P-II-1 Zasypané město

Označme a_1, a_2 a a_3 počty studentů prvního ročníku jednotlivých specializací a b_1, b_2 a b_3 počty studentů druhého ročníku. Nejdříve si rozmyslíme, kdy studenty určitě nelze rozdělit do dvojic podle požadavků zadání. Například, pokud $a_1 + b_1 > N$, tak studenty do dvojic nelze rozdělit – každá dvojice může obsahovat nejvýše jednoho studenta s první specializací a pokud je takových studentů více než N, tak studenty do dvojic rozdělit nelze. Podobně studenty nelze rozdělit do dvojic pokud $a_2 + b_2 > N$ nebo $a_3 + b_3 > N$.

Nyní si ukážeme, že pokud $a_1 + b_1 \leq N$, $a_2 + b_2 \leq N$ a $a_3 + b_3 \leq N$, tak lze studenty do dvojic rozdělit vždy. Bez újmy na obecnosti můžeme předpokládat, že $a_1 \geq a_2 \geq a_3$ a $a_3 \leq \min\{b_1, b_2, b_3\}$. Pokud by druhá nerovnost neplatila, vyměníme navzájem hodnoty proměnných a_i a b_i pro $i = 1, 2, 3$. Všimněme si, že $b_1 \leq N - a_1 = a_2 + a_3$, tj. počet studentů druhého ročníku s první specializací je menší nebo roven počtu studentů prvního ročníku s druhou a třetí specializací.

Pokud $a_2 \leq b_1$, vytvoříme a_2 dvojic složených ze studentů prvního ročníku s druhou specializací a studentů druhého ročníku s první specializací. Zbylých $b_1 - a_2 \leq a_3$ studentů druhého ročníku s první specializací bude ve dvojici se studenty prvního ročníku s třetí specializací. Ostatních $a_3 - (b_1 - a_2)$ studentů prvního ročníku s třetí specializací pak tvoří dvojice se studenty druhého ročníku s druhou specializací (dle našich předpokladů je $a_3 \leq b_2$, a tedy lze tyto dvojice vytvořit). Protože všech a_1 zbylých studentů druhého ročníku má druhou nebo třetí specializaci, mohou tvořit dvojice se studenty prvního ročníku s první specializací.

Uvažme nyní případ, kdy $a_2 > b_1$. Pak studenti prvního ročníku s druhou specializací vytvoří b_1 dvojic se studenty druhého ročníku s první specializací a $a_2 - b_1$ dvojic se studenty druhého ročníku s třetí specializací. Všimněte si, že tolik dvojic lze opravdu vytvořit neboť $b_3 = N - b_1 - b_2 \geq a_2 - b_1$. Studenti prvního ročníku s třetí specializací pak tvoří dvojice se studenty druhého ročníku s druhou specializací (dle našich předpokladů je $a_3 \leq b_2$). Nakonec, podobně jako v předchozím případě, rozřadíme a_1 zbylých studentů druhého ročníku, kteří všichni mají buď druhou nebo třetí specializaci, do dvojic se studenty prvního ročníku s první specializací.

Naše úvahy nás dovedly k následujícímu algoritmu. Nejdříve načteme vstupní soubor a spočítáme čísla a_1, a_2, a_3, b_1, b_2 a b_3. Poté ověříme, že $a_1 + b_1 \leq N$, $a_2 + b_2 \leq N$ a $a_3 + b_3 \leq N$, a pokud některá z těchto nerovností není splněna, vypíšeme do výstupního souboru, že studenty nelze do dvojic rozdělit. Pokud jsou všechny tři nerovnosti splněny, nalezneme výše popsaným způsobem rozdělení studentů do dvojic. Časová i paměťová složitost popsaného algoritmu je kromě načítání vstupního a vytvoření výstupního souboru konstantní, s nimi časová složitost vzroste na lineární.

```
program mesto;
var N,a1,a2,a3,b1,b2,b3:longint;
const s1:string='starovek';
      s2:string='stredovek';
      s3:string='archeologie';
var rocniky_prohozeny:boolean; { pokud a3<b3, "prohodíme" ročníky }
    soubor:text;

procedure nacti;
var soubor:text;
    i:longint;
    rocnik:integer;
    spec:string;
begin
  assign(soubor,'mesto.in');
  reset(soubor);
  readln(soubor,N);
  a1:=0; a2:=0; a3:=0;
  b1:=0; b2:=0; b3:=0;
  for i:=1 to 2*N do
    begin
      readln(soubor,rocnik,spec);
      if (rocnik=1) and (spec=' '+s1) then inc(a1);
      if (rocnik=1) and (spec=' '+s2) then inc(a2);
      if (rocnik=1) and (spec=' '+s3) then inc(a3);
      if (rocnik=2) and (spec=' '+s1) then inc(b1);
      if (rocnik=2) and (spec=' '+s2) then inc(b2);
      if (rocnik=2) and (spec=' '+s3) then inc(b3);
    end;
  rocniky_prohozeny:=false;
end;
```

```
procedure serad;
var x:longint;
    s:string;
begin
  if a1<a2 then
    begin
      x:=a1; a1:=a2; a2:=x;
      s:=s1; s1:=s2; s2:=s;
    end;
  if a1<a3 then
    begin
      x:=a1; a1:=a3; a3:=x;
      s:=s1; s1:=s3; s3:=s;
    end;
  if a2<a3 then
    begin
      x:=a2; a2:=a3; a3:=x;
      s:=s2; s2:=s3; s3:=s;
    end;
  if b3<a3 then
    begin
      x:=a1; a1:=b1; b1:=x;
      x:=a2; a2:=b2; b2:=x;
      x:=a3; a3:=b3; b3:=x;
      rocniky_prohozeny:=true;
      serad;
    end;
end;

procedure vypis(pocet: longint; spec1,spec2: integer);
  { vypíše "pocet" dvojic studentů se specializacemi spec1 a spec2 }
var x:integer;
begin
  if rocniky_prohozeny then
    begin
      x:=spec1; spec1:=spec2; spec2:=x
    end;
  for x:=1 to pocet do
    begin
      case spec1 of
        1: write(soubor,s1,' ');
        2: write(soubor,s2,' ');
        3: write(soubor,s3,' ');
        end;
      case spec2 of
        1: writeln(soubor,s1);
        2: writeln(soubor,s2);
        3: writeln(soubor,s3);
        end;
    end;
end;

begin
  nacti;
  serad;
  assign(soubor,'mesto.out');
  rewrite(soubor);
  if (a1+b1>N) or (a2+b2>N) or (a3+b3>N) then
    begin
      writeln(soubor,'Studenty nelze rozdelit do dvojic.');
    end
  else
    if (a2<=b1) then
      begin
```

```
        vypis(a2,2,1);
        vypis(b1-a2,3,1);
        vypis(a3-(b1-a2),3,2);
        vypis(a1-b3,1,2);
        vypis(b3,1,3);
      end
    else
      begin
        vypis(b1,2,1);
        vypis(a2-b1,2,3);
        vypis(a3,3,2);
        vypis(b2-a3,1,2);
        vypis(b3-(a2-b1),1,3);
      end;
  close(soubor);
end.
```

P-II-2 Okružní jízda

Nejdříve si zopakujme značení, které jsme používali v řešení obdobné úlohy domácího kola. Mapě Stínové Prahy říkáme *graf*, křižovatkám *vrcholy* a ulicím *hrany*. Hranu, která vede mezi vrcholy u a v, značíme uv. Počet vrcholů grafu je n a počet jeho hran m. *Stupeň* vrcholu je počet hran, které do něj vedou. Podle zadání je stupeň každého vrcholu v našem grafu sudý. Hrany e a f na sebe navazují, pokud obě vedou ze stejného vrcholu. Posloupnost navzájem různých hran $e_1, e_2, \ldots, e_k$ taková, že e_{i+1} navazuje na e_i (pro $1 \leq i < k$), se nazývá *tah*. Jestliže navíc e_1 navazuje na e_k, je tento tah *uzavřený*, a pokud obsahuje všechny hrany, je *eulerovský*. Dvojici hran e a f takové, že f navazuje na e, budeme říkat *přechod*.

Úlohou je nalézt uzavřený eulerovský tah, který navíc neobsahuje žádný ze zakázaných přechodů. Oba algoritmy, které jsme popsali v řešení domácího kola, lze upravit pro nalezení eulerovského tahu v neorientovaném grafu se třemi zakázanými přechody u každého vrcholu. My si zde však popíšeme řešení, které se od obou těchto algoritmů liší. Základní myšlenka tohoto třetího algoritmu je tato: Nejprve si graf rozložíme na několik uzavřených tahů, které nebudou obsahovat zakázané přechody, a pak tyto tahy budeme spojovat, dokud z nich nevytvoříme jeden eulerovský tah.

Protože u každého vrcholu jsou tři zakázané přechody, v grafu se nemůže vyskytovat vrchol stupně dva. Dále se kvůli zjednodušení popisu zbavme vrcholů stupně 4 (v programu tyto úpravy grafu neprovádíme – níže popsaný algoritmus funguje i bez nich, po drobných změnách): Předpokládejme, že v grafu je vrchol stupně čtyři. Jelikož jsou u něj tři zakázané přechody, pak ho buď nelze projít vůbec – to pokud všechny zakázané přechody obsahují stejnou hranu, případně zakážeme všechny přechody mezi trojicí hran – nebo je jednoznačně určeno, které hrany u něj na sebe navazují. V prvním případě můžeme skončit, ve druhém vrchol odebereme a nahradíme ho dvěma hranami. Tyto úpravy zjevně neovlivní, zda v grafu eulerovský tah existuje, nebo ne.

Dalším krokem je rozložení grafu na uzavřené tahy. U každého vrcholu si libovolně zvolíme, které hrany na sebe budou navazovat tak, abychom se vyhnuli zakázaným přechodům. To lze udělat například tak, že si nejprve zvolíme navazující hrany pro ty hrany, které jsou obsaženy v zakázaných přechodech, a pro ostatní hrany si poté navazující hrany vybereme zcela libovolně. Snadno nahlédneme, že takové přiřazení vždy existuje. Toto přiřazení nám určuje nějaké uzavřené tahy (začneme libovolnou hranou e, přejdeme na hranu, která na ni navazuje, atd., až dokud se nevrátíme zpět na e). Pokud takto vytvoříme jen jeden tah, nalezli jsme řešení. Zabývejme se tedy situací, kdy takto vzniklo několik tahů.

Nyní si vybereme libovolný z vytvořených tahů a nazveme ho hlavním. K hlavnímu tahu budeme postupně lepit zbývající tahy. Jestliže neexistuje vrchol, kterým by procházel jak hlavní, tak nějaký jiný tah, pak buď máme řešení, nebo graf není souvislý (a řešení neexistuje). Uvažme tedy vrchol v, kterým prochází hlavní tah, a jeden nebo více jiných tahů.

Pokud u v existují navazující hrany e_1e_2 hlavního tahu a e_3e_4 nějakého jiného tahu tak, že ani e_1e_3 ani e_2e_4 nejsou zakázané, pak tyto tahy spojíme tak, že e_1 bude navazovat na e_3 a e_2 na e_4. Tím zjevně hlavní tah prodloužíme a celkový počet tahů snížíme. Dokažme si nyní, že takové hrany musí existovat:

- Předpokládejme nejprve, že hlavní tah prochází vrcholem v jen jednou, přes hrany e_1 a e_2. Pak u vrcholu v existují ještě alespoň dvě dvojice navazujících hran jiných tahů, e_3e_4 a e_5e_6. Uvažme následující čtyři dvojice přechodů: e_1e_3 a e_2e_4, e_1e_4 a e_2e_3, e_1e_5 a e_2e_6, e_1e_6 a e_2e_5. Protože u v jsou jen tři zakázané přechody, oba přechody v alespoň jedné z těchto dvojic jsou povolené. Tahy pak lze navázat přes tyto povolené přechody.
- Nechť hlavní tah prochází vrcholem v alespoň dvakrát, přes přechody e_1e_2 a e_3e_4. Vrcholem v také musí procházet nějaký další tah, přechodem e_5e_6. Podobně jako v minulém případě nahlédneme, že tahy lze spojit přes alespoň jednu z následujících čtyř dvojic přechodů: e_1e_5 a e_2e_6, e_1e_6 a e_2e_5, e_3e_5 a e_4e_6, e_3e_6 a e_4e_5.

Pokud je graf souvislý, můžeme takto postupovat tak dlouho, dokud spojováním tahů nezískáme jediný tah.

Zbývá určit časovou a paměťovou složitost navrženého řešení úlohy. Budeme si pamatovat, které hrany na sebe v jednotlivých tazích navazují, a které hrany patří do hlavního tahu. Dále si u každého vrcholu budeme udržovat seznam hran, které nepatří do hlavního tahu, a případně jeden nebo dva přechody, které do hlavního tahy patří. S těmito údaji snadno dokážeme prodloužit hlavní tah u zadaného vrcholu v konstantním čase. Dále si budeme pamatovat seznam W vrcholů, které hlavní tah zcela nepokrývá – s jeho pomocí si v konstantním čase dokážeme najít vrchol, u nějž je možné tahy lepit. Pokaždé, když hlavní tah prodloužíme, si projdeme část tahu, kterou jsme k němu přidali (v čase lineárním v délce přidané

části), označíme si, že jeho hrany patří do hlavního tahu, upravíme si seznamy hran u vrcholů, které projdeme, a případně tyto vrcholy přidáme do seznamu W. Vzhledem k tomu, že každou hranu takto projdeme právě jednou, je celková časová (a samozřejmě i paměťová) složitost algoritmu lineární, tj. $O(m+n)$.

```
program eulerovsky_tah_2;

const MAXN = 1000;
type phrana = ^hrana;
     prvek_seznamu = ^prvek_seznamu_r;
     hrana = record
               v : array[1..2] of integer;      {vrcholy mezi nimiž hrana vede}
               t : array[1..2] of phrana;       {navazující hrany u v[1] a v[2]}
               {odkaz na odpovidajici prvek v seznamu hran hlavni ci nehlavni u vrcholu v[1] a v[2]}
               tah_u_vrcholu : array[1..2] of prvek_seznamu;
             end;
     prvek_seznamu_r = record
                              e : pointer;      {hodnota prvku}
                              p, n : prvek_seznamu;
                                                {předchozí a následující prvek}
                            end;
     seznam = record
                hlava : prvek_seznamu_r;        {hlava seznamu}
              end;

type pvrchol = ^vrchol;
     vrchol = record
                cislo : integer;                {číslo vrcholu}
                stupen : integer;               {stupeň vrcholu}
                hlavni : seznam;                {hrany v hlavním tahu}
                nehlavni : seznam;              {hrany mimo hlavní tah}
                zakaz : array[1..3, 1..2] of integer;
                                                {zakázané přechody}
              end;

type graf = record
              n : integer;                      {počet vrcholů}
              m : integer;                      {počet hran}
              delka_tahu : integer;             {počet hran v hlavním tahu}
              v : array[1..MAXN] of vrchol;     {vrcholy}
            end;

{Otestuje, zda je S prázdný}
function prazdny (var s : seznam) : boolean;
begin
  prazdny := s.hlava.n = @s.hlava;
end;

{Nainincializuje prazdny seznam S}
procedure udelej_prazdny (var s : seznam);
begin
  s.hlava.n := @s.hlava;
  s.hlava.p := @s.hlava;
  s.hlava.e := nil;
end;

{Odebere a vrati prvni prvek seznamu S}
function odeber_prvni (var s : seznam) : pointer;
var p : prvek_seznamu;
begin
  p := s.hlava.n;
  s.hlava.n := p^.n;
  p^.n^.p := @s.hlava;
  odeber_prvni := p^.e;
  dispose(p);
end;
```

```
{Přidá prvek E do seznamu S, a vrátí jeho pozici}
function pridej (var s : seznam; e : pointer) : prvek_seznamu;
var p : prvek_seznamu;
begin
  new (p);
  p^.e := e;
  p^.n := s.hlava.n;
  p^.p := @s.hlava;
  s.hlava.n^.p := p;
  s.hlava.n := p;
  pridej := p;
end;

{Vrátí N-tý prvek seznamu S}
function nty (var s : seznam; n : integer) : pointer;
var i : integer;
    p : prvek_seznamu;
begin
  p := @s.hlava;
  for i := 1 to n do
    p := p^.n;
  nty := p^.e;
end;

{Přesune prvek P do seznamu S}
procedure presun (p : prvek_seznamu; var s : seznam);
var pr, nx : prvek_seznamu;
begin
  pr := p^.p;
  nx := p^.n;
  pr^.n := nx;
  nx^.p := pr;

  p^.n := s.hlava.n;
  p^.p := @s.hlava;
  s.hlava.n := p;
  p^.n^.p := p;
end;

{Přidá hranu V1V2 do grafu GR}
procedure pridej_hranu (var gr : graf; v1, v2 : integer);
var h : phrana;
    p : prvek_seznamu;
begin
  new (h);
  h^.v[1] := v1;
  h^.v[2] := v2;
  h^.t[1] := nil;
  h^.t[2] := nil;

  p := pridej (gr.v[v1].nehlavni, h);
  h^.tah_u_vrcholu[1] := p;
  p := pridej (gr.v[v2].nehlavni, h);
  h^.tah_u_vrcholu[2] := p;
  inc (gr.v[v1].stupen);
  inc (gr.v[v2].stupen);
end;

{Načte graf}
procedure nacti_graf (var gr : graf);
var i, v1, v2 : integer;
begin
  readln (gr.n, gr.m);
  gr.delka_tahu := 0;
```

```
  for i := 1 to gr.n do
    with gr.v[i] do
      begin
        cislo := i;
        stupen := 0;
        udelej_prazdny (hlavni);
        udelej_prazdny (nehlavni);
      end;
  for i := 1 to gr.m do
    begin
      readln (v1, v2);
      pridej_hranu (gr, v1, v2);
    end;
  for i := 1 to gr.n do
    with gr.v[i] do
      begin
        read (zakaz[1][1], zakaz[1][2]);
        read (zakaz[2][1], zakaz[2][2]);
        readln (zakaz[3][1], zakaz[3][2]);
      end;
end;

{Najde hranu, ktera u vrcholu V navazuje na E}
function navazujici_u_vrcholu (var v : vrchol; e : phrana) : phrana;
begin
  if e^.v[1] = v.cislo then
    navazujici_u_vrcholu := e^.t[1]
  else
    navazujici_u_vrcholu := e^.t[2];
end;

{Najde v seznamu S u vrcholu V jednu nebo dve dvojice navazujících hran
 E1E2 a E3E4, a vrátí počet nalezených dvojic}
function vyber_navazujici_hrany (var v : vrchol; var s : seznam;
                                 var e1, e2, e3, e4 : phrana) : integer;
begin
  e1 := nty (s, 1);
  e2 := navazujici_u_vrcholu (v, e1);
  if nty (s, 3) = nil then
    begin
      e3 := nil;
      e4 := nil;
      vyber_navazujici_hrany := 1;
    end
  else
    begin
      e3 := nty (s, 2);
      if e3 = e2 then e3 := nty (s, 3);
      e4 := navazujici_u_vrcholu (v, e3);
      vyber_navazujici_hrany := 2;
    end;
end;

{Přesune hranu H na jejím I-tém konci do hlavního tahu}
procedure presun_do_hlavniho (var gr : graf; var lepene : seznam; h : phrana; i : integer);
var v : pvrchol;
    p, np : prvek_seznamu;
begin
  v := @gr.v[h^.v[i]];
  p := h^.tah_u_vrcholu[i];

  if prazdny (v^.hlavni) and (v^.stupen >= 6) then
    np := pridej (lepene, v);
  presun (p, v^.hlavni);
end;
```

```
{Vrátí hranu navazující na AKT různou od POSL}
function navazujici (posl, akt : phrana) : phrana;
begin
  if akt^.t[1] = posl then
    navazujici := akt^.t[2]
  else
    navazujici := akt^.t[1];
end;

{Přidá tah mezi navazujícími hranami ZAC a KON do hlavního tahu, upraví
 odkazy z grafu GR a seznam vrcholu pro lepeni LEPENE}
procedure pridej_k_hlavnimu (var gr : graf; var lepene : seznam;
                             zac, kon : phrana);
var posledni, dalsi : phrana;
begin
  posledni := kon;
  repeat
    dalsi := navazujici (posledni, zac);
    presun_do_hlavniho (gr, lepene, zac, 1);
    presun_do_hlavniho (gr, lepene, zac, 2);
    inc (gr.delka_tahu);
    posledni := zac;
    zac := dalsi;
  until posledni = kon;
end;

{Vrátí true pokud je přechod E1E2 u V zakázaný}
function zakazany (var v : vrchol; e1, e2 : phrana) : boolean;
var u, w, i : integer;
begin
  if e1^.v[1] = v.cislo then u := e1^.v[2] else u := e1^.v[1];
  if e2^.v[1] = v.cislo then w := e2^.v[2] else w := e2^.v[1];
  for i := 1 to 3 do
    with v do
      begin
        if ((zakaz[i][1] = u) and (zakaz[i][2] = w)) or
            ((zakaz[i][2] = u) and (zakaz[i][1] = w)) then
          begin
            zakazany := true;
            exit;
          end;
      end;
  zakazany := false;
end;

{Naváže E1 na E2 ve vrcholu V}
procedure navaz (var v : vrchol; e1, e2 : phrana);
begin
  if e1^.v[1] = v.cislo then e1^.t[1] := e2 else e1^.t[2] := e2;
  if e2^.v[1] = v.cislo then e2^.t[1] := e1 else e2^.t[2] := e1;
end;

{Zkusí slepit tahy přes přechody EH1, EV1 a EH2, EV2 u vrcholu V, vrátí true, pokud se to podaří}
function zkus_slepit (var gr : graf; var lepene : seznam; var v : vrchol;
                      eh1, ev1, eh2, ev2 : phrana) : boolean;
begin
  if zakazany (v, eh1, ev1) or zakazany (v, eh2, ev2) then
    begin
      zkus_slepit := false;
      exit;
    end;

  pridej_k_hlavnimu (gr, lepene, ev1, ev2);
  navaz (v, eh1, ev1);
```

```
  navaz (v, eh2, ev2);
  zkus_slepit := true;
end;

{Přilepí nějaký tah u vrchlu VRCHOL do hlavního tahu}
procedure lep_u_vrcholu (var gr : graf; var lepene : seznam; vrchol : integer);
var v : pvrchol;
    e1, e2, e3, e4, e5, e6, e7, e8 : phrana;
    n1, n2 : integer;
    vys : boolean;
begin
  v := @gr.v[vrchol];

  n1 := vyber_navazujici_hrany (v^, v^.hlavni, e1, e2, e3, e4);
  n2 := vyber_navazujici_hrany (v^, v^.nehlavni, e5, e6, e7, e8);

  if n1 = 2 then
    begin
      if not zkus_slepit (gr, lepene, v^, e1, e5, e2, e6) then
      if not zkus_slepit (gr, lepene, v^, e1, e6, e2, e5) then
      if not zkus_slepit (gr, lepene, v^, e3, e5, e4, e6) then
        vys := zkus_slepit (gr, lepene, v^, e3, e6, e4, e5);
    end
  else
    begin
      if not zkus_slepit (gr, lepene, v^, e1, e5, e2, e6) then
      if not zkus_slepit (gr, lepene, v^, e1, e6, e2, e5) then
      if not zkus_slepit (gr, lepene, v^, e1, e7, e2, e8) then
        vys := zkus_slepit (gr, lepene, v^, e1, e8, e2, e7);
    end;
end;

{Vrátí true, pokud E je spárovaný u vrcholu V}
function sparovano (var v : vrchol; e : phrana) : boolean;
begin
  if e^.v[1] = v.cislo then
    sparovano := e^.t[1] <> nil
  else
    sparovano := e^.t[2] <> nil;
end;

{Vrátí počet zakázaných přechodů z hrany E u vrcholu V}
function pocet_zakazanych (var v : vrchol; e : phrana) : integer;
var soused, i, j, pocet : integer;
begin
  if e^.v[1] = v.cislo then soused := e^.v[2] else soused := e^.v[1];
  pocet := 0;
  for i := 1 to 3 do
    for j := 1 to 2 do
      if v.zakaz[i][j] = soused then inc (pocet);
  pocet_zakazanych := pocet;
end;

{Najde hranu u vrcholu V, na niž je povolený přechod z hrany E, a pokud je to možné, preferuje hrany,
 na které vede nějaký zakázaný přechod}
function povoleny_prechod (var v : vrchol; e : phrana) : phrana;
var p : prvek_seznamu;
    ae, kandidat : phrana;
begin
  kandidat := nil;

  p := v.nehlavni.hlava.n;
  while p^.e <> nil do
    begin
      ae := p^.e;
```

```
      p := p^.n;

      if (ae = e) or zakazany (v, e, ae) or sparovano (v, ae) then
        continue;

      if pocet_zakazanych (v, ae) <> 0 then
        begin
          povoleny_prechod := ae;
          exit;
        end
      else
        kandidat := ae;
    end;

  povoleny_prechod := kandidat;
end;

{Najde hranu u vrcholu V, na niž je zakázaný přechod z hrany E}
function zakazany_prechod (var v : vrchol; e : phrana) : phrana;
var p : prvek_seznamu;
    ae: phrana;
begin
  p := v.nehlavni.hlava.n;
  while p^.e <> nil do
    begin
      ae := p^.e;
      if (ae <> e) and zakazany (v, e, ae) then
        begin
          zakazany_prechod := ae;
          exit;
        end;
      p := p^.n;
    end;
  zakazany_prechod := nil;
end;

{Pokud je u V zakázáno párování, páruje jeho hrany, jinak vrátí false}
function sparuj_zakazane_parovani (var v : vrchol) : boolean;
var e : array[1..6] of phrana;
    pe, z, i : integer;
    znam : boolean;
    ae : phrana;
    p : prvek_seznamu;
begin
  pe := 0;
  p := v.nehlavni.hlava.n;

  while p^.e <> nil do
    begin
      ae := p^.e;
      p := p^.n;

      z := pocet_zakazanych (v, ae);
      if z = 0 then continue;
      if z > 1 then
        begin
          sparuj_zakazane_parovani := false;
          exit;
        end;

      znam := false;
      for i := 1 to pe do
        if e[i] = ae then
          begin
            znam := true;
```

```
            break;
          end;
      if znam then continue;

      inc (pe);
      e[pe] := ae;
      inc (pe);
      e[pe] := zakazany_prechod (v, ae);
    end;

  navaz (v, e[1], e[4]);
  navaz (v, e[3], e[6]);
  navaz (v, e[5], e[2]);

  sparuj_zakazane_parovani := true;
end;

{Pokud je u V zakázán trojúhelník, spáruje jeho hrany, jinak vrátí false}
function sparuj_trojuhelnik (var v : vrchol) : boolean;
var e : array[1..3] of phrana;
    pe, z : integer;
    ae : phrana;
    p : prvek_seznamu;
begin
  pe := 0;
  p := v.nehlavni.hlava.n;

  while p^.e <> nil do
    begin
      ae := p^.e;
      p := p^.n;

      z := pocet_zakazanych (v, ae);
      if z = 0 then continue;
      if z <> 2 then
        begin
          sparuj_trojuhelnik := false;
          exit;
        end;

      inc (pe);
      e[pe] := ae;
    end;

  navaz (v, e[1], povoleny_prechod (v, e[1]));
  navaz (v, e[2], povoleny_prechod (v, e[2]));
  navaz (v, e[3], povoleny_prechod (v, e[3]));

  sparuj_trojuhelnik := true;
end;

{Spáruje hranu u hrany z V, z níž je nejvíc zakázaných přechodů}
procedure sparuj_nejvetsi (var v : vrchol);
var z, zkand : integer;
    ae, kand : phrana;
    p : prvek_seznamu;
begin
  p := v.nehlavni.hlava.n;
  kand := nil;
  zkand := 0;

  while p^.e <> nil do
    begin
      ae := p^.e;
      p := p^.n;
```

```
      z := pocet_zakazanych (v, ae);
      if z <= zkand then continue;

      zkand := z;
      kand := ae;
    end;

  {vybereme si povolený přechod, pokud možno na hranu, z níž vede poslední zbývající zakázaný přechod}
  navaz (v, kand, povoleny_prechod (v, kand));
end;

{Páruje hrany do povolených přechodů u vrcholu V, vrátí true, pokud se to povede}
function sparuj_hrany_vrchol (var v : vrchol) : boolean;
var p : prvek_seznamu;
    e, vynuc : phrana;
    z : integer;
begin
  if v.stupen = 4 then
    begin
      p := v.nehlavni.hlava.n;
      while p^.e <> nil do
        begin
          e := p^.e;
          z := pocet_zakazanych (v, e);
          if (z = 0) or (z = 3) then
            begin
              {pokud není z e zakázán žádný přechod, pak je zakázaný trojúhelník na zbývajících
               třech hranách a párování neexistuje; pokud jsou zakázány všechny přechody,
               pak samozřejmě také neexistuje}
              sparuj_hrany_vrchol := false;
              exit;
            end;
          if z = 2 then
            begin
              {pokud nezabere předchozí podmínka, pak jsou právě dva vynucené přechody tvořící
               dobré párování}
              vynuc := povoleny_prechod (v, e);
              navaz (v, e, vynuc);
            end;
          p := p^.n;
        end;
    end
  else
    begin
      {zakázané přechody pokrývají 6 hran}
      if not sparuj_zakazane_parovani (v) then
      {zakázané přechody tvoří trojúhelník}
      if not sparuj_trojuhelnik (v) then
      {v ostatních případech stačí spárovat hranu, z níž vedou 2 zakázané přechody, s nějakou jinou
       hranou, na kterou vede zakázaný přechod, a ostatní přechody jsou pak všechny povolené}
        sparuj_nejvetsi (v);

      {zbylé hrany spárujeme libovolně}
      vynuc := nil;
      p := v.nehlavni.hlava.n;
      while p^.e <> nil do
        begin
          e := p^.e;
          if not sparovano (v, e) then
            begin
              if vynuc = nil then
                vynuc := e
              else
                begin
```

```
                navaz (v, e, vynuc);
                vynuc := nil;
              end;
          end;
        p := p^.n;
      end;
  end;
end;

{Spáruje hrany do povolených přechodů, vrátí true, pokud se to povede}
function sparuj_hrany (var gr : graf) : boolean;
var i : integer;
begin
  for i := 1 to gr.n do
    if not sparuj_hrany_vrchol (gr.v[i]) then
      begin
        sparuj_hrany := false;
        exit;
      end;
  sparuj_hrany := true;
end;

{Vrátí číslo společného vrcholu navazujících hran E1 a E2}
function spolecny_vrchol (e1, e2 : phrana) : integer;
begin
  if e1^.t[1] = e2 then
    spolecny_vrchol := e1^.v[1]
  else
    spolecny_vrchol := e1^.v[2];
end;

{Vypíše tah}
procedure vypis_tah (var gr : graf);
var ae, ke, pe, se, e1, e2 : phrana;
    i : integer;
begin
  i := vyber_navazujici_hrany (gr.v[1], gr.v[1].hlavni, se, ke, e1, e2);

  pe := ke;
  ae := se;
  write ('1');
  while true do
    begin
      e1 := ae;
      ae := navazujici (pe, ae);
      pe := e1;
      if ae = se then
        break;

      write (' ', spolecny_vrchol (ae, pe));
    end;
end;

var gr : graf;                  {graf}
    i, av : integer;
    lepene : seznam;            {seznam vrcholu k lepení}
    e1, e2, e3, e4: phrana;
    v : pvrchol;
begin
  nacti_graf (gr);
  if not sparuj_hrany (gr) then
    begin
      writeln ('Okruzni jizda neexistuje.');
      exit;
    end;
```

```
  i := vyber_navazujici_hrany (gr.v[1], gr.v[1].nehlavni, e1, e2, e3, e4);
  udelej_prazdny (lepene);
  pridej_k_hlavnimu (gr, lepene, e1, e2);
  av := 1;
  while true do
    begin
      while not prazdny (gr.v[av].nehlavni) do
        lep_u_vrcholu (gr, lepene, av);
      if prazdny (lepene) then
        break;
      v := odeber_prvni (lepene);
      av := v^.cislo;
    end;
  if gr.m = gr.delka_tahu then
    vypis_tah (gr)
  else
    writeln ('Okruzni jizda neexistuje.');
end.
```

P-II-3 Pizza kolem

Úlohu vyřešíme metodou dynamického programování. Nebudeme tuto metodu vysvětlovat v plné obecnosti, ale pouze popíšeme její aplikaci v této konkrétní úloze.

V průběhu algoritmu určíme pro i-tý úsek nejmenší počet pizzerií nutných k pokrytí i-tého až N-tého úseku. Označme tento počet P_i. Kromě hodnot P_i spočítáme i největší index K_i takový, že pomocí P_i pizzerií lze obsloužit i-tý až N-tý úsek a navíc i první až K_i-tý. Označme si ještě součet počtu zákazníků v nejdelší posloupnosti úseků, které začínají i-tým úsekem a jejichž součet počtu zákazníků je nejvýše K, jako S_i. Hodnoty P_i a K_i budeme počítat od posledního úseku kružnice, tj. od $i = N$ k $i = 1$.

Nejdříve určíme všechna i, pro která $P_i = 1$. Zřejmě $P_N = 1$ a K_N je největší číslo takové, že součet počtu zákazníků v N-tém a prvním až i-tém úseku je nejvýše K. S_N je rovno tomuto součtu. Předpokládejme, že už známe hodnoty P_{i+1}, K_{i+1} a S_{i+1} a chceme určit hodnoty P_i, K_i a S_i. Pokud $A_i + S_{i+1} \leq K$, pak $P_i = 1$, $K_i = K_{i+1}$ a $S_i = A_i + S_{i+1}$. V opačném případě, tj. pokud $A_i + S_{i+1} > K$, budeme od S_{i+1} postupně odečítat A_j pro $j = K_{i+1}, K_{i+1} - 1, \ldots$, dokud hodnota součtu $A_i + S_{i+1}$ nebude nejvýše K. V tomto okamžiku známe hodnoty K_i a S_i. Takto postupujeme, dokud $K_i \geq 1$.

Nyní zbývá počítat hodnoty P_i, K_i a S_i, pro něž $P_i > 1$. Nechť j je maximální index takový, že součet počtu zákazníků v $(i+1)$-ním až j-tém úseku je nejvýše K. Hodnotu j a také S_{i+1} budeme znát z předchozího kroku. Postupným snižováním hodnoty j nalezneme největší index takový, že součet počtů zákazníků v i-tém až j-tém úseku je menší nebo roven K. Tento součet je hodnota S_i. Zřejmě $P_i = P_{j+1} + 1$ a $K_i = K_{j+1}$.

Výše uvedeným postupem spočítáme hodnoty P_i a K_i, $i = 1, \ldots, N$, v čase lineárním v N. Uvažme nyní libovolné optimální řešení úlohy a nechť i_0 je nejmenší index takový, že nějaká pizzerie v tomto řešení pokrývá úseky počínaje i_0-tým úsekem. Dle definice hodnot P_i a K_i, je P_i počet pizzerií v optimálním řešení a $K_i \geq i$. K nalezení optimálního řešení nám tedy stačí určit nejmenší P_i pro které $K_i \geq i$. To lze jistě udělat v čase lineárním v N. Pokud si navíc při výpočtu hodnot P_i a K_i budeme pamatovat, kde končí nejdelší posloupnost úseků začínající i-tým úsekem, jejichž součet počtu zákazníků je menší nebo roven K, podaří se nám optimální řešení též sestavit v čase $O(N)$. Můžeme tedy odvodit, že celková časová i paměťová složitost právě popsaného řešení úlohy je lineární v počtu úseků, tedy $O(N)$.

```
program pizza;
const maxN=100000;
var A: array[1..maxN] of longint;
    N,K: longint;
    Pocet: array[1..maxN] of longint;
    Kam: array[1..maxN] of longint;
    Dalsi: array[1..maxN] of longint;
    { Pocet[i] je minimální počet intervalů nutných k pokrytí úseků od i do N;
      tyto intervaly navíc pokryjí všechny úseky od i do Kam[i];
      v optimálním pokrytí začíná po intervalu se začátkem i
      další interval na Dalsi[i] }
    soubor: text;
    i,j: longint;
    soucet: longint;
    optimum: longint;
begin
  assign(soubor,'pizza.in');
  reset(soubor);
  readln(soubor,N,K);
  for i:=1 to N do readln(soubor,A[i]);
```

```
close(soubor);
{ nejdříve inicializujeme konec pole }
{ j bude největší číslo takové, že součet A[1] až A[j] je menší nebo roven K }
j:=1; soucet:=A[1];
while (j<=N) and (soucet+A[j]<=K) do
  begin
    soucet:=soucet+A[j];
    inc(j);
  end;
if j=N+1 then
  begin
    { stačí jedna pizzerie }
    assign(soubor,'pizza.out');
    rewrite(soubor);
    writeln(soubor,1);
    writeln(soubor,1,' ',N);
    close(soubor);
    exit
  end;
{ nyní spočítáme hodnoty na konci pole }
i:=N;
{ i bude index prvku, jehož hodnoty nyní počítáme }
while j>0 do
  begin
    if soucet+A[i]>K then
      begin
        soucet:=soucet-A[j];
        dec(j);
        continue;
      end;
    Pocet[i]:=1;
    Kam[i]:=j;
    Dalsi[i]:=j+1;
    soucet:=soucet+A[i];
    dec(i);
  end;
while soucet+A[i]<=K do
  begin
    Pocet[i]:=1;
    Kam[i]:=0;
    Dalsi[i]:=N+1;
    soucet:=soucet+A[i];
    dec(i);
  end;
{ i je stále index počítaného prvku a j je maximální
  index takový, že součet A[i+1] až A[j] je menší nebo roven K }
j:=N;
while i>0 do
  begin
    if soucet+A[i]>K then
      begin
        soucet:=soucet-A[j];
        dec(j);
        continue;
      end;
    Pocet[i]:=Pocet[j+1]+1;
    Kam[i]:=Kam[j+1];
    Dalsi[i]:=j+1;
    soucet:=soucet+A[i];
    dec(i);
  end;
{ zbývá nalézt optimum }
optimum:=Pocet[1]; j:=1;
for i:=1 to N do
  if (Kam[i]+1>=i) and (optimum>Pocet[i]) then
```

```
      begin
        optimum:=Pocet[i];
        j:=i;
      end;
  assign(soubor,'pizza.out');
  rewrite(soubor);
  writeln(soubor,optimum);
  i:=j;
  while Pocet[j]>1 do
    begin
      writeln(soubor,j,' ',Dalsi[j]-1);
      j:=Dalsi[j];
    end;
  if i=1 then
    writeln(soubor,j,' ',N)
  else
    writeln(soubor,j,' ',i-1);
  close(soubor);
end.
```

P-II-4 Grafomat na lovu

V této úloze jsme se dopustili drobné nešikovnosti ve formulaci zadání – pokud jsou všechna $x = 0$, správně by se automaty měly zastavit, aniž by jakkoliv změnily svůj vnitřní stav. Jenže zadání pouze žádalo vrátit nulový výsledek. Tato drobnost způsobuje, že úloha má velice jednoduché řešení v konstantním čase založené na následujícím triku: v prvním taktu všechny automaty zjistí, jaké dostaly x, a pokud bylo nulové, zastaví se. Navíc všechny nastaví pomocnou proměnnou z na jedničku. Pokud tedy byla všechna x nulová, hned v prvním taktu se všechny automaty zastaví, jak zadání žádá. Pokud někde bylo $x = 1$, výpočet pokračuje druhým taktem, což automaty poznají podle z, a mohou hned vystřelit. My si ovšem ukážeme i řešení původně zamýšlené úlohy:

Nejprve vyřešme, jak zasynchronizovat dva vrcholy l, p spojené cestou, čili jak způsobit, aby nějaká událost ve vrcholu p způsobila jinou událost současně ve vrcholech l a p. K tomu se bude hodit trik s šířením signálů různými rychlostmi použitý v řešení domácího kola. Vyšleme z vrcholu p směrem k vrcholu l dva signály, říkejme jim třeba A a B, přičemž A se bude šířit plnou rychlostí a B rychlostí poloviční. Jakmile A dorazí do vrcholu l, „odrazí se“ a bude pokračovat jako A' stejnou rychlostí zpět k p. Všimněte si, že A' se vrátí do p přesně v okamžiku, kdy B dorazí do l. Program by vypadal takto:

```
var x: 0..3;                    { 1=l, 2=p, 3=p během výpočtu, 0=ostatní }
    y: 0..1 = 0;                { výstup }
    a, aa, b: 0..3 = 0;         { signály a kolik taktů zbývá do jejich předání dál }
begin
   { Na počátku vyšleme signály A a B. }
   if x=2 then begin x:=3; a:=2; b:=3; end;

   { Signál A putuje vlevo rychlostí 1, vlevo se odrazí a je z něj A'. }
   if a>0 then dec(a);
   if S[1].a=1 then a:=1;
   if (x=1) and (a>0) then begin a:=0; aa:=2; end;

   { Signál A' putuje vpravo rychlostí 1. }
   if aa>0 then dec(aa);
   if S[2].aa=1 then aa:=1;

   { Signál B putuje vlevo rychlostí 1/2. }
   if b>0 then dec(b);
   if S[1].b=1 then b:=2;

   { Když A' a B doputují na protilehlý konec, vypálíme. }
   if (x=1) and (b>0) then begin b:=0; y:=1; end;
   if (x=3) and (aa>0) then begin aa:=0; y:=1; end;

   { Není-li co dělat, končíme. }
   if (a=0) and (aa=0) and (b=0) then stop;
end.
```

Když už umíme synchronizovat dvojice vrcholů, můžeme použít metodu rozděl a panuj a vyřešit problém pro celý cyklus. Představíme si cyklus jako interval, jehož počátkem je označený vrchol (lovec, který viděl zvěř) a tentýž vrchol slouží jako zarážka za koncem intervalu. Nalezneme vrchol v polovině intervalu, ten zasynchronizujeme s počátečním vrcholem a tím se

nám interval rozpadl na dvě poloviny, v nichž můžeme tentýž problém s poloviční počtem lovců řešit současně. Pokud bude počet lovců $N = 2^k$, po k iteracích se dobereme k jednoprvkovým intervalům (ke všem současně) a tehdy už může každý lovec rovnou vystřelit.

Zbývá dořešit hledání středu. K tomu můžeme použít opět šíření signálů různými rychlostmi: z počátečního vrcholu vyšleme doprava signál C plnou rychlostí a signál D rychlostí třetinovou. Signál C se od pravého okraje odrazí jako C' a na cestě zpět se potká se signálem D přesně v polovině intervalu (pokud si vrcholy očíslujeme $0, \ldots, N-1$ a N bude sudé, potkají se ve vrcholu $N/2$).

Každá iterace tohoto algoritmu potřebuje lineární počet taktů vzhledem k délce intervalů, které se zrovna zpracovávají. Celková časová složitost tedy bude $O(N + N/2 + N/4 + \ldots + 1) = O(N)$.

Poznámka: Kdyby N nebylo mocninou dvojky, museli bychom se vyrovnat s nerovnoměrným dělením intervalů. To ale není nijak těžké: při hledání středu v intervalu liché délky se zastavíme na $(N-1)/2$ a dokonce poznáme, že se nám to stalo, podle toho, že se signály nepotkají přesně na hranici periody signálu D. Pak můžeme prostřední vrchol intervalu vynechat a zpracovat zbylé stejně velké části identicky. Nakonec nám zbudou jednoprvkové intervaly a mezi některými z nich osamocené vrcholy. Stačí tedy, aby intervaly místo výstřelu nastavily nějakou značku „zamířit“ a v následujícím taktu vystřelí každý lovec, který buďto tuto značku má, nebo ji vidí u svého souseda. Program si tím ovšem komplikovat nebudeme.

```
var x: 0..3;                          { 1=počátek, 2=okraj, 3=střed intervalu, 0=ostatní }
    y: 0..1 = 0;                      { výstřel }
    a, aa, b, c, cc, d: 0..3 = 0;     { signály a kolik taktů zbývá do jejich předání dál }

begin
   { Triviální vstup => vypálíme hned. }
   if (x=1) and (S[1].x=1) then begin y:=1; stop; end;

   { Na počátku vyšleme signály C a D. }
   if x=1 then begin x:=2; d:=3; end;
   if S[2].x=1 then c:=2;

   { Signál C putuje vpravo rychlostí 1, pak se odrazí jako C'. }
   if c>0 then dec(c);
   if S[2].c=1 then c:=1;
   if (x=2) and (c>0) then begin c:=0; cc:=2; end;

   { Signál C' putuje vlevo rychlostí 1. }
   if cc>0 then dec(cc);
   if S[1].cc=1 then cc:=1;

   { Signál D putuje vpravo rychlostí 1/3. }
   if d>0 then dec(d);
   if S[2].d=1 then d:=3;

   { Když se C' a D potkají, máme střed. Vyšleme z něj A a B. }
   if (x=0) and (cc>0) and (d>0) then begin cc:=0; d:=0; x:=3; a:=2; b:=3; end;

   { Signál A putuje vlevo rychlostí 1, tam se odrazí a je z něj A'. }
   if a>0 then dec(a);
   if S[1].a=1 then a:=1;
   if (x=2) and (a>0) then begin a:=0; aa:=2; end;

   { Signál A' putuje vpravo rychlostí 1. }
   if aa>0 then dec(aa);
   if S[2].aa=1 then aa:=1;

   { Signál B putuje vlevo rychlostí 1/2. }
   if b>0 then dec(b);
   if S[1].b=1 then b:=2;

   { Když A' doputuje zpět do středu a B doleva, spustíme se znovu v obou polovinách. }
   if (x=2) and (b>0) then begin b:=0; x:=1; end;
   if (x=3) and (aa>0) then begin aa:=0; x:=1; end;

   { Není-li co dělat, končíme. }
   if (a=0) and (aa=0) and (b=0) and (c=0) and (cc=0) and (d=0) and (x<>1) then stop;
end.
```