastním vyhodnocením obdélníků. Počet operací potřebných k provedení celého výpočtu je proto úměrný hodnotě $N.M.M$.

Celý výpočet je možné provádět také symetricky tak, že se v první fázi určí délky souvislých vodorovných řad jedniček v poli A. Ve druhé fázi výpočtu by se potom pro každou volbu polohy levého horního rohu zkoumaného obdélníku zkoušely všechny možné polohy jeho levého dolního rohu. V tomto případě by celý výpočet vyžadoval provedení řádově $N.N.M$ operací. Pokud bychom se tedy snažili o maximální možnou optimalizaci časových nároků navrženého algoritmu, bylo by možné volit uvnitř algoritmu až na základě znalosti konkrétních rozměrů N, M pole A tu variantu řešení, která by byla výhodnější (varianta, kterou jsme zde podrobně rozebrali, je výhodnější pro $N > M$).

Rozborem zadaného algoritmu *P* snadno zjistíme, že činnost algoritmu můžeme popsat následovně: Zadané pole *A* se prochází sekvenčně zleva doprava. Při tomto průchodu se zapamatuje a odstraní z pole *A* nejprve první prvek. Tento prvek bude »přenesen« bezprostředně za souvislý úsek po něm následujících čísel menších, než je on sám. Celý tento úsek čísel se v poli *A* posune o jednu pozici doleva. Jakmile se při průchodu polem narazí na nějaké číslo větší, než je právě zapamatovaný a přenášený prvek, tento přenášený prvek se umístí do pole *A*. Místo něj se zapamatuje nalezené větší číslo, odstraní se z pole *A* a opět se bude přenášet stejným způsobem dál. Tento postup se opakuje tak dlouho, dokud se algoritmus nedostane k pravému okraji pole *A*. Tam je pak umístěn naposledy přenášený prvek a tím je celý výpočet ukončen.

Z rozboru je zřejmé, že každý prvek je umístěn na své výsledné místo v poli *A*, jestliže

— buď je posunut o jedno místo doleva a přes něj je přeneseno právě zapamatované větší číslo;
— nebo byl přenášen směrem doprava přes úsek menších čísel (tento úsek může být i prázdný) a nyní je umisťován za tento úsek, neboť algoritmus narazil na větší číslo uložené v poli *A* nebo na pravý okraj pole *A*.

V průběhu výpočtu jsou umisťovány na svá výsledná místa prvky pole *A* v takovém pořadí, že jsou postupně obsazována jednotlivá místa v poli *A* zleva doprava. Po ukončení práce algoritmu bude tedy pole *A* vzestupně uspořádáno právě teh-

dy, jestliže jsou čísla umisťována na svá výsledná místa v poli A v neklesajícím pořadí.

Na základě rozboru zadaného algoritmu a s využitím právě uvedeného tvrzení nyní již můžeme zapsat program řešící danou úlohu. Program »modeluje« výpočet algoritmu P, ale do pole A nezasahuje a pouze si udržuje informaci o hodnotě právě zapamatovaného a přenášeného prvku a o dosud největším již umístěném prvku pole A. Zároveň s napodobováním výpočtu algoritmu P bude náš program sledovat, zda čísla umisťovaná do pole A následují po sobě v neklesajícím pořadí.

```
program TRIDENI (input, output);

const N = 100;                          {velikost pole A}
var A: array [1..N] of integer;         {zadané pole A}
    PREN: integer;                      {přenášený prvek}
    MAX: integer;          {maximální již umístěný prvek}
    CH: Boolean;           {příznak chybného uspořádání}
    I: integer;            {pomocná proměnná — index v A}
begin
for I: = 1 to N do read (A[I]);         {načteni pole A}
CH: = false;                            {zatím chyba není}
PREN: = A[1];                  {první přenášený prvek}
I: = 2;                        {průchod od prvku A[I]}
MAX: = —maxint;                {inicializace maxima}

while (not CH) and (I < = N) do
  begin
  if A[I] < PREN then          {přenést PREN přes A[I]}
```

```
    if A[I] < MAX then
        CH: = true                {chyba v uspořádání!}
    else
      MAX: = A[I]                 {nová hodnota maxima}
  else
    begin                         {narazil na větší číslo než PREN}
      MAX: = PREN;                {dosud přenášený prvek PREN bude umístěn do pole A a stane se tak novým maximem z umístěných čísel}
    PREN: = A[I]                  {zapamatuje se následující číslo a stane se tak novým přenášeným prvkem PREN}
    end;
  I: = I + 1                      {bude zkoumat další prvek A}
  end;

if CH then
  writeln ('Algoritmus zadané pole neuspořádá.')
else
  writeln ('Algoritmus zadané pole uspořádá.')
end.
```

Správnost programu vyplývá z provedeného rozboru úlohy. Náš program provádí zcela obdobný výpočet jako algoritmus P uvedený v zadání úlohy, pouze namísto vlastních výměn prvků uložených v poli *A* si v pomocných proměnných *PREN* a *MAX* udržuje potřebné řídicí informace. Pomocí proměnné *MAX* je zároveň kontrolováno, zda bude po ukončení výpočtu pole *A* vzestupně uspořádáno.

Výpočet programu je jistě konečný, neboť je tvořen maximálně N kroky (kde N je počet prvků pole A). Pokud se během výpočtu zjistí, že by pole A nebylo po ukončení výpočtu uspořádáno, je výpočet předčasně ukončen ještě dříve. Program má stejně jako zadaný algoritmus P lineární časovou složitost.

P - II - 4

Existuje celá řada v principu zcela odlišných algoritmů, které řeší tuto úlohu. Nejlepší z nich mají lineární časovou složitost, tzn. počet operací nezbytných k vyřešení úlohy je úměrný počtu vrcholů zadaného mnohoúhelníku. Namísto jednoho detailního vzorového řešení úlohy si proto raději ukážeme hlavní myšlenky tří různých algoritmů (s lineární časovou složitostí) a možnou programovou realizaci jednoho z nich.

1. Protože známe souřadnice všech vrcholů daného mnohoúhelníku i souřadnice zkoumaného bodu (označme ho B), můžeme snadno spočítat vzdálenosti bodu B od jednotlivých stran mnohoúhelníku. Zároveň můžeme zjistit, od které strany mnohoúhelníku má bod B nejmenší vzdálenost — nechť je to strana $A(k)\,A(k+1)$. Nejkratší spojnici bodu B se stranou mnohoúhelníku nemůže protínat (ani se jí dotýkat) žádná jiná strana mnohoúhelníku, která by bod B od strany $A(k)\,A(k+1)$ geometricky »oddělila«. K vyřešení úlohy proto stačí zjistit, zda bod B leží napravo nebo nalevo od úsečky $A(k)\,A(k+1)$. Přesněji řečeno, bod B leží uvnitř daného mnohoúhelníku právě tehdy, jestliže pomocná funkce VPRAVO zavolaná se souřadnicemi bodů $A(k+1)$, $A(k)$, B

na místě parametrů (v tomto pořadí) dává hodnotu »pravda«. Záměna pořadí bodů $A(k)$, $A(k+1)$ při vyvolání funkce VPRAVO je nutná z toho důvodu, že bod ležící na straně mnohoúhelníku musí být označen jako ležící uvnitř.

2. Ze zkoumaného bodu B povedeme polopřímku libovolným směrem takovým, aby na této polopřímce neležel žádný vrchol zadaného mnohoúhelníku. Zjistíme počet průsečíků této polopřímky se stranou mnohoúhelníku. Budeme-li se po polopřímce pohybovat ve směru od bodu B, každý průsečík polopřímky se stranou mnohoúhelníku bude znamenat přechod z oblasti uvnitř mnohoúhelníku do oblasti vně mnohoúhelníku nebo naopak. Volba takového směru polopřímky, aby na ní neležel žádný vrchol mnohoúhelníku, nám odstraní všechny nežádoucí případy, že by polopřímka mnohoúhelník pouze »tečovala« (dotkla by se ho v jednom bodě nebo v celé straně, ale ke změně oblasti by nedošlo). Polopřímka samozřejmě vede do oblasti vně mnohoúhelníku, neboť je nekonečná, zatímco mnohoúhelník je konečný útvar. Počet průsečíků polopřímky se stranami mnohoúhelníku proto jednoznačně určuje, zda se výchozí bod B nachází uvnitř nebo vně mnohoúhelníku. Je-li tento počet lichý, leží bod B uvnitř, je-li sudý, leží vně zadaného mnohoúhelníku.

3. Nejprve zvlášť vyšetříme, zda zkoumaný bod B není roven přímo některému z vrcholů $A(i)$ daného mnohoúhelníku. Pokud ano, leží B uvnitř mnohoúhelníku. Jestliže tomu tak není, má smysl hovořit o velikosti úhlů $A(i)BA(i+1)$, tzn. o velikosti úhlů, pod nimiž jsou z bodu B vidět jednotlivé strany mnohoúhelníku. Velikosti těchto úhlů budeme měřit podle konvence proti směru hodinových ručiček (podobně jako v pomocné funkci UHEL) a jejich hodnotu budeme udá-

vat ve stupních v rozmezí (−180; 180⟩. Jestliže budeme procházet po obvodu mnohoúhelníku z bodu $A(1)$ postupně přes body $A(2)$, $A(3)$, ... až zpět do bodu $A(1)$, tzn. také proti směru hodinových ručiček, a budeme přitom sčítat úhly měřené výše uvedeným způsobem, bude nám tento částečný součet pro bod $A(k)$ udávat velikost úhlu $A(1)BA(k)$. Hodnota součtu úhlů po projití celým obvodem mnohoúhelníku bude jednoznačně určovat polohu bodu B vůči zadanému mnohoúhelníku. Leží-li bod B uvnitř mnohoúhelníku, má výsledný součet úhlů hodnotu 360 stupňů, v opačném případě je nulový. Volbou hodnoty +180 stupňů pro přímý úhel je zajištěno, že bod B ležící na straně mnohoúhelníku je správně vyhodnocen jako ležící uvnitř (při volbě −180 stupňů by byl označen jako ležící vně).

Na závěr předvedeme programovou realizaci jednoho z algoritmů řešících danou úlohu. Pro tento účel zvolíme algoritmus popsaný výše jako 3. v pořadí. Program očekává na vstupu nejprve počet vrcholů mnohoúhelníku, po něm následují dvojice souřadnic jeho vrcholů a nakonec dvojice souřadnic zkoumaného bodu.

```
program UVNITR (input, output);
const MAX = 50; {maximální počet vrcholů mnohoúhelníku}
var AX, AY: array [1..MAX] of real;
        {x-ová a y-ová souřadnice vrcholů mnohoúhelníku}
    N: integer;                {počet vrcholů mnohoúhelníku}
    BX, BY: real;              {souřadnice zkoumaného bodu}
    SOUCET: real;              {součet velikostí úhlů}
    VYSLEDEK: Boolean;         {výsledná poloha bodu B}
    I: integer;                {pomocná proměnná}
```

```
function UHL (X1, Y1, X2, Y2, X3, Y3: real) : real;
  {předefinování pomocné funkce UHEL tak, aby dávala
  hodnoty ve stupních z rozmezí (−180; 180 > }
var U: real;
begin
U: = UHEL (X1, Y1, X2, Y2, X3, Y3);
if U > 180 then U: = U − 360;
UHL: = U
end;
begin
{Načtení vstupních hodnot:}
read (N);                              {počet vrcholů mnoh. }
for I: = 1 to N do read(AX[I], AY[I]); {souřadnice vrcholů}
read (BX, BY);                         {souřadnice bodu B}

{Kontrola, zda bod B nesplývá s vrcholem mnohoúhelníku:}
VYSLEDEK: = false;
I: = 1;
while not VYSLEDEK and (I < = N) do
  begin
  if (BX = AX[I]) and (BY = AY[I]) then
    VYSLEDEK: = true;
  I: = I + 1
  end;
{Sčítání velikostí úhlů pro celý obvod mnohoúhelníku:}
if not VYSLEDEK then
  begin
  SOUCET: = UHL (AX[N], AY[N], BX, BY,
                 AX[1], AY[1]);
  for I: = 1 to N − 1 do
```

```
    SOUCET: = SOUCET + UHL (AX[I], AY[I],
                            BX, BY, AX[I + 1], AY[I + 1]);
  if SOUCET = 360 then
    VYSLEDEK: = true
  end;
{Vypsání výsledku úlohy:}
if VYSLEDEK then
  writeln ('Bod leží uvnitř mnohoúhelníku.')
else
  writeln ('Bod leží vně mnohoúhelníku.')
end.
```

K uvedenému programu je třeba poznamenat, že při skutečné praktické realizaci zvoleného algoritmu je nutné provádět jiným způsobem testování rovnosti dvou reálných čísel. Zde jsme pro zjednodušení a pro přehlednost uvedli příslušné testy ve tvaru

if $(BX = AX[I])$ **and** $(BY = AY[I])$ **then** ...

a

if $SOUCET = 360$ **then** ...

Tento tvar ovšem není vhodný pro výpočet, neboť reálná čísla jsou v počítači zobrazena přibližně. Místo přímého testu na rovnost dvou reálných čísel je proto třeba zkoumat, zda se obě čísla od sebe příliš neliší, například takto:

if $(\text{abs}(BX - AX[I]) < EPS)$ **and** $(\text{abs}(BY - AY[I]) < EPS)$ **then** ... pro vhodnou velmi malou konstantu *EPS*. Druhou z uvedených podmínek je v tomto konkrétním případě možné zapsat také jednodušeji například jako

if $SOUCET > 300$ **then** ... ,

neboť víme, že po ukončení výpočtu nabude proměnná *SOUCET* jedné z dvou možných hodnot, 360 nebo 0.

P - III - 1

Jistě existuje nějaká konečná posloupnost operací sjednocení a rozdělení, která převede P na Q. Jednou z možných cest je například nejprve sjednotit všechny množiny Pi, $i = 1, \ldots, m$, a takto vzniklou množinu potom postupně rozdělit na množiny Qj, $j = 1, \ldots, n$. Existuje proto také nějaká nejkratší posloupnost operací převádějící P na Q.

Nejprve ukážeme, že v takové nejkratší posloupnosti operací mohou být nejprve provedeny všechny operace sjednocení a potom všechny operace rozdělení. K tomu stačí dokázat následující tvrzení: Předchází-li v nejkratší možné posloupnosti operací převádějící P na Q nějaká operace rozdělení operaci sjednocení, lze pořadí těchto dvou operací zaměnit (přitom se pochopitelně mohou změnit i množiny, s nimiž se operace provádějí). Platnost tvrzení dokážeme rozborem případů. Nechť je v uvažované posloupnosti operací rozdělení množiny A na množiny $A1$, $A2$ následováno sjednocením množin B, C na množinu D.

1. Pokud se žádná z množin $A1$, $A2$ nerovná žádné z množin B, C, jsou obě operace zcela nezávislé, a můžeme proto zaměnit jejich pořadí bez vlivu na výsledek.

2. Pokud $\{A1, A2\} = \{B, C\}$, pak se provedením uvažované dvojice operací obnoví přesně stejný stav souboru množin, jaký byl před provedením těchto operací. Obě operace

je tudíž možné vynechat, což je spor s minimalitou délky naší posloupnosti operací. Tento případ tedy nemůže nastat.

3. Jestliže například množiny $A2$, C jsou shodné a množiny $A1$, B různé (obdobně pro ostatní kombinace), můžeme místo naší dvojice operací provést nejprve sjednocení množin A, B a v dalším kroku takto vzniklou množinu rozdělit na $A1$, D. Výsledný soubor množin bude stejný jako u původní posloupnosti operací.

Jiný případ nemůže nastat. Dokázali jsme tedy, že existuje nejkratší posloupnost operací převádějící P na Q, ve které se nejprve provedou všechna sjednocení a potom rozdělení množin.

Množiny $P1$, $P2$, ..., Pm budeme nejprve sjednocovat, až dostaneme soubor množin $A = \{A1, A2, \ldots, Ar\}$, $n <= r <= m$. K tomu je třeba provést $m - r$ operací sjednocení. Potom množiny tohoto souboru rozdělíme na množiny $Q1$, $Q2$, ..., Qn provedením $n - r$ operací rozdělení. Celkem se tedy provede $K = (m + n) - 2r$ operací. Hodnota $m + n$ je pevně dána zadanými soubory množin P a Q. Musíme proto hledat co největší hodnotu r. Pro další úvahy budeme zatím předpokládat, že všechny množiny $P1$, ..., Pm, $Q1$, ..., Qn jsou neprázdné. Úpravu řešení pro případ výskytu prázdných množin v zadaných souborech P a Q provedeme v závěru.

Každá množina Ai ($i = 1, 2, \ldots, r$) je sjednocením několika množin souboru P (neboť tak vznikla) a lze ji vyjádřit také jako sjednocení několika množin souboru Q (tak se bude dělit). Pokud mají množiny Pi, Pj s množinou Qh neprázdný průnik, musí být Pi, Pj spojeny v jedné z množin souboru A. Naopak, bude-li uvedená podmínka splněna, bude každá množina Qh podmnožinou nějaké množiny souboru A, a bude

tedy možné získat Q z A operacemi rozdělení. Nebudeme provádět více sjednocení, než je nezbytně nutné (tj. než uvádí tato podmínka), neboť naší snahou je minimalizovat počet všech operací neboli získat soubor A tvořený co největším počtem r množin.

Navrhneme nyní algoritmus na určení, které z množin souboru P je nutné sjednotit vždy do jedné množiny souboru A. Pro tento účel je možné převést naši úlohu na jeden standardní grafový algoritmus. Soubory množin P a Q můžeme reprezentovat grafem G o $m + n$ vrcholech odpovídajících jednotlivým množinám. Vrcholy grafu Pi a Qj jsou spojeny hranou právě tehdy, jestliže množiny Pi a Qj mají neprázdný průnik. Jiné hrany v grafu G nejsou (jedná se tedy o tzv. bipartitní graf). Podle výše uvedené podmínky na sjednocování množin souboru P je nutné spojit do jedné množiny souboru A vždy právě ty množiny, které patří do jedné komponenty souvislosti grafu G (tzn. ty množiny, které jsou v grafu G spojeny hranami). Hledaný počet množin r souboru A je proto roven počtu komponent souvislosti grafu G.

Algoritmus na nalezení komponent souvislosti daného grafu je dobře znám jako jeden ze standardních algoritmů teorie grafů, a nebudeme ho zde tudíž podrobněji rozebírat. Je vysvětlen v každé základní učebnici teorie grafů. Struktura algoritmu je velmi podobná algoritmu nalezení cesty mezi dvěma vrcholy grafu, který je uveden jako vzorové řešení úlohy P — I — 1.

Soubory množin P, Q můžeme v programu reprezentovat maticí logických hodnot $T[1..m, 1..n]$ takovou, že
$T[i,j] = 1$, jestliže Pi, Qj mají společný prvek,
$T[i,j] = 0$, jestliže Pi, Qj nemají společný prvek.

Matice T má minimální možnou velikost postačující k uložení všech informací o vzájemných vztazích množin souborů P a Q. Zároveň je vhodnou částí matice sousednosti grafu G obsahující všechny potřebné informace o grafu G. Na základě matice T určíme počet komponent souvislosti r grafu G. Výsledný minimální počet operací potřebných k převedení souboru množin P na soubor Q je potom dán výrazem $m + n - 2r$.

Zbývá dořešit případ, že se v souborech P a Q mohou vyskytovat prázdné množiny. Nechť soubor P obsahuje p prázdných množin a soubor Q obsahuje q prázdných množin. Potom platí $K = (m - p) + (n - q) - 2r + |p - q|$, kde číslo r sestrojíme výše uvedeným postupem ze systémů P', Q' vzniklých z P, Q vynecháním prázdných množin. K převedení p prázdných množin na q pak potřebujeme provést ještě $|p - q|$ operací: jestliže $p >= q$, bude to $p - q$ sjednocení, pokud $p < q$, provedeme $q - p$ rozdělení.

P - III - 2

Podle zadání úlohy má každá jednička v matici A (případně s výjimkou těch, které jsou na okraji matice) právě dva jedničkové sousední prvky, kde za sousední považujeme pouze pole ve směru vlevo, vpravo, nahoru a dolů. Tak vznikají v matici A »čáry« tvořené jedničkami, které celou plochu matice A dělí na jednotlivé oblasti tvořené nulovými prvky. Jedna z oblastí je zvolena k vybarvení tím, že je zadán jeden její prvek $A[I, J]$.

Dva nulové prvky matice A patří do téže oblasti, jestliže nejsou odděleny žádnou »čarou« z jedniček. Z toho vyplývá,

že víme-li o prvku $A[K, L] = 0$, že patří do zvolené oblasti, pak do této oblasti budou patřit také všechny nulové prvky pole A, které s $A[K, L]$ sousedí ve směru svislém, vodorovném nebo šikmém. Naopak, jestliže prvek $A[K, L] = 0$ (různý od prvku $A[I, J]$) nesousedí ve směru svislém, vodorovném ani šikmém s žádným prvkem oblasti zadané prvkem $A[I, J]$, pak $A[K, L]$ do této oblasti nepatří.

Z uvedeného rozboru plyne následující jednoduchý algoritmus. Vezmeme zadaný prvek $A[I, J]$ a obarvíme ho (tzn. dosadíme do něj 2). Potom prohlédneme všech osm prvků, s nimiž sousedí, a obarvíme ty z nich, které jsou dosud nulové. Pro každý z takto nově obarvených prvků celý postup opakujeme. Výpočet a obarvování probíhá tak dlouho, dokud je možné obarvit nějaký další prvek matice A.

Algoritmus je možné realizovat pomocí rekurzívní procedury nebo třeba pomocí zásobníku. Ukážeme si zde obě řešení. První z nich má kratší a přehlednější zápis, druhé bude pracovat o něco rychleji.

Řešení pomocí rekurze

```
program BARVENI 1 (input, output);

const M = 10;                {počet řádků matice}
      N = 10;                {počet sloupců matice}
var A: array [1..M, 1..N] of integer;
    K, L: integer;           {pomocné proměnné}
    I, J: integer;           {zadané výchozí pole}

procedure VYBARVI (X, Y: integer);
  {vybarvení prvku A[X, Y] a rekurzívní vyvolání téže procedury na všechny prvky v okolí}
```

```
var K, L: integer;          {pomocné proměnné}
begin
A[X, Y]: = 2;               {obarvení prvku A[X, Y]}
for K: = X − 1 to X + 1 do
  for L: = Y − 1 to Y + 1 do
    if (K > = 1) and (K < = M) and (L > = 1) and
    (L < = N) then
        if A[K, L] = 0 then VYBARVI (K, L)
end; {VYBARVI}

begin
for K: = 1 to M do
  for L: = 1 to N do
    read (A[K, L]);         {přečtení obsahu pole A}
read (I, J);                {přečtení výchozího prvku}
VYBARVI (I, J);             {obarvení pole od prvku A[I, J]}
for K: = 1 to M do
  begin
  writeln;
  for L: = 1 to N do
    write (A[K, L] : 2) {vytisknutí obarveného A}
  end;
writeln
end.
```

Řešení pomocí zásobníku

```
program BARVENI 2 (input, output);

const M = 10;               {počet řádků}
```

```
    N = 10;              {počet sloupců}
    MAX = 100;           {velikost zásobníku — max. M × N}

type SOUR = record X, Y: integer end;
                         {souřadnice prvku ukládané do zásobníku}
var A: array [1..M, 1..N] of integer;
    K, L: integer;       {pomocné proměnné}
    I, J: integer;       {zadání výchozího prvku}
    X, Y: integer;       {souřadnice řešeného prvku}
    STACK: array [1..MAX] of SOUR;
                         {zásobník souřadnic}
    SP: integer;         {ukazatel vrcholu zásobníku STACK}

begin
for K: = 1 to M do
  for L: = 1 to N do
    read(A[K, L]);       {přečtení obsahu pole A}
read (I, J);             {přečtení výchozího prvku}

STACK [1].X : = I;
STACK [1]. Y : = J;
SP : = 1;                {inicializace zásobníku}

while SP > 0 do
  begin
  X: = STACK [SP].X;
  Y: = STACK [SP]. Y;
  SP: = SP — 1;          {odebrán vrchní prvek ...}
  A [X, Y] : = 2;        {... a obarven na »2«}
```

```
  for K: = X − 1 to X + 1 do
    for L: = Y − 1 to Y + 1 do
      if (K >= 1) and (K <= M) and (L >= 1) and
      (L <= N) then
        if A[K, L] = 0 then
        begin                {sousední prvek je nulový}
        SP: = SP + 1;
        STACK[SP].X: = K;
        STACK[SP].Y: = L; {... uložen do zásobníku}
        end
  end;
for K: = 1 to M do

  begin
  writeln;
  for L: = 1 to N do
    write (A[K, L] : 2)  {vytisknutí obarveného A}
  end;
writeln

end.
```

Správnost navrženého algoritmu snadno ukážeme s použitím rozboru uvedeného na začátku řešení. Při práci algoritmu bude obarven prvek $A[I, J]$ a dále všechny další prvky matice A, které sousedí s některým dříve obarveným prvkem. V okamžiku, kdy již není možné obarvit žádný další prvek a kdy tedy výpočet podle algoritmu končí, bude proto obarvena právě celá oblast původně nulových prvků matice A určená prvkem $A[I, J]$. Přesně to bylo naším úkolem.

Výpočet podle algoritmu je jistě konečný, neboť v každém kroku algoritmu je obarven jeden původně nulový prvek matice A. Tato matice má konečnou velikost, a tedy také konečný počet všech nulových prvků.

Algoritmus má maximální časovou složitost úměrnou hodnotě $M.N$ neboli počtu prvků matice A. Vyžaduje totiž provedení tolika kroků výpočtu, kolik je v zadané matici A nulových prvků v oblasti určené prvkem $A[I, J]$. Řádově rychlejší algoritmus řešící zadanou úlohu nemůže existovat, neboť při libovolném postupu řešení musí být obarven každý nulový prvek zvolené oblasti a těch může být až $M.N$. V praxi se ovšem používají jiné algoritmy, které jsou sice složitější z hlediska zápisu a vysvětlení, ale mají alespoň v průměrném případě (nikoli v nejhorším) značně menší paměťové nároky a jsou i o něco rychlejší. Zde popsaný algoritmus totiž může navštívit a testovat jeden prvek matice A až devětkrát (jednou při vlastním obarvování prvku a osmkrát při obarvování jeho sousedů) a tento počet je možné vhodnou organizací výpočtu snížit.

P - III - 3

Věta: Je-li před prvkem $A[I]$, $1 < I <= N$, tzn. na místech v poli A s indexem menším než I, celkem P čísel větších, než je $A[I]$, pak se tato čísla dostanou za $A[I]$ po právě P průchodech vnějšího cyklu algoritmu.

Důkaz: Při každém průchodu se za $A[I]$ dostane právě jedno z takových čísel.

a) Nemůže jich být více, neboť s číslem $A[I]$ se provede pouze jedna výměna, kterou se číslo $A[I]$ dostane v poli A o jedno místo dopředu na pozici $A[I - 1]$. Na tomto novém místě může být naše číslo testováno opět až v dalším průchodu cyklem.

b) Jedno z P čísel (pro $P > 0$) větších než $A[I]$ se za $A[I]$ jistě dostane. Bude to číslo $A[S] = \max \{A[1], \ldots, \ldots, A[I - 1]\}$, neboť pro tento prvek $A[S]$ bude vždy splněna podmínka v programu, takže se při jednom průchodu cyklem dostane až na pozici $A[I - 1]$, a protože je větší než $A[I]$, bude ještě v témže průchodu cyklem vyměněno i s číslem $A[I]$.

Při prvním průchodu cyklem tedy zůstane v poli A před naším sledovaným číslem ještě $P - 1$ větších čísel a celý postup se bude opakovat. Přesně po P průchodech se všechna čísla větší než sledované číslo dostanou v poli A za něj, což jsme měli dokázat.

K vzestupnému setřídění celého pole A je třeba, aby se za každý prvek pole A dostala všechna větší čísla, která jsou na začátku práce algoritmu umístěna před ním. Podle dokázané věty každý prvek pole A vyžaduje, aby počet průchodů K byl větší nebo roven počtu prvků pole A s menšími indexy, které jsou větší než on sám. Pro takové K budou před každým prvkem stát pouze prvky menší nebo stejné, což je přesně vyjádření vzestupného uspořádání prvků pole A.

V hledaném algoritmu tedy stačí pro každý prvek pole A zjistit, kolik prvků pole A s menším indexem je větších než on sám, a maximum z těchto čísel vzít za hodnotu K.

```
program TRID (input, output);

const N = 10;                          {velikost pole A}
var A: array [1..N] of integer;        {tříděné pole A}
    I, J, K, L: integer;               {pomocné proměnné}

begin
for I: = 1 to N do read (A[I]);        {počáteční hodnoty pole A}
K: = 0;                                {hledaná hodnota K}
for I: = 2 to N do
  begin                                {zkoumáme prvek A[I]}
  L: = 0;                              {počet větších předchůdců}
  for J: = 1 to I - 1 do
    if A[J] > A[I] then L: = L + 1;
                                       {další větší předchůdce}
  if L > K then K: = L
  end;
writeln (K)                            {tisk výsledku}

end.
```

Správnost algoritmu plyne z provedeného rozboru. Konečnost výpočtu je zřejmá, neboť počet průchodů každým cyklem je předem omezen. Algoritmus má konstantní paměťové nároky (nepočítáme-li zadané pole A, které se nesmí měnit) a kvadratickou časovou složitost.

P - III - 4

a) Nejrychlejší známé algoritmy provádějící triangulaci (tj. rozdělení na trojúhelníky) daného mnohoúhelníku v čase $N.\log(N)$, nebo dokonce $N.\log(\log(N))$, kde N je počet

vrcholů mnohoúhelníku, jsou velmi komplikované. Ukážeme si zde proto značně jednodušší kvadratický algoritmus. Odvození algoritmu rozdělíme do několika kroků.

1. Nejprve ukážeme, že do každého mnohoúhelníku s více než třemi vrcholy lze umístit diagonálu, tj. úsečku, která leží celá uvnitř mnohoúhelníku a která spojuje dva vrcholy mnohoúhelníku, jež spolu nesousedí na obvodu. Pro konvexní mnohoúhelník je toto tvrzení triviální, stačí spojit libovolnou dvojici nesousedících vrcholů. Je-li mnohoúhelník nekonvexní, existuje v něm vrchol, u něhož leží vnitřní úhel větší než 180 stupňů. Při pohledu z tohoto vrcholu »dovnitř« mnohoúhelníku proto jistě uvidíme alespoň dvě jeho různé strany. Na předělu těchto stran musí být vrchol mnohoúhelníku, který je z našeho vrcholu také vidět a ke kterému tudíž můžeme vést diagonálu.

2. Pro libovolný mnohoúhelník (s alespoň třemi vrcholy) existuje taková jeho triangulace, v níž alespoň jeden z trojúhelníků, které rozdělením mnohoúhelníku vzniknou, má dvě své strany shodné se stranami mnohoúhelníku. O takovém trojúhelníku budeme dále říkat, že leží »na obvodu« mnohoúhelníku. Ukážeme, jak takovou triangulaci sestrojit. Podle bodu 1 lze daný mnohoúhelník rozdělit diagonálou na dva menší. Každý z nich má právě jednu svoji stranu tvořenou diagonálou a všechny zbývající jeho strany jsou stranami původního mnohoúhelníku. Zvolíme libovolný z menších mnohoúhelníků a budeme pokračovat v jeho dělení. Opět podle 1 je možné rozdělit ho diagonálou. Ze dvou vzniklých menších mnohoúhelníků jeden znovu splňuje podmínku, že jediná jeho strana je diagonálou původního mnohoúhelníku a zbývající jeho strany jsou stranami původního mnoho-

úhelníku. S tímto mnohoúhelníkem budeme pokračovat v dělení stejným způsobem. Při každém dělení se zmenšuje počet vrcholů mnohoúhelníku, s nímž právě pracujeme, takže po konečně mnoha krocích získáme trojúhelník. Ten má opět vlastnost, že pouze jedna jeho strana je diagonálou výchozího mnohoúhelníku. Jedná se tudíž o trojúhelník »na obvodu« mnohoúhelníku. Triangulaci zbývajících dílčích mnohoúhelníků, kterými jsme se dosud nezabývali, již provedeme libovolně. Existence nějaké triangulace každého mnohoúhelníku přímo plyne z 1.

3. Na základě tvrzení 2 o existenci trojúhelníku »na obvodu« mnohoúhelníku již můžeme sestrojit algoritmus řešení úlohy. Vyjdeme od prvního vrcholu mnohoúhelníku $A[1]$ a budeme postupovat po obvodu ve zvoleném směru proti pohybu hodinových ručiček (v souladu se způsobem zadání mnohoúhelníku). Přitom budeme hledat v pořadí první vrchol mnohoúhelníku $A[I]$ takový, aby trojúhelník tvořený vrcholy $A[I-1]$, $A[I]$, $A[I+1]$ byl trojúhelníkem »na obvodu« mnohoúhelníku. To bude splněno právě tehdy, jestliže vnitřní úhel mnohoúhelníku u vrcholu $A[I]$ bude konvexní (tj. menší než 180 stupňů) a jestliže do tohoto trojúhelníku nezasahuje žádný jiný vrchol mnohoúhelníku. Tvrzení 2 nám zaručuje, že takový trojúhelník existuje, a při systematickém zkoumání všech vrcholů ho tedy jistě najdeme. Po nalezení trojúhelníku »na obvodu« $A[I-1]\,A[I]\,A[I+1]$ tento trojúhelník od mnohoúhelníku oddělíme diagonálou spojující vrcholy $A[I-1]$ a $A[I+1]$ (stává se součástí hledané triangulace). Zbylý mnohoúhelník má o jeden vrchol méně — o vrchol označený $A[I]$. V triangulaci budeme pokračovat stejným způsobem počínaje od vrcholu $A[I-1]$ (ten musíme znovu

prozkoumat, neboť vnitřní úhel u něj ležící se oddělením trojúhelníku $A[I-1]A[I]A[I+1]$ zmenšil) ve stejném směru proti pohybu hodinových ručiček. Výpočet skončí v okamžiku, kdy nám zbude poslední trojúhelník.

4. Správnost uvedeného algoritmu plyne z dokázaných tvrzení a byla zdůvodněna zároveň s odvozováním algoritmu. Výpočet skončí po konečně mnoha krocích, neboť v každém kroku je od mnohoúhelníku oddělen jeden trojúhelník (je nalezena jedna diagonála náležející do výsledné triangulace), takže počet kroků výpočtu je předem znám. Do mnohoúhelníku o N vrcholech se při triangulaci umístí přesně $N-3$ diagonál.

5. Algoritmus má kvadratickou časovou složitost. Otestování jednoho vrcholu mnohoúhelníku, zda u něj leží trojúhelník »na obvodu«, vyžaduje řádově N operací (testy na náležení do trojúhelníku pro všechny zbývající vrcholy). Přitom celá triangulace se provede při jednom »obejití« po obvodu mnohoúhelníku, tzn. při provedení řádově N takových testování. Poslední tvrzení plyne ze zvoleného pořadí, v jakém vyšetřujeme jednotlivé vrcholy. Při zkoumání vrcholu $A[I]$ totiž víme, že u žádného z vrcholů $A[1], \ldots, A[I-1]$ neleží trojúhelník »na obvodu«. Jestliže nyní oddělíme trojúhelník $A[I-1]\,A[I]\,A[I+1]$, mohla se změna dotknout pouze vrcholu $A[I-1]$, u kterého se zmenšil vnitřní úhel. Od tohoto vrcholu se proto pokračuje v prohledávání.

6. Na závěr uvedeme programovou realizaci algoritmu. Na vstupu programu je očekáván nejprve počet vrcholů mnohoúhelníku a dále souřadnice všech vrcholů. Výstupem je seznam diagonál tvořících triangulaci.

```
program TRIANGULACE (input, output);
const MAX = 100;                    {maximální počet vrcholů}

var AX, AY: array [0..MAX] of real;
                                    {souřadnice vrcholů}
    N: integer;                     {skutečný počet vrcholů}
    NALEZEN: Boolean;               {příznak nalezení trojúhelníku}
    MOZNOST: Boolean;               {příznak možného trojúhelníku}
    MEZ: integer;                   {počet všech diagonál}
    I, J, K: integer;               {pomocné proměnné}

begin
read (N);
for I: = 1 to N do
  read (AX[I], AY[I]);              {přečteny souřadnice vrcholů}
AX[0]: = AX[N];                     {kopie prvního a posledního ..}
AY[0]: = AY[N];                     {... vrcholu mnohoúhelníku ...}
AX[N + 1] : = AX[1];                {... z technických důvodů — ..}
AY[N + 1] : = AY[1];                {... — pro snazší testování}

I: = 1;                             {začneme od vrcholu A[1]}
MEZ: = N — 3;                       {počet všech diagonál}
for J: = 1 to MEZ do
  begin                             {sestrojíme další diagonálu}

  NALEZEN: = false;
  while not NALEZEN do  {hledáme trojúh. »na obvodu«}
    begin
    if UHEL (AX[I + 1], AY[I + 1], AX[I], AY[I],
                    AX[I — 1], AY[I — 1]) < 180 then
```

```
      begin                {konvexní vnitřní úhel}
      K: = 1;
      MOZNOST: = true;
      while MOZNOST and (K < = N) do
        begin
        if abs (K - I) > 1 then {vrchol A[K] je různý
                    od A[I] a nesousedí s ním na obvodu}
            if VNITR (AX[K], AY[K], AX[I - 1],
              AY[I - 1], AX[I], AY[I], AX[I + 1],
              AY[I + 1]) then
                MOZNOST: = false;     {A[K] leží uvnitř}
        K: = K + 1
        end;
      if MOZNOST then NALEZEN: = true
                          {žádný jiný vrchol není uvnitř
                      trojúhelníku A[I - 1] A[I] A[I + 1]}
      else I: = I + 1                 {zkusíme další vrchol}
      end
    else I: = I + 1                   {zkusíme další vrchol}
    end;
  writeln ('Diagonála:', AX[I - 1], AY[I - 1], AX[I + 1],
            AY[I + 1]);       {diag. nalezena a vytištěna}
  for K: = I to N do
    begin
    AX[K]: = AX[K + 1]; {vynechání vrcholu A[I] ...}
    AY[K]: = AY[K + 1] {... z mnohoúhelníku}
    end;
  if I = 1 then
    begin                     {kopie nového vrcholu A[1]}
    AX[N]: = AX[1];
```

```
    AY[N]: = AY[1]
    end;
  if I = N then                {kopie nového vrcholu A[N]}
    begin
    AX[0]: = AX[N − 1];
    AY[0]: = AY[N − 1]
    end;
  N: = N − 1;                  {nový počet vrcholů}
  I: = I − 1;                  {nový zkoumaný vrchol}
  if I = 0 then I: = N
    end
end.
```

b) Lomená čára spojující po řadě body $P(1), \ldots, P(N), P(1)$ je zřejmě uzavřená. Podle definice tedy tvoří mnohoúhelník právě tehdy, jestliže sama sebe nikde neprotíná. Stačí proto ověřit, zda existuje nějaká dvojice stran mnohoúhelníku, které spolu na obvodu nesousedí a které přitom mají společný bod. Lomená čára $P(1), \ldots, P(N), P(1)$ tvoří mnohoúhelník právě tehdy, jestliže taková dvojice úseček neexistuje.

Při řešení úlohy budeme vyšetřovat vzájemnou polohu každé dvojice různých spolu nesousedících úseček ze zadané lomené čáry. Správnost tohoto řešení je zřejmá. Algoritmus je jistě konečný, neboť všech úseček je konečně mnoho a každou dvojici úseček budeme vyšetřovat pouze jednou. Z toho zároveň plyne, že algoritmus bude mít kvadratickou časovou složitost.

Zbývá ještě vyřešit otázku, jak určit, zda dvě úsečky zadané souřadnicemi svých koncových bodů mají nějaký společný bod. Existuje několik různých postupů, jak vyhledat společný

bod úseček. Jednou možností je nepoužít vůbec předdefinované funkce a celý problém vyřešit prostředky analytické geometrie pomocí parametrických rovnic obou úseček. Ukážeme si zde jiné řešení, využívající funkci VPRAVO. Označme zkoumané úsečky *AB*, *CD* a souřadnice jejich krajních bodů *XA*, *YA*, ... atd. Jestliže některý z krajních bodů jedné úsečky leží na druhé úsečce, pak úsečky jistě mají společný bod. V opačném případě stačí vyšetřit, zda oba body *A*, *B* leží ve stejné polorovině určené přímkou *CD* a zda oba body *C*, *D* leží ve stejné polorovině určené přímkou *AB*. Úsečky *AB*, *CD* mají společný bod právě tehdy, pokud ani jedna z těchto podmínek neplatí. K uvedenému vyšetřování polohy bodů použijeme přímo předdefinovanou funkci VPRAVO. Hraniční přímku již můžeme zahrnout do libovolné poloroviny, neboť případ, že některý krajní bod jedné úsečky leží na druhé úsečce, jsme již vyřešili dříve.

Uvedené řešení dílčího problému společného bodu dvou úseček nyní naprogramujeme. Pomocná lokální funkce *NAUSECCE*(*X1*, *Y1*, *X2*, *Y2*, *X3*, *Y3*) dává logickou výstupní hodnotu podle toho, zda na úsečce s krajními body *P1* = (*X1*, *Y1*) a *P2* = (*X2*, *Y2*) leží bod *P3* = (*X3*, *Y3*). Hodnota výsledné logické funkce *PRUSECIK*(*XA*, *YA*, *XB*, *YB*, *XC*, *YC*, *XD*, *YD*) je pak určena tím, zda úsečky *AB*, *CD* mají společný bod.

```
function PRUSECIK(XA, YA, XB, YB, XC, YC, XD,
                  YD : real): Boolean;

function NAUSECCE(X1, Y1, X2, Y2, X3, Y3: real):
                  Boolean;
```

```
  begin
  NAUSECCE: = VPRAVO (X1, Y1, X2, Y2, X3,Y3)and
                VPRAVO (X2, Y2, X1, Y1, X3, Y3) and
                (X3 > = min (X1, X2)) and
                (X3 < = max (X1, X2)) and
                (Y3 > = min (Y1, Y2)) and
                (Y3 < = max (Y1, Y2))
  end; {NAUSECCE}
begin
if NAUSECCE (XA, YA, XB, YB, XC, YC) or
   NAUSECCE (XA, YA, XB, YB, XD, YD) or
   NAUSECCE (XC, YC, XD, YD, XA, YA) or
   NAUSECCE (XC, YC, XD, YD, XB, YB) then
     PRUSECIK: = true
else
  PRUSECIK: = not (
    (VPRAVO (XA, YA, XB, YB XC, YC) =
    VPRAVO (XA, YA, XB, YB, XD, YD))
  or
    (VPRAVO (XC, YC, XD, YD, XA, YA) =
    VPRAVO (XC, YC, XD, YD, XB, YB)))
end; {PRUSECIK}
```

Na závěr uvedeme celý program řešící naši úlohu. Program využívá výše uvedenou funkci *PRUSECIK*. Na vstupu programu je očekávána nejprve hodnota N a za ní postupně souřadnice všech vrcholů zadané lomené čáry. Výstupem je sdělení, zda tato lomená čára tvoří mnohoúhelník.

```
program MNOHOUHELNIK (input, output);
```

```
const MAX = 100;                {max. délka lomené čáry}

var PX, PY: array [1..MAX] of real; {souřadnice bodů}
    N: integer;                 {počet bodů}
    SPOLECNY: Boolean;          {příznak společ. bodu}
    I, J: integer;              {pomocné proměnné}

function PRUSECIK (XA, YA, XB, YB, XC, YC, XD,
      YD: real): Boolean;
  external;

begin
read (N);
for I: = 1 to N do read (PX[I], PY[I]); {přečteny souřad.}
PX[N + 1]: = PX[1];
PY[N + 1]: = PY[1];
SPOLECNY: = false;
I: = 1;
while (not SPOLECNY) and (I < = N - 2) do
  begin                         {testujeme úsečku P(I) P(I + 1)}
  J: = I + 2;
  while (not SPOLECNY) and
        ((J < N) or ((I > 1) and (J = N))) do
    begin                       {druhou úsečkou je P(J) P(J + 1)}
    SPOLECNY: = PRUSECIK(PX[I], PY[I], PX[I + 1],
      PY[I + 1], PX[J], PY[J], PX[J + 1], PY[J + 1]);
    J: = J + 1
    end;
  I: = I + 1
  end;
```

```
write ('Zadaná lomená čára');
if SPOLECNY then write ('ne');
writeln ('tvoří mnohoúhelník.')
end.
```