MATEMATICKÁ OLYMPIÁDA NA STREDNÝCH ŠKOLÁCH

54. ročník, školský rok 2004/2005

Riešenia úloh 3. kola kategórie P

1. súťažný deň

P-III-1

Na vyriešenie tejto úlohy zjavne stačí vedieť priradiť náhrdelníkom nové kódy tak, aby náhrdelníky, ktoré sa líšia len rotáciou alebo preklopením mali rovnaký kód. Potom už stačí len ukladať do stroja tieto nové kódy.

Jedno takéto kódovanie vieme ľahko definovať: Napíšeme si všetky možné kódy daného náhrdelníka a za jeho nový kód vyberieme ten z nich, ktorý je lexikograficky najmenší.

Zostáva teda vymyslieť, ako čo najrýchlejšie pre zadaný reťazec tento kód nájsť. Zjavne stačí vedieť riešiť úlohu bez preklápania, teda vybrať lexikograficky najmenšiu spomedzi všetkých rotácií daného reťazca R. (Tento postup potom použijeme na pôvodný reťazec, na jeho zrkadlový obraz a vyberieme menšie z oboch riešení.)

Nech r je dĺžka zadaného reťazca R. Jeho lexikograficky najmenšiu rotáciu označíme R_{min}.

Existujú štandardné stringologické algoritmy, pomocou ktorých vieme rýchlo nájsť lexikograficky najmenšiu rotáciu daného reťazca – vieme napríklad v čase $O(r)$ zostrojiť pre daný reťazec tzv. sufixový strom, prípadne v čase $O(r.\log r)$ jednoduchším algoritmom tzv. sufixové pole. Tieto dátové štruktúry však presahujú rámec tohto textu.

Ukážeme si iné riešenie, ktoré nájde lexikograficky najmenšiu rotáciu daného reťazca v čase $O(r)$.

Predstavme si, že R napíšeme na kružnicu. Všetky podreťazce budeme teda uvažovať cyklicky – napr. podreťazec, ktorý začína $(r-2)$. písmenom a má dĺžku 4, končí prvým písmenom reťazca R.

Budeme postupne zostrojovať množiny „kandidátov“ S_l (kde l je číslo od 0 do r). Prvkami týchto množín budú úseky reťazca R, určené budú svojim začiatkom a dĺžkou. Ak o nich budeme niekedy hovoriť ako o reťazcoch, myslíme tým reťazce tvorené príslušnými písmenami reťazca R.

Naše množiny S_l budú spĺňať nasledujúce podmienky:

1. Všetky úseky v S_l majú dĺžku najviac l. Úseky dĺžky l voláme *aktívne*, ostatné *neaktívne*. Množinu aktívnych úsekov v S_l označíme S'_l. V programe si aktívne úseky pamätáme oddelené od neaktívnych.

   Úseky v S_l môžu mať dĺžku 0. Presnejšie, úsek dĺžky 0 je na každej pozícii, ktorá nie je obsiahnutá v žiadnom neprázdnom úseku.

2. Každý reťazec v S_l je prefixom hľadaného slova R_{min} (teda je zhodný s jeho začiatkom). Ak niektorý úsek v S_l nie je aktívny, tak po pridaní nasledujúceho písmena v R dostaneme reťazec, ktorý už nie je prefixom R_{min}.

   Úseky v S'_l sú práve všetky úseky dĺžky l, ktoré zodpovedajú prefixom R_{min} dĺžky l.

3. Žiadne dva úseky v S_l sa neprekrývajú, ani na seba bezprostredne nenadväzujú. Budeme si ich pamätať v poradí, v akom v R začínajú.

Na začiatku nech S_0 je množina všetkých úsekov dĺžky 0, všetky budú aktívne. Táto množina zjavne spĺňa požadované vlastnosti.

Ak už máme zostrojenú množinu S_l, nasledujúcu množinu zostrojíme takto:

- Nájdeme najmenšie písmeno c, ktoré nasleduje za niektorým úsekom z S'_l. Všetky úseky z S'_l, za ktorými nasleduje c, o jedno (toto) písmeno predĺžime. Neaktívne úseky nezmeníme. Takto upravenú množinu nazveme X.

- X zjavne spĺňa prvú podmienku, kladenú na S_{l+1}. Takisto druhá podmienka je splnená – reťazce, ktoré doteraz boli aktívne, ale nepredĺžili sme ich, sa zjavne na nasledujúcej pozícii líšia od R_{min}. Ak teda X spĺňa aj tretiu podmienku, položíme $S_{l+1} = X$ a máme zostrojenú ďalšiu množinu.

- Čo ale ak X tretiu podmienku nespĺňa? Úseky v X sa nemôžu prekrývať – totiž úseky v S_l sa neprekrývali ani na seba nenadväzovali. Jediné, čo sa teda mohlo stať, je že niektoré úseky v X na seba nadväzujú.

  Ak na seba postupne (cyklicky) nadväzujú všetky aktívne úseky, je reťazec R periodický s periódou $l + 1$. V tomto prípade začiatkom najmenšej rotácie je každý začiatok aktívneho úseku a môžeme skončiť beh algoritmu.

Uvedomte si, že táto situácia skôr či neskôr nastane. V najhoršom prípade (ak R je aperiodický, a teda má len jednu najmenšiu rotáciu) zostrojíme časom množinu S_n, ktorá bude obsahovať jediný aktívny úsek nadväzujúci sám na seba.

- Zostáva nám teda vyriešiť prípad, ak niektoré aktívne úseky v X na seba nenadväzujú. Neaktívny úsek nemôže nadväzovať na úsek, ktorý nasleduje za ním, lebo sme ho nepredĺžili. Takže v X nám tretiu podmienku kazí niekoľko postupností na seba nadväzujúcich aktívnych úsekov, pričom niektoré tieto postupnosti môžu ešte byť ukončené neaktívnym úsekom. Každú takúto postupnosť úsekov spojíme do jedného úseku. Takto dostaneme novú množinu Y. Nech je dĺžka najdlhšieho z týchto nových úsekov q. Potom tvrdíme, že $Y = S_q$.

  Nami zostrojená množina zjavne spĺňa prvú a tretiu podmienku kladenú na S_q. Pozrime sa podrobnejšie na platnosť druhej podmienky:

  Nech aktívny úsek v X zodpovedá reťazcu s. Potom každý neaktívny úsek je prefixom s (z druhej podmienky pre X), a teda ak je $w = ss \ldots sn$ aktívny reťazec v Y, je ľubovolný iný reťazec $w' = ss' \ldots sn'$ v Y jeho prefixom – pretože n' je prefix s a ak w aj w' obsahujú rovnaký počet úsekov s, n musí byť aspoň tak dlhé ako n' a n' je teda prefix n.

  Prefix R_{min} dĺžky q musí byť w, pretože R nemá podreťazec dĺžky $(l + 1)$ menší než s ani podreťazec dlžky $|n|$ menší než n vďaka druhej podmienke pre X. Akýkoľvek úsek v Y kratší ako q sa na nasledujúcej pozícii líši od w – inak by podľa druhej podmienky pre X mal byť aspoň o túto pozíciu dlhší. Každý úsek R zodpovedajúci w je v Y aktívny, pretože všetky jeho podúseky zodpovedajúce s aj n museli byť v X vďaka druhej podmienke.

  Teda aj druhá podmienka je splnená, čím sme ukázali, že naozaj $Y = S_q$ a opäť sa nám podarilo zostrojiť novú množinu úsekov.

Tento postup opakujeme, až kým niekedy v jeho treťom kroku neskončíme. To sa nutne musí stať, pretože každým opakovaním konštrukcie predĺžime aktívne reťazce aspoň o jeden znak.

Správnosť algoritmu bola ukázaná v popise. Časová zložitosť je $O(r)$. To nahliadneme týmto spôsobom:

- V prvom kroku konštrukcie novej množiny sú dva prípady – písmeno nasledujúce po danom úseku je buď c alebo nie je. V prvom prípade sa toto písmeno stane súčasťou daného úseku a už sa naň

nikdy nepozrieme – to sa môže stať len r krát, raz pre každé písmeno. V druhom prípade úsek prestane byť aktívny – to sa úseku stane nanajvýš raz, nové úseky vznikli len na začiatku konštrukcie a bolo ich r, teda aj toto sa stane nanajvýš r krát počas celého behu programu.

- Operácie v ďalších krokoch robíme len s úsekmi, ktoré sú tou dobou aktívne – to sú ale vždy práve úseky, ktoré sme v predchádzajúcom kroku o písmeno predĺžili.Všetky tieto kroky vieme urobiť v čase lineárnom od počtu momentálne aktívnych úsekov. Preto v každom z týchto krokov urobíme rádovo rovnako práce ako v jemu predchádzajúcom prvom kroku – a teda aj v týchto krokoch vykonáme dokopy najviac lineárne veľa operácií.

Pamäťová zložitosť je tiež $O(r)$, pretože v lineárnom čase nestihneme viac pamäte použiť.

Listing programu:

```
program Collector;

const MaxN = 100;

function ReverseString (var s : string) : string; { obráti reťazec }
var i, n : integer;
    rev : string;
begin
  n := length (s);
  rev := '';
  for i := n downto 1 do rev := rev + s[i];
  ReverseString := rev;
end;

function RotateString (var s : string; k : integer) : string; { rotuje }
var i, n : integer;
    rot : string;
begin
  n := length (s);
  rot := '';
  for i := k to n do rot := rot + s[i];
  for i := 1 to k - 1 do rot := rot + s[i];
  RotateString := rot;
end;
```

```
{ prevedie index do intervalu 1 .. total -- neefektívne, lebo stačí takto }
function RotIndex (idx, total : integer) : integer;
begin
  while idx < 1 do idx := idx + total;
  while idx > total do idx := idx - total;
  RotIndex := idx;
end;

var activePositions : array [1..MaxN] of integer; { zoznam akt. pozícií }
    nActivePositions : integer; { počet aktívnych pozícií }
    activeLength : integer; { dĺžka aktívnych reťazcov }
    segmentLength : array [1..MaxN] of integer;
      { dĺžka úseku začínajúceho na danej pozícii }

{ rozšíri aktívne úseky o písmeno }
procedure ExtendByLetter (var s : string);
var i, tActivePositions, position : integer;
    ch, minCh : char;
begin
  minCh := s[RotIndex (activePositions[1] + activeLength, length (s))];
  for i := 2 to nActivePositions do
    begin
      ch := s[RotIndex (activePositions[i] + activeLength, length (s))];
      if ch < minCh then minCh := ch;
    end;

  tActivePositions := 0;
  for i := 1 to nActivePositions do
    begin
      position := activePositions[i];
      if s[RotIndex (position + activeLength, length (s))] = minCh then
        begin
          inc (tActivePositions);
          activePositions[tActivePositions] := position;
          inc (segmentLength[position]);
        end;
    end;
  nActivePositions := tActivePositions;

  inc (activeLength);
end;

{ skontroluje, či dva úseky na seba nadväzujú }
function ConsecutiveSegments (u1, u2, len : integer) : boolean;
  var pos1, pos2, e1 : integer;
```

```
begin
  pos1 := activePositions[u1];
  e1 := RotIndex (pos1 + segmentLength[pos1], len);
  pos2 := activePositions[u2];
  ConsecutiveSegments := e1 = pos2;
end;

{ spojí po sobě následující úseky }
procedure MergeSegments (len : integer);
var i, j, tActivePositions, position, endPosition, tActiveLength: integer;
    pActivePositions : array [1 .. MaxN] of integer;
begin
  tActivePositions := 0;
  tActiveLength := activeLength;

  for i := 1 to nActivePositions do begin
    if ConsecutiveSegments(rotIndex(i-1, nActivePositions), i, len) then
      continue;

    position := activePositions[i];
    j := i;
    while ConsecutiveSegments(j, rotIndex(j+1, nActivePositions), len)
    do begin
      j := rotIndex (j + 1, nActivePositions);
      inc (segmentLength[position], activeLength);
    end;
    endPosition := rotIndex (activePositions[j] + activeLength, len);
    inc (segmentLength[position], segmentLength[endPosition]);

    if segmentLength[position] > tActiveLength then
      tActiveLength := segmentLength[position];

    inc (tActivePositions);
    pActivePositions[tActivePositions] := position;
  end;

  nActivePositions := 0;
  for i := 1 to tActivePositions do
    begin
      position := pActivePositions[i];
      if segmentLength[position] = tActiveLength then
        begin
          inc (nActivePositions);
          activePositions[nActivePositions] := position;
        end;
```

```
    end;

  activeLength := tActiveLength;
end;

{ nájde lexikograficky najmenšiu rotáciu reťazca }
function MinimalRotation (var s : string) : string;
var i, l : integer;
begin
  l := length (s);
  nActivePositions := l;
  activeLength := 0;
  for i := 1 to nActivePositions do
    begin
      activePositions[i] := i;
      segmentLength[i] := 0;
    end;

  while true do
    begin
      ExtendByLetter (s);
      if nActivePositions * activeLength = l then
        break;
      MergeSegments (l);
    end;

  MinimalRotation := RotateString (s, activePositions[1]);
end;

{ nájde lexikograficky najmenší kód reťazca }
function MinimalPosition (var s : string) : string;
var rev, minS, minRev : string;
begin
  rev := ReverseString (s);
  minS := MinimalRotation (s);
  minRev := MinimalRotation (rev);

  if minS < minRev then
    MinimalPosition := minS
  else
    MinimalPosition := minRev;
end;

var n, i : integer;
    code : string;
```

```
begin
  readln (n);
  for i := 1 to n do
    begin
      readln (code);
      code := MinimalPosition (code);

      if MamHo (code) then
        writeln ('Ten uz mas.')
      else
        begin
          writeln ('Kup ho!');
          Pridaj (code);
        end;
    end;
end.
```

P-III-2

Ukážeme si riešenie s časovou zložitosťou $O(NM)$, kde N a M sú rozmery mapy areálu MO. Riešenie si najprv popíšeme všeobecne a až v druhej časti sa zameriame na to, ako dosiahnuť časovú zložitosť $O(NM)$.

V mape si najskôr určíme jednotlivé budovy, ktoré sa v areáli MO nachádzajú. Budovy si očíslujeme a každý štvorček `x` nahradíme číslom budovy, do ktorej patrí. Potom si všimneme budovu číslo 1 a pozrieme sa na dĺžky mostov, ktoré môžu z tejto budovy viesť. Všimnite si, že ak si zvolíme políčko na okraji budovy a smer, tak je dĺžka mostu (i budova, do ktorej by viedol) už jednoznačne určená. Zo všetkých týchto mostov vezmeme najkratší a ten do mapy pridáme. Tým spojíme budovu číslo 1 s budovou číslo i. Teraz sa pozrime na všetky možné mosty, ktoré by sme mohli viesť z budovy číslo 1 alebo z budovy číslo i do ostatných budov a najkratšiu z nich pridajme do mapy.

Všeobecne, keď B je množina budov, ktoré sme už vzájomne prepojili pomocou mostov, pridáme najkratší možný most z niektorej budovy v množine B do niektorej z ostatných budov. Ak je takých mostov viac, pridáme ľubovoľný z nich. Algoritmus skončí, keď sme už všetky budovy vzájomne prepojili, alebo sa žiadna budova nedá mostom k už prepojeným budovám pripojiť (v takom prípade vypíšeme vhodnú správu).

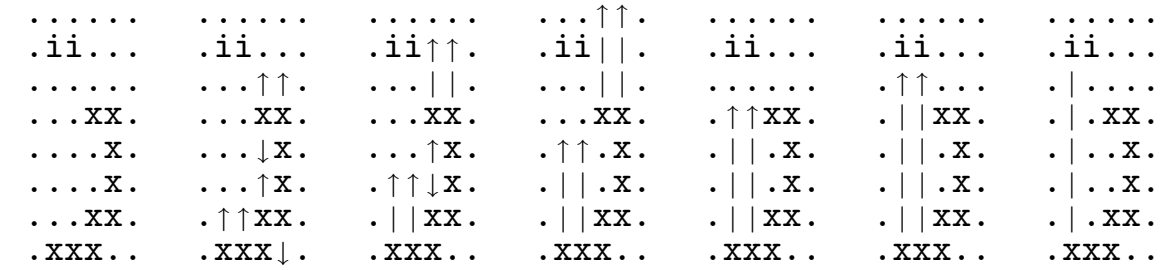
[Pre znalcov dodávame, že to nie je nič iné ako Primov-Jarníkov algoritmus na hľadanie minimálnej kostry grafu.]

Ďalej si rozmyslíme, že ak sa nám podarí prepojiť všetky budovy, tak je nami nájdené riešenie optimálne. Nech $M = \{m_1, \ldots, m_k\}$ je množina mostov, ktoré obsahuje nami nájdené riešenie a predpokladajme, že most m_i bol pridaný do riešenia ako i-ty. Pre spor teraz predpokladajme, že v optimálnom riešení M' je súčet dĺžok mostov je menší než súčet dĺžok mostov z množiny M. Naviac zvoľme za M' také optimálne riešenie, ktoré obsahuje ako podmnožinu čo najväčšiu počiatočnú podpostupnosť $m_1, \ldots, m_l$, t.j. $\{m_1, \ldots, m_l\} \subseteq M'$ a l je maximálne možné. Pretože $M \neq M'$, musí platiť $l < k$.

Nech B je množina budov, ktoré sú vzájomne prepojené mostami $m_1, \ldots, m_l$ a nech j je číslo budovy, do ktorej vedie z množiny B most m_{l+1}. Pretože mosty z množiny M' prepojujú všetky budovy, existuje cesta z množiny budov B do budovy číslo j používajúca mosty z M'. Všimnime si najkratšiu takú cestu $m'_1, \ldots, m'_{l'}$. Z voľby mostu m_{l+1} plynie, že most m'_1 nie je kratší ako most m_{l+1}. Nahraďme teraz v M' most m_{l+1} mostom m'_1. Súčet dĺžok mostov sa týmto krokom nezvýšil. Naviac všetky budovy sú stále prepojené: namiesto mostu m'_1 sa dá použiť cesta tvorená mostami $m_{l+1}, m'_{l'}, \ldots, m'_2$. To je ale spor s voľbou množiny M' ako optimálneho riešenia, ktoré obsahuje čo najväčšiu počiatočnú podpostupnosť $m_1, \ldots, m_l$. Nami nájdené riešenie M je teda optimálne.

Zamerajme sa ešte na to, ako práve popísaný algoritmus implementovať, aby sme dosiahli časovú zložitosť $O(NM)$. Rospoznanie jednotlivých budov ľahko zvládneme v čase $O(NM)$ – mapu postupne prechádzame a v okamžiku, keď narazíme na políčko `x`, prehľadáme (do hĺbky alebo do šírky) v mape oblasť tvorenú touto budovou a všetky políčka `x` nahradíme číslom práve nájdenej budovy. Tento krok zrejme vyžaduje čas $O(NM)$, lebo každé políčko mapy navštívime najviac dvakrát (prvýkrát pri prechode mapou a druhýkrát pri označovaní budovy).

Teraz spustíme druhú fázu algoritmu, v ktorej budeme budovy medzi sebou prepájať mostami. Označme K celkový počet budov. Aby sme mohli rýchlo rozpoznať, ktoré budovy sme už vzájomne prepojili, budeme používať pomocné pole veľkosti K, kde si pre každú budovu uložíme, či je už s budovou číslo 1 spojená, alebo nie je. Z každého okrajového políčka budovy číslo 1 vyšleme súčasne lúče (hore a dolu): v prvom kroku sa lúče nachádzajú na políčkach susediacich priamo s budovou číslo 1 (viď obrázok), v druhom kroku sú vo vzdialenosti 2, atď. Ak lúč narazí na budovu s číslom 1 či opustí mapu, už ho ďalej nepredlžujeme. Takto postupujeme, kým prvý lúč nenarazí na inú budovu. Povedzme, že táto budova má číslo i. Zrejme dráha, ktorú lúč urazil, zodpovedá najkratšiemu možnému mostu z budovy číslo 1 do inej budovy. Tento most pridáme do mapy a budovy najvzájom prepojíme.



Lúče putujúce v mape – zobrazenie po jednotlivých krokoch

Potom vyšleme lúče z budovy číslo i a budeme ich predlžovať, kým nenarazíme na inú budovu, alebo ich dĺžka nebude rovnaká ako dĺžka lúčov vyslaných z budovy číslo 1. V okamžiku, keď dĺžka lúčov vyslaných z budovy číslo i dosiahne dĺžku lúčov vyslaných z budovy číslo 1, začneme predlžovať súčasne lúče vyslané z budovy číslo 1 a zároveň i tie, vyslané z budovy číslo i. Ak však najskôr narazíme na inú budovu, povedzme s číslom i', pripojíme ju mostom k budove číslo i. Z budovy číslo i' vyšleme lúče, kým nenarazíme na inú budovu, alebo ich dĺžka nedosiahne dĺžku lúčov vyslaných z budovy číslo i. V prvom prípade vyšleme lúče z novo-pripojenej budovy, v druhom prípade začneme spoločne predlžovať lúče z budov číslo i a i', kým nedosiahnu dĺžku lúčov vyslaných z budovy číslo 1 (alebo nenarazíme na nepripojenú budovu). Všeobecne, keď narazíme na novú budovu, prerušíme predlžovanie súčasných lúčov a vyšleme lúče z novej budovy. Predlžovanie lúčov obnovíme v okamžiku, keď dĺžka lúčov z novej budovy bude rovnaká ako dĺžka lúčov, ktorých predlžovanie sme prerušili. Všimnite si, že v jednom okamihu môže byť prerušené predlžovanie až K rôznych množín lúčov.

Je zrejmé, že vyššie popísaným postupom pripojíme mostom vždy budovu, ktorú vieme spojiť s už prepojenými budovami najkratším mostom. Pretože každé políčko na mape navštívi každý lúč najviac dvakrát (raz lúč „letiaci“ smerom hore, raz smerom dolu), spotrebuje druhá fáza nášho algoritmu čas iba $O(NM)$. Samotné lúče si budeme udržovať v poli dĺžky $2NM$ zotriedené zostupne podľa ich dĺžky a vždy budeme predlžovať prvý najkratší lúč, ktorý sa v poli nachádza (aby sme udržali lúče utriedené zostupne). Aby sme sa vyhli zbytočnému prechádzaniu tohoto pomocného poľa, budeme si naviac udržovať odkazy na pozíciu lúčov, ktorých predlžovanie sme prerušili. Zoznam lúčov by sme tiež mohli udržovať v spájanom zozname. Pamäťová zložitosť práve popísaného riešenia je, rovnako ako jeho časová zložitosť, $O(NM)$.

Listing programu:

```
program mosty;
const MAX=100;
```

```
var mapa:array[1..MAX,1..MAX] of longint;
    { mapa areálu MO: 0 = močál, -1 = budova, -2 = most
      1, 2, ... = čísla budov }
    delka:longint; { součet délek mostů v řešení }
    N,M:longint; { rozměry mapy }

procedure nacti;
var i, j: longint;
    s: string[MAX];
begin
  readln(M,N);
  for i:=1 to N do
    begin
      readln(s);
      for j:=1 to M do
        if s[j]='.' then mapa[i][j]:=0 else mapa[i][j]:=-1
    end;
end;

procedure urci_na_mape(cislo: longint; x: longint; y: longint);
  { určí na mapě budovu, jež obsahuje souřadnice x a y a
    změní všechny -1 patřící této budově na cislo }
var fronta: array[1..MAX*MAX] of record x,y: longint end;
    hlava, ocas: longint;
begin
  mapa[x][y]:=cislo;
  fronta[1].x:=x;
  fronta[1].y:=y;
  hlava:=0;
  ocas:=1;
  while hlava<ocas do
    begin
      inc(hlava);
      x:=fronta[hlava].x;
      y:=fronta[hlava].y;

      if x>0 then
        if mapa[x-1][y]=-1 then
          begin
            mapa[x-1][y]:=cislo;
            inc(ocas);
            fronta[ocas].x:=x-1;
            fronta[ocas].y:=y;
          end;
```

```
      if y>0 then
        if mapa[x][y-1]=-1 then
          begin
            mapa[x][y-1]:=cislo;
            inc(ocas);
            fronta[ocas].x:=x;
            fronta[ocas].y:=y-1;
          end;

      if x<N then
        if mapa[x+1][y]=-1 then
          begin
            mapa[x+1][y]:=cislo;
            inc(ocas);
            fronta[ocas].x:=x+1;
            fronta[ocas].y:=y;
          end;

      if y<M then
        if mapa[x][y+1]=-1 then
          begin
            mapa[x][y+1]:=cislo;
            inc(ocas);
            fronta[ocas].x:=x;
            fronta[ocas].y:=y+1;
          end;
    end;
end;

procedure pridej_most(x, y, delka, smer: longint);
begin
  while delka>0 do
    begin
      x:=x+smer;
      mapa[x][y]:=-2;
      dec(delka);
    end;
end;

function propoj:boolean;

var paprsky:array[1..2*MAX*MAX] of record
      x, y: longint;     { aktuální pozice paprsku }
      smer: longint;     { směr pohybu paprsku, -1 nahoru, +1 dolů }
      delka: longint;    { délka paprsku }
```

```
  end;
  paprsku:longint;

procedure pridej(x, y: longint);
  var fronta: array[1..MAX*MAX] of record x,y: longint end;
      hlava, ocas: longint;
      cislo: longint;
  begin
    cislo:=mapa[x][y];
    mapa[x][y]:=0;
    fronta[1].x:=x;
    fronta[1].y:=y;
    hlava:=0;
    ocas:=1;
    while hlava<ocas do
      begin
        inc(hlava);

        x:=fronta[hlava].x;
        y:=fronta[hlava].y;

        inc(paprsku);
        paprsky[paprsku].x:=x;
        paprsky[paprsku].y:=y;
        paprsky[paprsku].smer:=+1;
        paprsky[paprsku].delka:=0;

        inc(paprsku);
        paprsky[paprsku].x:=x;
        paprsky[paprsku].y:=y;
        paprsky[paprsku].smer:=-1;
        paprsky[paprsku].delka:=0;

        if x>0 then
          if mapa[x-1][y]=cislo then
            begin
              mapa[x-1][y]:=0;
              inc(ocas);
              fronta[ocas].x:=x-1;
              fronta[ocas].y:=y;
            end;

        if y>0 then
          if mapa[x][y-1]=cislo then
            begin
```

```
            mapa[x][y-1]:=0;
            inc(ocas);
            fronta[ocas].x:=x;
            fronta[ocas].y:=y-1;
          end;

        if x<N then
          if mapa[x+1][y]=cislo then
            begin
              mapa[x+1][y]:=0;
              inc(ocas);
              fronta[ocas].x:=x+1;
              fronta[ocas].y:=y;
            end;

        if y<M then
          if mapa[x][y+1]=cislo then
            begin
              mapa[x][y+1]:=0;
              inc(ocas);
              fronta[ocas].x:=x;
              fronta[ocas].y:=y+1;
            end;
      end;
    while ocas>0 do
      begin
        mapa[fronta[ocas].x][fronta[ocas].y]:=cislo;
        dec(ocas)
      end
  end;

var budov:longint;
    napojeno:array[1..MAX] of boolean;
    i,j:longint;
    pozice:array[0..MAX] of record
      kde: longint; { pozice v poli paprsky, odkud se zpracovává paprsek }
      kam: longint; { pozice v poli, kam se ukládá prodloužený paprsek }
    end;
    pozic:longint;

begin
  { nejdříve nalezneme budovy }
  budov:=0;
  paprsku:=0;
  for i:=1 to N do
```

```
  for j:=1 to M do
    if mapa[i][j]=-1 then
      begin
        inc(budov);
        urci_na_mape(budov,i,j);
        if budov=1 then
          begin
            pridej(i,j);
          end;
      end;

{ nyní si nainicializujeme zbylé datové struktury }
delka:=0;
napojeno[1]:=true;
for i:=2 to budov do napojeno[i]:=false;
pozic:=0;
pozice[0].kde:=1;
pozice[0].kam:=1;
{ sledujeme paprsky a přidáváme mosty }
while paprsku>0 do
  begin
    if pozic=0 then
      begin
        pozice[1].kde:=1;
        pozice[1].kam:=1;
        pozic:=1;
      end;
      if paprsky[pozice[pozic-1].kde].delka >
         paprsky[pozice[pozic].kde].delka+1
      then begin
        pozice[pozic+1].kde:=pozice[pozic].kde;
        pozice[pozic+1].kam:=pozice[pozic].kam;
        pozice[pozic].kde:=pozice[pozic].kam;
        inc(pozic);
      end;
    while pozice[pozic].kde<=paprsku do
      begin
        if ( paprsky[pozice[pozic].kde].x
             + paprsky[pozice[pozic].kde].smer < 1 )
           or
           ( paprsky[pozice[pozic].kde].x
             + paprsky[pozice[pozic].kde].smer > N )
        then begin
          inc(pozice[pozic].kde);
          continue;
```

```
    end;
    paprsky[pozice[pozic].kam].x :=
      paprsky[pozice[pozic].kde].x
      + paprsky[pozice[pozic].kde].smer;
    paprsky[pozice[pozic].kam].y :=
      paprsky[pozice[pozic].kde].y;
    paprsky[pozice[pozic].kam].smer :=
      paprsky[pozice[pozic].kde].smer;
    paprsky[pozice[pozic].kam].delka :=
      paprsky[pozice[pozic].kde].delka+1;
    inc(pozice[pozic].kde);
    if mapa
       [ paprsky[pozice[pozic].kam].x ]
       [ paprsky[pozice[pozic].kam].y ] > 0
    then
      if napojeno[
           mapa[ paprsky[pozice[pozic].kam].x ]
                [ paprsky[pozice[pozic].kam].y ] ]
      then
        continue
      else
        begin
          i:=paprsky[pozice[pozic].kam].x;
          j:=paprsky[pozice[pozic].kam].y;
          dec(paprsky[pozice[pozic].kam].delka);
          delka:=delka+paprsky[pozice[pozic].kam].delka;
          pridej_most( i, j,
            paprsky[ pozice[pozic].kam ].delka,
            - paprsky[ pozice[pozic].kam ].smer );
          if paprsky[pozice[pozic].kam].delka>1 then
            begin
              inc(pozic);
              pozice[pozic].kam:=paprsku+1;
              pozice[pozic].kde:=paprsku+1;
            end;
          napojeno[mapa[i][j]]:=true;
          pridej(i,j);
          break;
        end
    else
      inc(pozice[pozic].kam);
  end;
if pozice[pozic].kde>paprsku then
  begin
    paprsku:=pozice[pozic].kam-1;
```

```
          dec(pozic);
        end;
    end;
  { zkontrolujeme, že jsme propojili všechny budovy }
  propoj:=true;
  for i:=1 to budov do
    if not napojeno[i] then
      propoj:=false
end;

procedure vypis;
var i, j: longint;
begin
  writeln('Celková délka mostů v optimálním řešení je ',delka,'.');
  for i:=1 to N do
    begin
      for j:=1 to M do
        case mapa[i][j] of
          0: write('.');
          -2: write('|');
          else write('x');
        end;
      writeln
    end
end;

begin
  nacti;
  if propoj then
    vypis
  else
    writeln('Všechny budovy nelze propojit mosty.');
end.
```

P-III-3

Najskôr si ukážeme riešenie v konštantnom čase s kvadraticky veľkými registrami. Použijeme pri tom triky z príkladov v zadaní a zo vzorových riešení minulých kôl. Hodiť sa nám bude najmä násobenie, ktorým vieme naraz vytvoriť viacero kópií daného bloku bitov a zvyšok po delení, ktorým naopak dokážeme veľa kópií sčítať do jedného bloku.

Využijeme aj operáciu typu $x \wedge (x-1)$ pre nájdenie najnižšieho jednotkového bitu. Keďže by sme však potrebovali nájsť bit najvyšší, číslo si najskôr obrátime (na to nám konštantný čas a kvadratický priestor stačí – viď riešenie krajského kola); potom nájdeme najnižšiu jednotku (tejto funkcií budeme hovoriť *Low1*) a odčítame od veľkosti vstupu.

Funkciu *Low1* naprogramujeme nasledovne: vypočítame $(x \vee (x-1)) \oplus x$, čím sa nám objavia jednotky práve na mieste núl vpravo od poslednej jednotky a zvyšok čísla bude nulový. Hľadaná hodnota je teda počet týchto jednotiek. Ten zistíme tak, že násobením vhodnou konštantou vytvoríme N kópií čísla, v každej kópii *and*om vynulujeme všetky bity okrem jedného (v nultej kópii nultého, v prvej prvého atď.), číslo, ktoré vznikne pomyselne prerozdelíme na bloky veľkosti o 1 väčšie, čiže všetky nevynulované bity budú najnižšími bitmi bloku, a tie môžeme operáciou modulo $\mathbf{1}^{N+1}$ ľahko sčítať. To už sme tiež raz použili v riešení krajského kola, ale pre osvieženie pamäti si nakreslime, ako to bude vyzerať:

x_3	x_2	x_1	x_0	x_3	x_2	x_1	x_0	x_3	x_2	x_1	x_0	x_3	x_2	x_1	x_0	(N kópií bloku veľkosti N)
x_3	**0**	**0**	**0**	**0**	x_2	**0**	**0**	**0**	**0**	x_1	**0**	**0**	**0**	**0**	x_0	(bity, ktoré chceme sčítať)
x_3	**0**	**0**	**0**	**0**	x_2	**0**	**0**	**0**	**0**	x_1	**0**	**0**	**0**	**0**	x_0	(bloky veľkosti $N+1$)

Program bude veľmi jednoduchý:

	$x = \mathbf{0}^i\mathbf{1}\alpha$
$a := \mathit{Mirror}_N(x)$	$a = \beta\mathbf{1}\mathbf{0}^i$ (zrkadlenie N-bitového čísla)
$y := a \vee (a-1)$	$y = \beta\mathbf{1}\mathbf{1}^i$
$y := y \oplus x$	$y = \mathbf{0}^{N-i}\mathbf{1}^i$
$y := y * (\mathbf{0}^{N-1}\mathbf{1})^N$	$y = (\mathbf{0}^{N-i}\mathbf{1}^i)^N$
$y := y \wedge (\mathbf{1}0^{N-1})(\mathbf{0}\mathbf{1}0^{N-2}) \ldots (\mathbf{0}^{N-2}\mathbf{1}\mathbf{0})(\mathbf{0}^{N-1}\mathbf{1})$	
	$y = (\mathbf{0}^N) \ldots (\mathbf{0}^N)(\mathbf{0}^{N-i}\mathbf{1}\mathbf{0}^{i-1}) \ldots (\mathbf{0}^{N-1}\mathbf{1})$
	$= (\mathbf{0}^{N+1})^{N-i}(\mathbf{0}^N\mathbf{1})^i$
$y := y \,\%\, \mathbf{1}^{N+1}$	$y = i = \mathit{Low1}(a)$
$y := N - 1 - y$	$y = \mathit{Log}(x)$

My sa, samozrejme, s kvadratickou pamäťou neuspokojíme a zredukujeme ju na lineárnu. Urobíme to tak, že si vstup rozdelíme na rádovo $\sqrt{N}$ blokov veľkosti rádovo $\sqrt{N}$, každý blok skomprimujeme do jedného bitu (ktorý bude jednotkový, ak sa vnútri bloku vyskytuje aspoň jedna jednotka) a spočítame logaritmus takéhoto čísla použitím vyššie uvedeného algoritmu (naše číslo má len $\sqrt{N}$ bitov, vzhľadom k čomu máme v N-bitovom registri k dispozícii kvadraticky veľký priestor). Logaritmus nám povie, v ktorom bloku se nachádza najľavejšia jednotka a pre tento

blok spustíme spomínaný algoritmus znovu, čím dopočítáme, kde presne jednotka je.

Ostáva vyriešiť, ako presne kompresiu spraviť a ako sa vyrovnať s tými hodnotami N, ktoré nie sú druhými mocninami. Za veľkosť bloku zvolíme $b = \lceil\sqrt{N}\rceil + 1$ a vstup rozdelíme na $b-1$ blokov (keďže $b \cdot (b-1) \geq (\sqrt{N}+1) \cdot \sqrt{N} > N$, musíme na začiatok pridať ešte pár núl, ktoré nám neovplyvnia výsledok). Použijeme pracovné registre o $b^2 = O(N)$ bitoch.

Ak nastavíme najvyšší bit každého bloku na **1** a od každého bloku odčítame jednotku, zmenia sa najvyššie bity na **0** práve v blokoch, ktoré boli (až na najvyšší bit) celé nulové, inde ostane **1**. Potom pri*or*ujeme pôvodne najvyššie bity, takže pre každý blok teraz máme **1** alebo **0** podľa toho, či obsahoval jednotku alebo nie. Tieto bity už stačí len presunúť k sebe, k čomu použijeme opäť pomyselné prerozdelenie blokov a modulo. Obrázok popisuje situáciu pre 3 bloky o štyroch bitoch:

z_2	**0**	**0**	**0**	z_1	**0**	**0**	**0**	z_0	**0**	**0**	**0**	(pre každý blok jeden bit)
0	**0**	**0**	z_2	**0**	**0**	**0**	z_1	**0**	**0**	**0**	z_0	(posun doprava)
0	**0**	**0**	z_2	**0**	**0**	**0**	z_1	**0**	**0**	**0**	z_0	(prerozdelíme bloky po $b-1$)
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	z_2	z_1	z_0	(modulo $\mathbf{1}^{b-1}$)

Program bude vyzerať takto:

	$x = \beta_{b-2} \ldots \beta_0$ (bloky po b bitoch)
$z := x \vee (\mathbf{10}^{b-1})^{b-1}$	$z = (\mathbf{1}\ldots)\ldots(\mathbf{1}\ldots)$
$z := ((z - (\mathbf{0}^{b-1}\mathbf{1})^{b-1}) \vee x) \wedge (\mathbf{1}^{b-1}\mathbf{0})^{b-1}$	
	$z = (z_{b-2}\mathbf{0}^{b-1})\ldots(z_0\mathbf{0}^{b-1})$ (skomprimované bloky)
$z := z >> b-1$	$z = (\mathbf{0}^{b-1}z_{b-2})\ldots(\mathbf{0}^{b-1}z_0)$
$z := z \,\%\, \mathbf{1}^{b-1}$	$z = z_{b-2}\ldots z_0$
$i := Log(z)$	$i =$ *číslo bloku s najvyššou jednotkou*
$r := (x >> (b*i)) \wedge \mathbf{1}^b$	$r =$ *blok s najvyššou jednotkou*
$j := Log(r)$	$j =$ *pozícia najvyššej jednotky v bloku*
$y := b*i+j$	$y =$ *pozícia jednotky v pôvodnom čísle*

Tento program počíta dvojkový logaritmus stále v konštantnom čase a stačia mu lineárne veľké pracovné registre.

Poznámka: Doplňovanie čísla na dĺžku práve $b \cdot (b-1)$ alebo prevod úlohy na úlohu zrkadlovú vám môžu právom pripadať ako „besné“ triky, ktoré sa nedajú v obmedzenom čase súťaže vymyslieť. Úloha je ale riešiteľná i bez nich, samozrejme za cenu dlhšieho programu a ošetrovania rôznych okrajových prípadov, čomu sme sa chceli v tomto vzorovom riešení vyhnúť.

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY

54. ROČNÍK MATEMATICKEJ OLYMPIÁDY

Riešenia 3. kola kategórie P

1. súťažný deň

Vydala IUVENTA pre vnútornú potrebu Ministerstva školstva SR
Zodpovedný redaktor: M. Forišek
Sadzba programom LaTeX