P-III-1 Náhrdelníky

Pro vyřešení problému zřejmě stačí nalézt kódování náhrdelníků, které je nezávislé na rotacích a obracení. Pak dostaneme-li zadán původní kód, překódujeme si ho do nového kódování a na nový kód se zeptáme stroje. Bude nám tedy stačit pouze jeden dotaz pro každý náhrdelník.

Takové kódování dostaneme například takto: napíšeme si všechny kódy daného náhrdelníku a z nich si vybereme lexikograficky nejmenší. Zbývá tedy vymyslet, jak tento kód co nejrychleji nalézt. Vyřešme nejprve úlohu bez zrcadlení – tj. náhrdelník smíme pouze rotovat. Hledáme tedy pro zadaný řetězec R jeho lexikograficky nejmenší rotaci, označme si ji R_{min}. Délku R si označíme r.

Budeme si postupně konstruovat množiny S_l (pro l mezi 0 a r) podřetězců R zadaných jejich počátečním prvkem a délkou. Pokud se na prvky S_l odkazujeme jakou na úseky, máme na mysli tato jejich umístění v R, pokud se na ně odkazujeme jako na řetězce, máme na mysli hodnotu podřetězce R na dané pozici. Všechny úseky uvažujeme cyklicky podél R, tj. za posledním písmenem R následuje opět jeho první písmeno a naopak. S_l budou splňovat následující podmínky:

- Všechny úseky v S_l mají délku nanejvýš l. Množinu úseků z S_l délky právě l si označme S'_l a nazývejme tyto úseky aktivní. Podmnožinu S'_l si v algoritmu udržujeme zvlášť. Úseky v S_l mohou mít délku 0 – přesněji na každé pozici, která není obsažena v žádném úseku, si můžeme představit úsek délky 0.
- Každý řetězec v S_l je prefixem (počátečním úsekem) R_{min}. Pokud řetězec $s \in S_l$ není aktivní (může být i délky 0) a c je písmeno po něm následující v R, sc není prefix R_{min}. Prvky S'_l jsou právě všechny úseky v R odpovídající prefixu R_{min} délky l.
- Žádné dva úseky v S_l se nepřekrývají ani na sebe bezprostředně nenavazují. Tedy úseky lze seřadit v jejich cyklickém pořadí podél R a v tomto pořadí si je také pamatujeme.

Na začátku položíme S_0 rovnu množině všech úseků délky 0 (tj. bude v ní pro každou pozici v R jeden úsek délky 0) a všechny je prohlásíme za aktivní. Tím zřejmě splníme požadované vlastnosti.

Máme-li S_l, zkonstruujeme následující množinu tímto postupem:

- Najdeme minimální písmeno c následující po úsecích z S'_l. Všechny úseky z S'_l po nichž následuje c o toto písmeno prodloužíme. Takto vytvoříme množinu X.
- X zjevně splňuje první vlastnost pro S_{l+1}. Druhá je také splněna, neboť dříve aktivní řetězce, které jsme neprodloužili, se na následující pozici liší od minimálního rozšíření úseku z S'_l a tedy i od R_{min}, a naopak každý počáteční úsek nějaké minimální rotace R musel být v S'_l a prošel i do X. Pokud X splňuje i třetí podmínku, položíme $S_{l+1} = X$ a postup opakujeme.
- Nechť tedy X tuto podmínku porušuje. Úseky v X se nemohou překrývat, protože úseky z S_l se nepřekrývaly, nenavazovaly na sebe a prodloužili jsme je pouze o jedno písmeno. Mohou však na sebe navazovat. Pokud na sebe všechny aktivní úseky cyklicky navazují, je R periodický s periodou $(l+1)$ a jeho minimální rotace začínají právě na pozicích v X, tedy jsme hotovi a můžeme jako výsledek vrátit libovolnou z nich. Algoritmus vždy skončí tímto způsobem, neboť v nejhorším případě aperiodického řetězce nakonec zkonstruujeme S_n, které obsahuje právě jeden řetězec navazující sám na sebe.
- Zbývá případ, kdy sice X třetí podmínku porušuje, avšak některé aktivní úseky na sebe nenavazují. Neaktivní úsek nemůže navazovat na úsek po něm následující, protože jsme ho neprodloužili. Tedy S_l se rozloží na posloupnosti na sebe navazujících aktivních úseků, případně zakončených jedním neaktivním. Takovou posloupnost úseků spojíme do jednoho. Délka nejdelšího spojeného úseku buď k, pak takto vzniklá množina bude S_k. První a třetí podmínka je pro S_k splněna z konstrukce. Podívejme se podrobněji na platnost druhé podmínky:
  Nechť aktivní úsek v X odpovídá řetězci s. Pak každý neaktivní úsek je prefixem s (z druhé podmínky pro X), a tedy je-li $w = ss\ldots sn$ aktivní řetězec v S_k, je libovolný jiný řetězec $w' = ss\ldots sn'$ jeho prefixem, neboť n' je prefix s a pokud w i w' obsahují stejný počet úseků s, n musí být alespoň tak dlouhé jako n' a n' je tedy prefix n.
  Prefix R_{min} délky k musí být w, neboť R nemá podřetězec délky $(l+1)$ menší než s ani podřetězec délky $|n|$ menší než n díky druhé podmínce pro X. Jakýkoliv řetězec w' v S_k kratší než k se na po něm následující pozici odlišuje od w, neboť jinak by n' muselo být alespoň o tuto jednu pozici delší. Každý úsek R odpovídající w je v S'_k, neboť všechny jeho podúseky odpovídající s i n musely být v X díky druhé podmínce.
  Tedy i druhá podmínka je splněna, což končí naši konstrukci.

Tento postup opakujeme, dokud v jeho třetím kroku neskončíme. To se nutně stane, neboť každým opakováním konstrukce prodloužíme aktivní řetězce alespoň o jeden znak.

Správnost algoritmu byla ukázána v popisu. Časová složitost je $O(r)$. To nahlédneme následovně:

- V prvním kroku konstrukce jsou dva případy – písmeno následující po daném úseku buď je c, nebo není. V prvním případě bude písmeno zahrnuto dovnitř úseku a už se na něj nikdy nepodíváme – to se tedy může stát jen r krát, jednou pro každé písmeno. V druhém případě úsek přestane být aktivní – to se úseku stane

nejvýše jednou, nové úseky vznikly jen na začátku konstrukce a bylo jich r, tedy i toto se stane nejvýše r-krát.

- Složitost ostatních kroků je úměrná počtu aktivních úseků. Nicméně i složitost prvního kroku je úměrná tomuto počtu, a o ní jsme právě ukázali, že je $O(r)$.

Paměťová složitost je také $O(r)$, neboť v lineárním čase nestihneme víc paměti použít.

Zbývá ošetřit zrcadlení. To je ovšem snadné – nalezneme minimální rotaci pro obě zrcadlově symetrické varianty a z nich vezmeme tu menší. Tím zachováme lineární časovou složitost.

```
program Collector;

const MaxN = 100;

function ReverseString (var s : string) : string;                { obrátí řetězec }
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;     { zrotuje řetězec }
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;

{ převede index v cyklickém řetězci do intervalu 1 .. total }
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;    { seznam aktivních pozic }
    nActivePositions : integer;                        { počet aktivních pozic }
    activeLength : integer;                            { délka aktivních řetězců }
    segmentLength : array [1 .. MaxN ] of integer;     { délka úseku začínajícího na dané pozici }

procedure ExtendByLetter (var s : string);             { rozšíří aktivní úseky o písmeno }
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;

{ zkontroluje, zda dva úseky na sebe navazují }
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;

{ nalezne lexikograficky nejmenší rotaci řetězce }
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;

{ nalezne lexikograficky nejmenší kód náhrdelníku }
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 UzMam (code) then
        writeln ('Podvod')
      else
        begin
          writeln ('Kup to');
          Pridej (code);
        end;
    end;
end.
```

P-III-2 Mosty

Předvedeme si řešení s časovou složitostí $O(NM)$, kde N a M jsou rozměry mapy areálu MO. Řešení si nejprve popíšeme obecně a teprve v druhé části se zaměříme na to, jak dosáhnout časové složitosti $O(NM)$.

V mapě si nejdříve určíme jednotlivé budovy, které se v areálu MO nacházejí. Budovy si očíslujeme a každý čtvereček `x` nahradíme číslem budovy, které náleží. Poté uvážíme budovu číslo 1 a podíváme se na délky mostů, které lze z této budovy vést. Povšimněte si, že pokud si zvolíme políčko na okraji budovy a směr, pak je délka mostu (i budova, do které by vedl) již jednoznačně určena. Ze všech těchto mostů vezmeme nejkratší a ten do mapy přidáme. Tím spojíme budovu číslo 1 s budovou číslo i. Nyní se podívejme na všechny možné mosty, které bychom mohli vést z budovy číslo 1 nebo z budovy číslo i do ostatních budov, a nejkratší z nich přidejme do mapy. Obecně, když B je množina budov, které jsme již vzájemně propojili pomocí mostů, přidáme nejkratší možný most z budov v množině B do ostatních budov. Pokud je takových mostů více, přidáme libovolný z nich. Algoritmus skončí, pokud jsme již všechny budovy vzájemně propojili, anebo žádnou budovu nelze mostem k již propojeným budovám připojit (v takovém případě vypíšeme vhodnou zprávu). [Pro znalce dodáváme, že je není nic jiného než Jarníkův algoritmus na hledání minimální kostry grafu.]

Dále si rozmyslíme, že pokud se nám podaří propojit všechny budovy, pak je námi nalezené řešení optimální. Nechť $M = \{m_1, \ldots, m_k\}$ je množina mostů, které obsahuje námi nalezené řešení, a předpokládejme, že most m_i byl přidán jako i-tý most do řešení. Pro spor nyní předpokládejme, že existuje řešení M', jehož součet délek mostů je menší než součet délek mostů z množiny M. Navíc zvolme jako M' optimální řešení, které obsahuje jako podmnožinu co největší počáteční podposloupnost $m_1, \ldots, m_l$, tj. $\{m_1, \ldots, m_l\} \subseteq M'$ a l je maximální možné. Protože $M \neq M'$, musí platit $l < k$.

Nechť B je množina budov, které jsou vzájemně propojeny mosty $m_1, \ldots, m_l$, a nechť j je číslo budovy, do které vede z množiny B most m_{l+1}. Protože mosty z množiny M' propojují všechny budovy, existuje cesta z množiny budov B do budovy číslo j používající mosty z M'. Uvažme nejkratší takovou cestu $m'_1, \ldots, m'_{l'}$. Z volby mostu m_{l+1} plyne, že most m'_1 není kratší než most m_{l+1}. Nahraďme nyní v M' most m_{l+1} mostem m'_1. Součet délek mostů se tímto krokem nezvýšil. Navíc všechny budovy jsou stále propojeny: místo mostu m'_1 lze použít objížďku tvořenou mosty $m_{l+1}, m'_{l'}, \ldots, m'_2$. To je ale spor s volbou množiny M' jako optimálního řešení, které obsahuje co největší počáteční podposloupnost $m_1, \ldots, m_l$. Námi nalezené řešení M je tedy optimální.

Zaměřme se ještě na to, jak právě popsaný algoritmus implementovat, abychom dosáhli časové složitosti $O(NM)$. Rozpoznání jednotlivých budov snadno zvládneme v čase $O(NM)$ - mapu postupně procházíme a v okamžiku, kdy narazíme na políčko `x`, prohledáme (do hloubky nebo do šířky) v mapě oblast tvořenou touto budovou a všechny políčka `x` nahradíme číslem nově nalezené budovy. Tento krok zřejmě vyžaduje čas $O(NM)$, neboť každé políčko mapy navštívíme nejvýše dvakrát (jednou při průchodu mapou a podruhé při označování budovy).

Nyní spustíme druhou fázi algoritmu, ve které budeme budovy mezi sebou propojovat mosty. Označme K celkový počet budov. Abychom mohli rychle rozpoznat, které budovy jsme již vzájemně propojili, budeme používat pomocné pole velikosti K, kde si pro každou budovu uložíme, zda je již s budovou číslo 1 spojena či není. Z každého okrajového políčka budovy číslo 1 vyšleme současně paprsky (nahoru a dolů): v prvním kroku se paprsky nacházejí na políčkách sousedících přímo s budovou číslo 1 (viz obrázek), v druhém kroku jsou ve vzdálenosti 2, atd. Pokud paprsek narazí na budovu s číslem 1 či opustí mapu, již ho nadále neprodlužujeme. Takto postupujeme, dokud první paprsek nenarazí na jinou budovu. Řekněme, že tato budova má číslo i. Zřejmě dráha, kterou paprsek urazil, odpovídá nejkratšímu možnému mostu z budovy číslo 1 do jiné budovy. Tento most přidáme do mapy a budovy vzájemně propojíme.

```
......   ......   ......   ...↑↑.   ......   ......   ......
.ii...   .ii...   .ii↑↑.   .ii||.   .ii...   .ii...   .ii...
......   ...↑↑.   ...||.   ...||.   ......   .↑↑...   .|....
...xx.   ...xx.   ...xx.   ...xx.   .↑↑xx.   .||xx.   .|.xx.
....x.   ...↓x.   ...↑x.   .↑↑.x.   .||.x.   .||.x.   .|..x.
....x.   ...↑x.   .↑↑↓x.   .||.x.   .||.x.   .||.x.   .|..x.
...xx.   .↑↑xx.   .||xx.   .||xx.   .||xx.   .||xx.   .|.xx.
.xxx..   .xxx↓.   .xxx..   .xxx..   .xxx..   .xxx..   .xxx..
```

Paprsky putující v mapě – zobrazení po jednotlivých krocích

Poté vyšleme paprsky z budovy číslo i a budeme je prodlužovat, dokud nenarazíme na jinou budovu nebo jejich délka nebude stejná jako délka paprsků vyslaných z budovy číslo 1. V okamžiku, kdy délka paprsků vyslaných z budovy číslo i dosáhne délky paprsků vyslaných z budovy číslo 1, začneme prodlužovat současně jak paprsky vyslané z budovy číslo 1, tak i ty vyslané z budovy číslo i. Pokud však dříve narazíme na jinou budovu, řekněme s číslem i', připojíme ji mostem k budově číslo i. Z budovy číslo i' vyšleme paprsky, dokud nenarazíme na jinou budovu nebo jejich délka nedosáhne délky paprsků vyslaných z budovy číslo i. V prvním případě vyšleme paprsky z nově připojené budovy, v druhém případě začneme společně prodlužovat paprsky z budov číslo i a i', dokud nedosáhnou délky paprsků vyslaných z budovy číslo 1 (nebo nenarazíme na nepřipojenou budovu). Obecně, když narazíme na novou budovu, přerušíme prodlužování současných paprsků a vyšleme paprsky z nové budovy. Prodlužování paprsků obnovíme v okamžiku, kdy délka paprsků z nové budovy bude stejná jako délka paprsků, jejichž prodlužování jsme přerušili. Všimněte si, že v jednom okamžiku může být přerušeno prodlužování až K různých množin paprsků.

Je zřejmé, že výše popsaným postupem, připojujeme mostem vždy budovu, kterou lze spojit s již propojenými budovami nejkratším mostem. Protože každé políčko na mapě navštíví každý paprsek nejvýše dvakrát (jednou paprsek „letící“ směrem nahoru a jednou směrem dolů), spotřebuje druhá fáze našeho algoritmu čas pouze $O(NM)$. Samotné paprsky si budeme udržovat v poli délky $2NM$ setříděné dle jejich délky sestupně a vždy budeme prodlužovat první nejkratší paprsek, který se v poli nachází (abychom udrželi paprsky setříděné sestupně). Abychom se vyhnuli zbytečnému procházení tohoto pomocného

pole, budeme si navíc udržovat odkazy na pozici paprsků, jejichž prodlužování jsme přerušili. Seznam paprsků bychom též mohli udržovat ve spojovém seznamu. Paměťová složitost právě popsaného řešení je, stejně jako jeho časová složitost, $O(NM)$.

```
program mosty;
const MAX=100;
var mapa:array[1..MAX,1..MAX] of longint;   { mapa areálu MO:
                                              0 = volné prostranství, -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 paprsky, 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 ALÍK

Nejprve předvedeme řešení v konstantním čase s kvadraticky velkými registry. Jeho ingrediencemi budou triky použité v příkladech v zadání a ve vzorových řešeních minulých kol. Hodit se nám bude zejména násobení pro vytvoření mnoha kopií daného bloku bitů současně, zbytek po dělení, který naopak dokáže mnoho kopií posčítat do jednoho bloku, a operace typu $x \wedge (x-1)$ pro nalezení nejnižšího jedničkového bitu.

My bychom ovšem potřebovali najít bit nejvyšší, a tak si číslo nejprve obrátíme (na to nám konstantní čas a kvadratický prostor stačí, viz řešení krajského kola), pak nalezneme nejnižší jedničku (této funkci budeme říkat *Low1*) a odečteme od velikosti vstupu.

Funkci *Low1* naprogramujeme následovně: vypočteme $(x \vee (x-1)) \oplus x$, čímž se nám objeví jedničky právě na místě nul vpravo od poslední jedničky a zbytek čísla bude nulový. Hledaná hodnota je tedy počet těchto jedniček. Ten zjistíme tak, že násobením vhodnou konstantou vytvoříme N kopií čísla, v každé kopii *and*em vynulujeme všechny bity kromě jednoho (v nulté kopii nultého, v první prvního atd.), vzniklé číslo pomyslně přerozdělíme na bloky velikosti o 1 větší, čili všechny nevynulované bity budou nejnižšími bity bloku, a ty můžeme operací modulo $\mathbf{1}^{N+1}$ snadno sečíst. To jsme už také jednou použili v řešení krajského kola, ale pro osvěžení si nakresleme, jak to bude vypadat:

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 kopií bloku velikosti N)
x_3	**0**	**0**	**0**	**0**	x_2	**0**	**0**	**0**	**0**	x_1	**0**	**0**	**0**	**0**	x_0	(bity, které chceme sečíst)
x_3	**0**	**0**	**0**	**0**	x_2	**0**	**0**	**0**	**0**	x_1	**0**	**0**	**0**	**0**	x_0	(bloky velikosti $N+1$)

Program bude velice jednoduchý:

	$x = \mathbf{0}^i\mathbf{1}\alpha$
$a := \mathit{Mirror}_N(x)$	$a = \beta\mathbf{10}^i$ (zrcadlení N-bitového čísla)
$y := a \vee (a-1)$	$y = \beta\mathbf{11}^i$
$y := y \oplus x$	$y = \mathbf{0}^{N-i}\mathbf{1}^i$
$y := y * (\mathbf{0}^{N-1}\mathbf{1})^N$	$y = (\mathbf{0}^{N-i}\mathbf{1}^i)^N$
$y := y \wedge (\mathbf{10}^{N-1})(\mathbf{010}^{N-2})\ldots(\mathbf{0}^{N-2}\mathbf{10})(\mathbf{0}^{N-1}\mathbf{1})$	$y = (\mathbf{0}^N)\ldots(\mathbf{0}^N)(\mathbf{0}^{N-i}\mathbf{10}^{i-1})\ldots(\mathbf{0}^{N-1}\mathbf{1}) = (\mathbf{0}^{N+1})^{N-i}(\mathbf{0}^N\mathbf{1})^i$
$y := y \,\%\, \mathbf{1}^{N+1}$	$y = i = \mathit{Low1}(a)$
$y := N-1-y$	$y = \mathit{Log}(x)$

My se ovšem s kvadratickou pamětí nespokojíme a zredukujeme ji na lineární. Uděláme to tak, že si vstup rozdělíme na řádově $\sqrt{N}$ bloků velikosti řádově $\sqrt{N}$, každý blok zkomprimujeme do jednoho bitu (který bude jedničkový, pokud se uvnitř bloku vyskytuje alespoň jedna jednička) a spočítáme logaritmus vzniklého čísla použitím předchozího algoritmu (naše číslo má jen $\sqrt{N}$ bitů, vzhledem k čemuž máme v N-bitovém registru k dispozici kvadraticky velký prostor). Logaritmus nám řekne, ve kterém bloku se nachází nejlevější jednička a pro tento blok spustíme předchozí algoritmus znovu, čímž dopočítáme, kde přesně jednička je.

Zbývá vyřešit, jak přesně kompresi provést a jak se vyrovnat s těmi N, která nejsou druhými mocninami. Za velikost bloku zvolíme $b = \lceil\sqrt{N}\rceil + 1$ a vstup rozdělíme na $b-1$ bloků (jelikož $b \cdot (b-1) \geq (\sqrt{N}+1) \cdot \sqrt{N} > N$, musíme ještě přidat pár nul na začátek, které nám neovlivní výsledek). Použijeme pracovní registry o $b^2 = O(N)$ bitech.

Pokud nastavíme nejvyšší bit každého bloku na **1** a od každého bloku odečteme jedničku, změní se nejvyšší bity na **0** právě v blocích, které byly (až na nejvyšší bit) celé nulové, jinde **1** zůstane. Pak při*or*ujeme původní nejvyšší bity, takže pro každý blok nyní máme **1** nebo **0** podle toho, zda obsahoval jedničku či nikoliv. Tyto bity už stačí jen přesunout k sobě, k čemuž použijeme opět pomyslné přerozdělení bloků a modulo. Obrázek popisuje situaci pro 3 bloky o čtyřech bitech:

z_2	**0**	**0**	**0**	z_1	**0**	**0**	**0**	z_0	**0**	**0**	**0**	(pro 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	(přerozdělíme 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 vypadat takto:

	$x = \beta_{b-2} \dots \beta_0$ ($b-1$ bloků o b bitech)
$z := x \vee (\mathbf{10}^{b-1})^{b-1}$	$z = (\mathbf{1}\dots)\dots(\mathbf{1}\dots)$
$z := ((z - (\mathbf{0}^{b-1}\mathbf{1})^{b-1}) \vee x) \wedge (\mathbf{1}^{b-1}\mathbf{0})^{b-1}$	$z = (z_{b-2}\mathbf{0}^{b-1})\dots(z_0\mathbf{0}^{b-1})$ (zkomprimované bloky)
$z := z >> b-1$	$z = (\mathbf{0}^{b-1}z_{b-2})\dots(\mathbf{0}^{b-1}z_0)$
$z := z \,\%\, \mathbf{1}^{b-1}$	$z = z_{b-2}\dots z_0$
$i := \mathit{Log}(z)$	$i =$ *číslo bloku s nejvyšší jedničkou*
$r := (x >> (b*i)) \wedge \mathbf{1}^b$	$r =$ *blok s nejvyšší jedničkou*
$j := \mathit{Log}(r)$	$j =$ *pozice nejvyšší jedničky uvnitř bloku*
$y := b*i + j$	$y =$ *pozice jedničky uvnitř původního čísla*

Tento program počítá dvojkový logaritmus stále v konstantním čase a stačí mu lineárně velké pracovní registry.

Poznámka: Doplňování čísla na délku zrovna $b \cdot (b-1)$ nebo převod úlohy na úlohu zrcadlovou vám mohou právem připadat jako zběsilé triky, které takřka nelze v omezeném čase soutěže vymyslet. Úloha je ale docela snadno řešitelná i bez nich, ovšem za cenu delšího programu a ošetřování různých okrajových případů, čemuž jsme se chtěli vyhnout.

P-III-4 Zaklínadla

Jednotlivým písmenům v zaklínadle přiřadíme čísla zleva doprava podle následujícího magického receptu:

1. Písmeno α, dostane číslo 1, pokud se před ním v zaklínadle nevyskytuje jiné písmeno α, ani písmena α^- a α^+, kde $\alpha^\pm$ je písmeno v abecedě těsně před / za písmenem α. Například písmeno c dostane číslo 1, pokud se před ním v zaklínadle nevyskytuje žádné z písmen b, c, d.
2. V opačném případě písmenu α přiřadíme maximum z čísel přiřazených písmenům α^-, α a α^+, která se před ním vyskytují v zaklínadle, zvětšené o 1. Například písmeno c dostane číslo 4, pokud se před ním vyskytuje písmeno b s číslem 3 a žádné z písmen c a d.

V našem algoritmu bude klíčové následující pozorování:

Tvrzení: Dvě zaklínadla jsou ekvivalentní právě tehdy, když čísla přiřazená každému z písmen $a \dots z$ tvoří stejnou posloupnost.

Než si toto tvrzení dokážeme, demonstrujme si jeho platnost na zaklínadlech ze zadání úlohy:

```
121314   111234
abraka   krabaa
```

Posloupnosti čísel přiřazených písmenům a, b, k a r jsou následující: 134, 2, 1 a 1. Podle tvrzení, které jsme zatím nedokázali, jsou proto obě zaklínadla ekvivalentní.

```
11213   12131
dabra   badar
```

Čísla přiřazená písmenu a v prvním zaklínadle tvoří posloupnost 13, zatímco v druhém tvoří posloupnost 23. Zaklínadla tedy dle našeho tvrzení nejsou ekvivalentní.

Nyní si výše uvedené tvrzení dokážeme:

Důkaz: Pokud v zaklínadle vyměníme dvě sousední písmena, která nejsou v abecedě těsně vedle sebe, čísla přiřazená těmto písmenům se nezmění. Tedy v ekvivalentních zaklínadlech, musí být posloupnosti čísel přiřazených každému z písmen stejné.

Obrácenou implikaci, tj., že pokud čísla přiřazená každému z písmen $a \dots z$ tvoří stejnou posloupnost, pak jsou obě zaklínadla ekvivalentní, dokážeme indukcí dle délky zaklínadla. Tvrzení zřejmě platí pro jedno- a dvoupísmenná zaklínadla. Předpokládejme nyní, že obě zaklínadla jsou tvořena alespoň třemi písmeny. Zvolme v každém ze dvou zaklínadel písmena α a β, která mají přiřazeno největší číslo, a pokud je takových písmen více, tak ta, která jsou abecedně nejmenší. Protože, posloupnosti čísel přiřazených stejným písmenům jsou shodné v obou zaklínadlech, musí platit $\alpha = \beta$.

Protože α je písmeno v prvním zaklínadle s největším číslem, nenásleduje za ním už žádné z písmen α^-, α a α^+, a lze ho tedy posloupností výměn přemístit na konec zaklínadla. Podobně můžeme na konec druhého zaklínadla přemístit písmeno β. Protože výměny dvou sousedních písmen, která nejsou v abecedě těsně vedle sebe, nemění čísla přiřazená jednotlivým písmenům, jsou posloupnosti čísel přiřazených jednotlivým písmenům v obou nových zaklínadlech shodné. Tato vlastnost zůstane zachována i po odebrání písmen α a β. Na takto získaná kratší zaklínadla, použijeme indukční předpoklad. Protože jsou zkrácená zaklínadla ekvivalentní, jsou ekvivalentní i zaklínadla s písmeny α a β na konci, a tedy i původní zaklínadla.

Právě dokázané tvrzení nám dává návod, jak otestovat, zda jsou dvě zaklínadla ekvivalentní. Nejdříve dle výše uvedených pravidel přiřadíme každému písmenu v zaklínadlech číslo, a pak zkontrolujeme, že posloupnosti čísel přiřazených stejným písmenům se shodují. Abychom se vyhnuli při této kontrole vícenásobnému průchodu zadanými zaklínadly, při přiřazování čísel si rovnou vytvoříme posloupnosti pro jednotlivá písmena. Navíc, abychom se vyhnuli (pomalé) dynamické alokaci paměti, v pomocném poli si budeme uchovávat odkazy na následující stejné písmeno v zadaném zaklínadle. Časová i paměťová složitost takovéhoto algoritmu je pro dvojici N-písmenných zaklínadel $O(N+A)$, kde A je počet písmen, která se mohou v zaklínadlech vyskytovat (v našem případě $A = 26$).

```
program zaklinadla;

const MAX=1000000;
var zaklinadla:array[1..2,1..MAX] of char;   { zadaná zaklínadla }
    delka:longint;                           { délka zaklínadel }
    cisla:array[1..2,1..MAX] of longint;     { čísla přiřazená jednotlivým písmenům v zaklínadlech }
    prvni:array[1..2,'a'..'z'] of longint;   { první výskyt daného písmena v zaklínadle (0=není tam) }
    dalsi:array[1..2,1..MAX] of longint;     { odkaz na další výskyt stejného písmena v zaklínadle }
    vstup,vystup:text;                       { vstupní a výstupní soubory }

procedure nacti;
var i:longint;
    c:char;
begin
```

```
  for i:=1 to delka do
    begin
      read(vstup,zaklinadla[1][i],c,zaklinadla[2][i]);
      readln(vstup);
    end;
end;

procedure spocitej(z: integer);
var pozice:array['a'..'z'] of longint;
    i:longint;
    max:longint;
    c:char;
begin
  for c:='a' to 'z' do
    begin
      prvni[z][c]:=0;
      pozice[c]:=0;
    end;
  for i:=1 to delka do
    begin
      max:=0;
      if pozice[zaklinadla[z][i]]<>0 then
        max:=cisla[z][pozice[zaklinadla[z][i]]];
      if zaklinadla[z][i]<>'a' then
        if pozice[pred(zaklinadla[z][i])]<>0 then
          if cisla[z][pozice[pred(zaklinadla[z][i])]]>max then
            max:=cisla[z][pozice[pred(zaklinadla[z][i])]];
      if zaklinadla[z][i]<>'z' then
        if pozice[succ(zaklinadla[z][i])]<>0 then
          if cisla[z][pozice[succ(zaklinadla[z][i])]]>max then
            max:=cisla[z][pozice[succ(zaklinadla[z][i])]];
      cisla[z][i]:=max+1;
      if pozice[zaklinadla[z][i]]=0 then
        begin
          prvni[z][zaklinadla[z][i]]:=i;
          pozice[zaklinadla[z][i]]:=i;
        end
      else
        begin
          dalsi[z][pozice[zaklinadla[z][i]]]:=i;
          pozice[zaklinadla[z][i]]:=i;
        end;
      dalsi[z][i]:=0;
    end;
end;

function porovnej:boolean;
var c:char;
    i1,i2:longint;
begin
  spocitej(1);
  spocitej(2);
  for c:='a' to 'z' do
    begin
      i1:=prvni[1][c]; i2:=prvni[2][c];
      while (i1<>0) and (i2<>0) do
        begin
          if cisla[1][i1]<>cisla[2][i2] then
            begin
              porovnej:=false; exit
            end;
          i1:=dalsi[1][i1];
          i2:=dalsi[2][i2];
        end;
      if (i1<>0) or (i2<>0) then
```

```
        begin
          porovnej:=false; exit
        end;
    end;
  porovnej:=true;
end;

begin
  assign(vstup,'magie.in');
  assign(vystup,'magie.out');
  reset(vstup);
  rewrite(vystup);
  readln(vstup,delka);
  while delka<>0 do
    begin
      nacti;
      if not porovnej then write(vystup,'ne');
      writeln(vystup,'ekvivalentni');
      readln(vstup,delka);
    end;
  close(vstup);
  close(vystup);
end.
```

P-III-5 Asfaltéři

Úlohu si nejdříve přeformulujeme do řeči teorie grafů: máme dán graf G a chceme zjistit, jak lze jeho hrany rozdělit do takových dvojic, že hrany ve dvojici mají společný vrchol. Úloha zřejmě nemá řešení, pokud je hran lichý počet. V ostatních případech ukážeme, jak nalézt hledané rozdělení hran na dvojice. Budeme prohledávat graf do hloubky a vždy před návratem z každého vrcholu (tedy poté, co byly zpracovány všechny sousední dosud neprojité vrcholy) provedeme následující: Vezmeme všechny dosud nespárované hrany sousedící s aktuálním vrcholem. Pokud je hran lichý počet, vynecháme hranu vedoucí k otci (ta musí být dosud nespárovaná). Nyní máme sudý počet hran a můžeme je tedy libovolně spárovat. Po spárování hran můžeme ukončit zpracování aktuálního vrcholu a vrátit se k otci. Tímto způsobem se nám muselo podařit spárovat všechny hrany, protože u každého vrcholu jsme nechali nespárovanou nejvýše hranu vedoucí k otci, ale ta byla nutně spárována při zpracovávání otce. Ani u posledního zpracovávaného vrcholu problém nastat nemohl (teoreticky by mohla nastat situace, že by nebylo kterou hranu vynechat, pokud by hran u tohoto vrcholu zbyl lichý počet) – hran byl celkem sudý počet, spárováním některých hran jsme mohli vyřídit pouze sudý počet hran, a tak nám u posledního vrcholu musel zbýt sudý počet hran, které snadno spárujeme. Algoritmus má časovou i paměťovou složitost $O(M+N)$, kde N je počet vrcholů grafu G a M počet jeho hran.

```
program asfalter;

const
  INP = 'asfalt.in';
  OUT = 'asfalt.out';
  MAXN = 10000;
  MAXM = 40000;

type
  {Mezi kterými vrcholy hrana vede a s kterou je spárována (-1 pokud dosud není)}
  edge = record
           a, b : Integer;
           pair : Longint;
         end;
  {Počátek hran od daného vrcholu v seznamu hran a stupeň vrcholu}
  vertex = record
             start : Longint;
             deg : Integer;
             visited : Integer;
           end;

var
  et : Array[1..MAXM] of edge;          {Pole se všemi hranami}
  ep : Array[1..2*MAXM] of Longint;     {Čísla hran seřazená podle vrcholů, ke kterým patří}
  v : Array[1..MAXN] of vertex;         {Pole s vrcholy}
  vn, en : Longint;                     {Počet vrcholů a hran}
```

```
{Načti vstup a vytvoř reprezentaci grafu}
procedure read_input;
var
  i : Longint;
  FI, FO : Text;
begin
  Assign(FI, INP);
  Reset(FI);
  Read(FI, vn, en);
  {Otestujeme, jestli je úloha zjevně neřešitelná}
  if en mod 2 = 1 then begin
    Assign(FO, OUT);
    Rewrite(FO);
    WriteLn(FO, 'Cesty nelze vyasfaltovat.');
    Close(FO);
    Close(FI);
    Halt;
  end;

  for i := 1 to vn do begin
    v[i].deg := 0;
    v[i].visited := 0;
  end;
  for i := 1 to en do begin
    Read(FI, et[i].a, et[i].b);
    et[i].pair := -1;
    Inc(v[et[i].a].deg);
    Inc(v[et[i].b].deg);
  end;
  Close(FI);

  {Seřadíme si pole s hranami}
  v[1].start := 1;
  for i := 2 to vn do begin
    v[i].start := v[i-1].start+v[i-1].deg;
    v[i-1].deg := 0;
  end;
  v[vn].deg := 0;
  for i := 1 to en do begin
    ep[v[et[i].a].start+v[et[i].a].deg] := i;
    Inc(v[et[i].a].deg);
    ep[v[et[i].b].start+v[et[i].b].deg] := i;
    Inc(v[et[i].b].deg);
  end;
end;

{Vrátí druhý konec hrany}
function edge_end(v : Longint; edgenum : Longint) : Longint;
begin
  if et[edgenum].a <> v then
    edge_end := et[edgenum].a
  else
    edge_end := et[edgenum].b;
end;

{Vytiskne výsledek}
procedure print_out;
var
  i : Longint;
  O : Text;
  v : Array[1..4] of Integer;
  tmp : Integer;
begin
  Assign(O, OUT);
```

```
  Rewrite(O);
  for i := 1 to en do begin
    if et[i].pair > i then begin
      v[1] := et[i].a;
      v[2] := et[i].b;
      v[3] := et[et[i].pair].a;
      v[4] := et[et[i].pair].b;

      if (v[1] = v[3]) or (v[1] = v[4]) then begin
        tmp := v[1];
        v[1] := v[2];
        v[2] := tmp;
      end;
      if (v[4] = v[1]) or (v[4] = v[2]) then begin
        tmp := v[3];
        v[3] := v[4];
        v[4] := tmp;
      end;
      WriteLn(O, v[1],' ', v[2],' ', v[4]);
    end;
  end;
  Close(O);
end;

{Prohledávání do hloubky na párování hran}
procedure DFS(act, parent : Integer);
var
  pair, i : Longint;
begin
  pair := -1;
  v[act].visited := 1;

  {Zavoláme se na všechny sousední vrcholy, ve kterých jsme dosud nebyli}
  for i := v[act].start to v[act].start+v[act].deg-1 do
    if v[edge_end(act, ep[i])].visited = 0 then
      DFS(edge_end(act, ep[i]), act);

  {Nyní spárujeme zbylé hrany}
  for i := v[act].start to v[act].start+v[act].deg-1 do
    if (et[ep[i]].pair = -1) and (edge_end(act, ep[i]) <> parent) then begin
      if pair = -1 then
        pair := ep[i]
      else begin
        et[ep[i]].pair := pair;
        et[pair].pair := ep[i];
        pair := -1;
      end;
    end;

  if pair <> -1 then begin      {Hrana zbyla?}
    {Spárujeme ji s hranou k rodiči}
    i := v[act].start;
    while edge_end(act, ep[i]) <> parent do
      Inc(i);
    et[ep[i]].pair := pair;
    et[pair].pair := ep[i];
  end;
end;

begin
  read_input;
  DFS(1, 1);
  print_out;
end.
```