P-I-1 Pizza kolem

Úlohu budeme řešit hladově. Nejdříve určíme potřebné počty kusů pizzy jednotlivých velikostí. Tedy kolik potřebujeme celých pizz, kolik kusů velikosti 5/6, kolik kusů velikosti 4/6 atd. To snadno zvládneme při načítání vstupu. Na řezání celých pizz není co řešit. Pro každý kus velký 5/6 potřebujeme zjevně upéct jednu celou pizzu. Po každém kusu velkém 5/6 zbude kousek velký 1/6 a zřejmě nic nezkazíme, když tyto kousky použijeme na pokrytí objednávek na kousky této velikosti. Pokud je náhodou zbytků velkých 1/6 více, než kolik vyžadují objednávky, nezbývá nám než zbytky zahodit (jiné využití pro ně nemáme).

Pro kusy velké 4/6 je situace podobná jako pro kusy velké 5/6. Pro každý kus opět potřebujeme jednu celou pizzu a zbytky velké 2/6 použijeme na pokrytí objednávek na kusy velké 2/6. Pokud je takových objednávek málo, tak kusy velké 2/6 ještě rozřežeme a použijeme na pokrytí zbývajících objednávek na části velikosti 1/6. Zjevně je lepší se nejdříve snažit uspokojit objednávky na velikost 2/6 a až pak na velikost 1/6 – kousek veliký 1/6 můžeme z libovolně velkého kusu odříznout vždy.

Dosud bylo řezání pizzy téměř jednoznačně určené, a tedy námi navržené dělení je nejlepší možné. Nyní se podívejme na kusy velikosti 3/6=1/2 a menší. Objednávky na části velikosti 1/2 vyřešíme tak, že vždy jednu pizzu rozdělíme na dvě poloviny. V následujících odstavcích ukážeme, že tento způsob dělení je optimální. Pokud byl počet takových objednávek lichý, zbude nám polovina pizzy. Pokud je ještě nějaká neuspokojená objednávka na kus velikosti 2/6, odřízneme ze zbylé poloviny tento díl a ten použijeme na pokrytí objednávky. Zbytek pak použijeme na pokrytí objednávek na velikost 1/6. Pokud zbyly ještě nějaké objednávky na kusy velké 2/6 (což je 1/3), budeme další pizzy dělit na třetiny a pokrývat objednávky. Případný zbytek po pokrytí objednávek na třetiny pak použijeme na pokrytí objednávek na šestiny. Pokud ani tak nebyly uspokojeny všechny objednávky na šestiny, vyrobíme ještě dostatečný počet pizz na jejich pokrytí.

Všimněme si, že dohromady nám při uspokojování objednávek na části velké nejvýše 1/2 zbyla méně než jedna pizza:

1) Pokud dokážeme objednávky na poloviny vyřešit beze zbytku (tedy počet objednávek na poloviny je sudý nebo máme dost zbylých objednávek na části velké 2/6 a 1/6), tak po vyřízení objednávek na 2/6 a 1/6 zbude zjevně méně než jedna pizza.
2) Pokud nám zbyla po vyřízení objednávek na poloviny 1/6 pizzy, zjevně už máme vyřízené všechny objednávky na části velké 1/6. Po vyřízení objednávek na části velké 2/6 nám zbudou nejvýše 4/6 pizzy a dohromady tedy zbude méně než jedna pizza.
3) Pokud nám zbude více než 1/6 (2/6 nebo 1/2), zjevně už jsme vyřídili všechny objednávky na části velké 1/6 a 2/6 a celkový zbytek je tedy opět menší než celá pizza.

Když celý náš postup shrneme, tak pro části větší než 1/2 bylo optimální dělení jednoznačně určeno. Zbytky jsme použili na pokrytí menších objednávek, pokud to šlo. Tato část řešení je tedy optimální. Nechť zbylé objednávky na části velké nejvýše 1/2 dají dohromady $k/6$. Na jejich pokrytí potřebujeme zřejmě upéct alespoň $\lceil k/6 \rceil$ pizz a protože nám v této fázi zbyla méně než jedna pizza, tak právě s tolika pizzami jsme si dokázali vystačit. Objednávky na části velikosti nejvýše 1/2 jsme tedy vyřešili také optimálně. Celkově jsme tedy použili nejmenší možný počet pizz.

Algoritmus má lineární časovou a konstantní paměťovou složitost. Program je přímým přepsáním výše uvedených úvah o dělení pizzy.

```
program pizza;
var
  n : word;                              { Počet objednávek }
  pozadavek : array[1..6] of longint;    { Kolik dílů příslušných velikostí potřebuji? }
  napect : longint;                      { Kolik pizz je třeba upéct? }

procedure nacti;
var
  F : Text;
  i, c : Integer;
begin
  for i := 1 to 6 do
    pozadavek[i] := 0;
  assign(F, 'pizza.in');
  reset(F);
  readln(F, n);
  for i := 1 to n do begin
    readln(F, c);
    { Vyřídíme nejdříve počet celých pizz }
    pozadavek[6] := pozadavek[6] + c div 6;
    { Nyní zvýšíme počet dílů příslušné velikosti }
    c := c mod 6;
```

```
    if c > 0 then
      inc(pozadavek[c]);
  end;
  close(F);
end;

procedure vypis;
var F : text;
begin
  assign(F, 'pizza.out');
  rewrite(F);
  writeln(F, napect);
  close(F);
end;

function min(a,b : longint) : longint;
begin
  if a < b then min := a
           else min := b;
end;

procedure spocti;
begin
  { Určitě potřebuji tolik pizz, kolik bylo objednávek na celé pizzy }
  napect := pozadavek[6];
  { Pro každý díl velký 5/6 potřebuji jednu pizzu }
  napect := napect + pozadavek[5];
  { Díly velké 5/6 lze doplnit pouze díly velkými 1/6 }
  pozadavek[1] := pozadavek[1] - min(pozadavek[5], pozadavek[1]);
  { Pro každý díl velký 4/6 potřebuji jednu pizzu }
  napect := napect + pozadavek[4];
  { Doplníme díly velkými 2/6 a 1/6 }
  if pozadavek[4] > pozadavek[2] then begin
    pozadavek[4] := pozadavek[4] - pozadavek[2];
    pozadavek[2] := 0;
    pozadavek[1] := pozadavek[1] - min(2*pozadavek[4], pozadavek[1]);
  end
  else
    pozadavek[2] := pozadavek[2] - pozadavek[4];
  { Díly velké 3/6=1/2 jdou kombinovat spolu }
  napect := napect + (pozadavek[3]+1) div 2;
  { Pokud je počet polovin lichý, doplníme je z 2/6 a 1/6 }
  if pozadavek[3] mod 2 > 0 then begin
    if pozadavek[2] > 0 then begin
      dec(pozadavek[2]);
      if pozadavek[1] > 0 then
        dec(pozadavek[1]);
    end
    else
      pozadavek[1] := pozadavek[1] - min(pozadavek[1], 3);
  end;
  { Nyní pro zbylé díly velké 2/6 = 1/3 }
  napect := napect + (pozadavek[2]+2) div 3;
  { Doplníme případnou 1 necelou pizzu z dílu 1/6 }
  if pozadavek[2] mod 3 > 0 then
    pozadavek[1] := pozadavek[1] - min(pozadavek[1], 2*(3 - pozadavek[2] mod 3));
  { Nyní díly velké 1/6 }
  napect := napect + (pozadavek[1]+5) div 6;
end;

begin
  nacti;
  spocti;
  vypis;
end.
```

P-I-2 Zasypané město

Nejprve si popíšeme jednodušší řešení, které bude časově a paměťově náročnější. V tomto řešení si budeme celou čtvercovou síť uchovávat v paměti jako dvourozměrné pole a zaznamenáme do něj, pod kterými políčky je písek a pod kterými je kamení. Navíc označíme všechna políčka jako *nenavštívená* (nenavštívená políčka budou odpovídat políčkům místností, které jsme dosud nenašli při průchodu sítí). Čtvercovou sítí budeme procházet postupně po řádcích a hledat políčko s pískem, které je zatím *nenavštívené*. Když ho najdeme, víme, že jsme právě objevili novou zasypanou místnost. Nyní přestaneme zpracovávat políčka postupně po řádcích a místo toho teď *navštívíme* všechna políčka nově nalezené místnosti. To uděláme pomocí *prohledávání do šířky*. Začneme v nově nalezeném políčku, které označíme jako *navštívené*. Pak budeme procházet jeho sousední políčka a každé takové políčko, pod kterým je písek a které je *nenavštívené*, označíme jako *navštívené* a zpracujeme. Zpracováním políčka rozumíme to, že zkontrolujeme, zda nemá nějakého *nenavštíveného* souseda a pokud ano, tak tohoto souseda označíme jako *navštíveného* a přidáme jej na konec seznamu políček, která musíme zpracovat. Políčka jsou tedy zpracovávána v pořadí, v jakém jsme je poprvé navštívili.

Poté, co navštívíme a zpracujeme všechna políčka nové místnosti, budeme pokračovat v procházení čtvercové sítě po řádcích tam, kde jsme skončili. Skončíme, až projdeme celou síť.

Takové řešení má časovou i paměťovou složitost lineární vůči počtu políček sítě, tj. $O(NM)$. Celá síť se nám ale vzhledem k omezením ze zadání úlohy do paměti nevejde, a tak musíme navrhnout paměťově úspornější řešení.

Danou síť budeme zpracovávat po řádcích. V jednom kroku zpracujeme vždy jeden řádek, přičemž si budeme pamatovat předchozí řádek. Poté, co řádek zpracujeme, stane se z něj předchozí řádek, načteme nový a budeme pokračovat, dokud nezpracujeme všechna políčka sítě.

Nyní si popíšeme, jak vlastně budeme řádky zpracovávat. Naším cílem je, aby všechna políčka s pískem na zpracovávaném řádku byla očíslována (obarvena) čísly 1 až b. Musí být navíc očíslována tak, že pokud z již zpracovaných řádků (včetně aktuálního) o nějakých dvou políčkách s pískem víme, že patří do společné místnosti, musí dostat v našem obarvení stejnou barvu. V opačném případě musí dostat barvu různou.

Celý algoritmus tedy bude fungovat následovně. Na začátku přidáme před celou síť řádek políček s kamením (ten je už zpracovaný, když na něm žádná políčka s pískem nejsou). Poté načteme další řádek a pomocí předchozího řádku ho zpracujeme, tj. obarvíme ho podle popsaného pravidla (podrobněji si popíšeme tento krok dále). V průběhu zpracovávání dalšího řádku spočítáme, kolik na předchozím řádku skončilo místností (je to počet barev, které nesousedí s žádným pískovým políčkem na aktuálním řádku) a tento počet připočteme k celkovému počtu místností města. Takový algoritmus je jistě správný: každá místnost někdy skončí, takže ji určitě započteme alespoň jednou, navíc ji ale kvůli tomu, jak obarvujeme zpracovávané řádky, nikdy nemůžeme započítat vícekrát než jednou. Složitost algoritmu bude M krát složitost zpracování jednoho řádku.

Ještě tedy musíme vyřešit zpracovávání načteného řádku. Předchozí řádek už máme obarven barvami 1 až b. Aktuální řádek nejprve obarvíme barvami od $b+1$ do c tak, že sousední písková políčka tohoto řádku dostanou stejnou barvu. Poté využijeme informace z předchozího řádku: pokud pískové políčko zpracovávaného řádku sousedí s pískovým políčkem předchozího řádku, musíme „sloučit“ jejich barvy. K tomuto účelu si budeme u každé barvy pamatovat seznam těch barev, kterými je obarvena stejná místnost jako touto barvou. Jedno „sloučení“ pak provedeme tak, že do seznamu jedné slučované barvy přidáme druhou a naopak. Poté, co projdeme celý aktuální řádek a zpracujeme všechna sloučení, získáme u každé barvy seznam ekvivalentních barev (barva a všechny barvy v jejím seznamu obarvují políčka stejné místnosti). Nicméně v seznamu u nějaké barvy nemusí být *všechny* barvy obarvující jednu místnost (třeba u barvy 1, pokud se sloučila 1 s 2 a pak 2 s 3, chybí 3). Naším cílem bude pro jednu barvu najít všechny s ní ekvivalentní barvy. Všechny tyto barvy pak přebarvíme na novou barvu. Pak vezmeme nějakou nepoužitou barvu, opět najdeme všechny ostatní barvy s ní ekvivalentní, všechny je přebarvíme na novou barvu, a tak dál, dokud nepoužijeme všechny původní barvy. Tak získáme obarvení aktuálního řádku novými barvami, které splňuje požadované vlastnosti.

Poslední problém, který musíme vyřešit, je jak pro danou barvu nalézt všechny ostatní, se kterými obarvuje stejnou místnost. Použijeme pro to už popsaný postup prohledávání do šířky: začneme s počáteční barvou a označíme ji jako použitou (ostatní barvy jsou na začátku označeny jako nepoužité). Poté budeme procházet její seznam barev a když narazíme na barvu, která je zatím nepoužitá, přidáme ji do seznamu barev, které musíme zpracovat. Až celý proces skončí, budeme znát všechny barvy, které se zadanou barvou obarvují stejnou místnost.

Povšimněme si nyní, že pro zpracování řádku si stačí pamatovat intervaly tvořené pískem a jejich barvy, tj. nemusíme pracovat s poli délky N, která by reprezentovala předchozí a nový řádek čtvercové sítě. Pokud předchozí řádek obsahuje K_1 intervalů tvořených pískem a nový K_2 takových intervalů, bude nám zpracování těchto dvou řádků trvat $O(K_1+K_2)$, protože počet dvojic protínajících se intervalů, tedy těch, pro které musíme sloučit nějakou dvojici barev, je nejvýše K_1+K_2-1. Samotné slučování barev (prohledávání do šířky) je pak lineární v počtu dvojic barev, které je třeba sloučit. Celková časová složitost našeho algoritmu bude tedy součet hodnot K_1 a K_2 přes všechny řádky, tj. $O(K+M)$ (do odhadu časové složitosti musíme započítat i počet řádků pro případ, že by čtvercová síť obsahovala hodně prázdných řádků, tj. K by bylo mnohem menší než M). Paměťová složitost je nejvýše lineární s počtem políček na jednom řádku, tj. $O(N)$.

```
program mesto;
type usek = record              { datový typ pro načtený úsek písku }
            barva: word;
            zacatek: word;
            konec: word;
          end;
```

```
const maxN = 50000;

type radek = record            { datový typ pro řádek mapy }
             pocet_useku: word;
             pocet_barev: word;
             useky: array [1..maxN] of usek;
           end;

var vstup: text;               { vstupní soubor }
    vystup: text;              { výstupní soubor }
    M, N: word;                { rozměry mapy }
    L: longint;                { počet dosud nalezených místností }
    predchozi_radek: radek;    { řádek, který byl už zpracován }
    novy_radek: radek;         { řádek, který je právě zpracováván }

procedure inicializuj;
begin
  assign(vstup, 'mesto.in');
  reset(vstup);
  readln(vstup, M, N, L);
  L := 0;
  predchozi_radek.pocet_useku := 0;
  predchozi_radek.pocet_barev := 0;
end;

type barva=record               { datový typ pro seznam barev, které mají být sloučeny }
             cislo: word;
             dalsi: ^barva;
           end;

procedure sluc_barvy;
  { provede sloučení místností uložených v predchozi_radek a novy_radek;
    zvýší L o jedna za každou místnost, která už nepokračuje na novém řádku }
var ma_byt_slouceno: array[1..2*maxN] of ^barva;  { seznam barev, které mají být sloučeny }
    vysledna_barva: array[1..2*maxN] of word;     { barva, na kterou má být přebarveno }
    novy_pocet_barev: word;                       { nový počet barev }
    seznam: array[1..2*maxN] of word;             { seznam pro prohledávání do šířky při slučování }
    prvku_v_seznamu: word;                        { počet prvků v seznamu }
    i, j: word;                                   { několik pomocných proměnných }
    b:^barva;
begin
  for i := 1 to novy_radek.pocet_barev do ma_byt_slouceno[i] := nil;
  i := 1; j := 1;
  while (i <= predchozi_radek.pocet_useku) and (j <= novy_radek.pocet_useku) do
    begin
      if (predchozi_radek.useky[i].zacatek < novy_radek.useky[j].konec) and
         (novy_radek.useky[j].zacatek < predchozi_radek.useky[i].konec) then
        begin
          new(b);
          b^.cislo := predchozi_radek.useky[i].barva;
          b^.dalsi := ma_byt_slouceno[novy_radek.useky[j].barva];
          ma_byt_slouceno[novy_radek.useky[j].barva] := b;
          new(b);
          b^.cislo := novy_radek.useky[j].barva;
          b^.dalsi := ma_byt_slouceno[predchozi_radek.useky[i].barva];
          ma_byt_slouceno[predchozi_radek.useky[i].barva] := b;
        end;
      if predchozi_radek.useky[i].konec < novy_radek.useky[j].konec then inc(i) else inc(j);
    end;
  novy_pocet_barev := 0;
  for i := 1 to novy_radek.pocet_barev do vysledna_barva[i] := 0;
  for i := predchozi_radek.pocet_barev+1 to novy_radek.pocet_barev do
    if vysledna_barva[i] = 0 then
      begin
        inc(novy_pocet_barev);
```

```
        vysledna_barva[i] := novy_pocet_barev;
        seznam[1] := i;
        prvku_v_seznamu := 1;
        j := 0;
        while j < prvku_v_seznamu do
          begin
            inc(j);
            b := ma_byt_slouceno[seznam[j]];
            while b<>nil do
              begin
                if vysledna_barva[b^.cislo] = 0 then
                  begin
                    vysledna_barva[b^.cislo] := novy_pocet_barev;
                    inc(prvku_v_seznamu);
                    seznam[prvku_v_seznamu] := b^.cislo;
                  end;
                b := b^.dalsi;
              end;
          end;
      end;
  for i := 1 to predchozi_radek.pocet_barev do
    begin
      if vysledna_barva[i] = 0 then inc(L);
      while ma_byt_slouceno[i] <> nil do
        begin
          b := ma_byt_slouceno[i]^.dalsi;
          dispose(ma_byt_slouceno[i]);
          ma_byt_slouceno[i] := b;
        end;
    end;
  novy_radek.pocet_barev := novy_pocet_barev;
  for i := 1 to novy_radek.pocet_useku do
    novy_radek.useky[i].barva := vysledna_barva[novy_radek.useky[i].barva];
  predchozi_radek := novy_radek;
end;

procedure zpracuj_radek;        { načte řádek a zavolá proceduru sluc_barvy }
var pisek, kameni: word;        { načtená dvojice čísel }
    sloupec: word;              { pozice na načítaném řádku }
begin
  sloupec := 1;
  novy_radek.pocet_useku := 0;
  novy_radek.pocet_barev := predchozi_radek.pocet_barev;
  while sloupec <= N do
    begin
      readln(vstup, pisek, kameni);
      if pisek<>0 then
        begin
          if (novy_radek.pocet_useku > 0) and
             (sloupec = novy_radek.useky[novy_radek.pocet_useku].konec+1) then
            novy_radek.useky[novy_radek.pocet_useku].konec :=
              novy_radek.useky[novy_radek.pocet_useku].konec + pisek
          else
            begin
              inc(novy_radek.pocet_useku);
              inc(novy_radek.pocet_barev);
              novy_radek.useky[novy_radek.pocet_useku].zacatek := sloupec;
              novy_radek.useky[novy_radek.pocet_useku].konec := sloupec+pisek-1;
              novy_radek.useky[novy_radek.pocet_useku].barva := novy_radek.pocet_barev;
            end;
        end;
      sloupec := sloupec + pisek + kameni;
    end;
  sluc_barvy;
end;
```

```
procedure ukonci;  { odsimuluje poslední řádek jako řádek, který je tvořen jen kamením }
begin
  novy_radek.pocet_useku := 0;
  sluc_barvy;
end;

procedure vypis;   { vypíše počet místností do výstupního souboru }
begin
  assign(vystup, 'mesto.out');
  rewrite(vystup);
  writeln(vystup, L);
  close(vystup);
end;

var i: word;
begin
  inicializuj;
  for i:=1 to M do zpracuj_radek;
  ukonci;
  vypis;
end.
```

P-I-3 Okružní jízda

Začněme tím, že si zavedeme označení, které se standardně používá v teorii grafů. Mapě Stínové Prahy budeme říkat *graf*, křižovatkám *vrcholy* a ulicím *hrany*. Hranu, která vede z vrcholu u do vrcholu v, označme uv. Počet vrcholů grafu označme n a počet jeho hran m. *(Vstupní) stupeň* vrcholu je počet hran, které do něj vchází (což v našem případě je rovno počtu hran, které z něj vychází). Hrany e a f na sebe navazují, pokud e vchází do vrcholu, z nějž f vychází. 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 tedy je nalézt uzavřený eulerovský tah, který navíc neobsahuje žádný ze zakázaných přechodů. Nejprve budeme ignorovat zakázané přechody, a najdeme eulerovský tah, který je nemusí respektovat. Poté si ukážeme, jak se se zakázanými přechody vypořádat.

Eulerovský tah zkonstruujeme takto: zvolíme libovolně počáteční vrchol v a vyrazíme z něj po libovolné hraně. Když dorazíme do vrcholu, pokračujeme po libovolné ještě nepoužité hraně, a toto opakujeme, dokud je to možné, tedy dokud z aktuálního vrcholu vede alespoň jedna nepoužitá hrana. Do každého vrcholu vchází stejný počet hran, jaký z něj vychází, proto se můžeme zastavit jedině ve vrcholu v. Tímto získáme nějaký uzavřený tah $e_1, e_2, \ldots, e_k$. Jestliže existuje nějaká dosud nepoužitá hrana e, která navazuje na hranu e_i tohoto tahu, pak stejným způsobem nalezneme uzavřený tah $e, f_1, \ldots, f_t$ z dosud nepoužitých hran, který obsahuje hranu e, a tahy spojíme – nový tah bude $e_1, \ldots, e_i, e, f_1, \ldots, f_t, e_{i+1}, \ldots e_k$. Budeme říkat, že nový tah vznikl *slepením přes přechody* e_ie_{i+1} *a* f_te. Tuto operaci opakujeme a prodlužujeme nalezený tah. Skončit můžeme ve dvou případech. Buď výsledný tah obsahuje všechny hrany, potom je eulerovský. Nebo se z hran obsažených v tahu nedá nijak dostat do zbývajících hran (tvoří *komponentu* nesouvislého grafu), a v takovém případě eulerovský tah nemůže existovat. Tento postup se dá implementovat s lineární časovou a paměťovou složitostí $O(m + n)$.

Nyní se zabývejme zakázanými přechody. Nejprve se můžeme zbavit vrcholů stupně 1 a 2:

- Zakázaný přechod u vrcholu vstupního stupně 1 nám znemožní tímto vrcholem projít, takže v tomto případě eulerovský tah nemůže existovat.
- Zakázaný přechod u vrcholu stupně 2 nám přesně předepíše, jak tímto vrcholem projít. Máme-li tedy hrany u_1v, u_2v, vw_1 a vw_2, a přechod z u_1v na vw_2 je zakázaný, můžeme vrchol v i s ním sousedícími hranami vyhodit, a přidat místo nich hrany u_1w_1 a u_2w_2. V novém grafu existuje povolený eulerovský tah, právě tehdy když existuje v původním grafu. Pro implementaci je jednodušší graf neměnit a jen si v algoritmu dávat pozor, abychom nepoužili zakázaný přechod u v – tedy pokud do něj dorazíme při procházení, odejít správnou hranou a nezkoušet v něm tah navázat.

Můžeme tedy předpokládat, že graf, kterým se zabýváme, obsahuje pouze vrcholy stupně alespoň 3. V tomto případě výše popsaným algoritmem nalezneme eulerovský tah, bez ohledu na zakázané přechody. Jestliže takový tah neexistuje, zřejmě tím spíš neexistuje takový, který by respektoval zakázané přechody. Jestliže nějaký eulerovský tah existuje, lze ho opravit tak, aby nepoužíval zakázané přechody:

Buď $e_1, \ldots, e_m$ uzavřený eulerovský tah, a nechť je bez újmy na obecnosti přechod e_me_1 u vrcholu v zakázaný. Protože v má stupeň alespoň 3, existují další dvě hrany e_a a e_b $(a < b)$, které vchází do v. Pak tah

$$e_1, e_2, \ldots, e_a, e_{b+1}, e_{b+2}, \ldots, e_m, e_{a+1}, e_{a+2}, \ldots, e_b$$

je také eulerovský a navíc nepoužívá zakázaný přechod u v. Takto postupně odstraníme všechny zakázané přechody. Zakázaných přechodů je n a na odstranění jednoho z nich potřebujeme čas $O(m)$ (musíme znát pořadí hran u vrcholu v v tahu,

který upravujeme), tedy časová složitost této části algoritmu je $O(mn)$. Zkusme se zamyslet, zda bychom se zakázaných přechodů nedokázali zbavit efektivněji.

Budeme provádět výše popsaný lineární algoritmus pro nalezení eulerovského tahu: Řekněme, že už jsme zkonstruovali uzavřený tah $T_1 = e_1, \ldots, e_k$. Náš algoritmus zaručí, že:

(1) tento tah neobsahuje zakázané přechody a

(2) pokud je u vrcholu v použitá alespoň jedna hrana a zakázaný přechod u v je ef, pak alespoň jedna z hran e a f je použitá.

Snažíme se tento tah prodloužit a přitom zachovat tyto vlastnosti. Vybereme si vrchol v stupně alespoň 3 ležící na tahu T_1, z nějž vede alespoň jedna nepoužitá hrana e. Pak procházíme graf a konstruujeme nový tah $T_2 = ef_1 \ldots f_t$ z dosud nepoužitých hran. Při výběru další hrany do tahu T_2 se řídíme následujícími pravidly:

- Nikdy do T_2 nepřidáváme zakázaný přechod.
- Jestliže je xy poslední hrana tahu, u vrcholu y je zakázaný přechod z hrany $x'y$ na hranu yz (kde $x' \neq x$) a hrana yz je ještě nepoužitá, přidáme do tahu právě hranu yz.

Nemůže se stát, že bychom došli do nějakého vrcholu x a jediný zbývající přechod by byl zakázaný: Pokud x patří do T_1, pak je alespoň jedna z hran ze zakázaného přechodu u x je obsažena v T_1 díky vlastnosti (1). Vrchol x má stupeň x alespoň dva, tedy jestliže x v T_1 není a vede z něj už jen jedna nepoužitá hrana, museli jsme ho už alespoň jednou navštívit při konstrukci tahu T_2 a díky druhému z pravidel pro tvorbu T_2 jsme tehdy použili alespoň jednu z hran zakázaného přechodu. Snadno také nahlédneme, že T_2 díky popsaným pravidlům splňuje podmínky (1) a (2).

Potřebujeme ještě tahy T_1 a T_2 spojit. Jelikož alespoň jedna hrana ze zakázaného přechodu u v je použita v T_1, konstrukce tahu T_2 se zastaví až tehdy, když v tazích T_1 a T_2 budou použity všechny hrany u vrcholu v. Alespoň jeden z T_1 a T_2 proto prochází vrcholem v alespoň dvakrát. Předpokládejme například, že T_2 projde v alespoň dvakrát, po přechodech f_te a f_sf_{s+1}. Tahy T_1 a T_2 můžeme slepit buď přes přechody e_ie_{i+1} a f_te nebo přes přechody e_ie_{i+1} a f_sf_{s+1}. Alespoň jedna z těchto možností nezpůsobí přidání zakázaného přechodu do tahu a tuto možnost si zvolíme. V tomto případě nemusíme znovu procházet celý tah, abychom našli hrany f_s a f_{s+1} – stačí si pro každou použitou hranu pamatovat jejího následníka v tahu, a projít hrany vstupující do vrcholu v. Jelikož spojování provádíme v každém vrcholu nejvýše jednou, časová složitost tohoto postupu je $O(m + n)$.

```
program eulerovsky_tah;

const MAXN = 100;

type phrana = ^hrana;
     hrana = record
               z, k: integer;          { hrana z vrcholu z do vrcholu k }
               pr, dal: phrana;        { předchozí a následující hrana v seznamu }
               tah: integer;           { číslo tahu, do nějž byla hrana přidána }
               tah_p, tah_d: phrana;   { předchozí a následující hrana v tahu }
             end;

     vrchol = record
                stupen: integer;       { stupeň vrcholu }
                pouzite: hrana;        { hlava seznamu použitých hran z vrcholu }
                nepouzite: hrana;      { hlava seznamu nepoužitých hran }
                zak_z, zak_do: phrana; { přechod ze zak_z na zak_do je zakázaný }
              end;

var n: integer;                        { počet vrcholů grafu }
    a: integer;                        { číslo aktuálního tahu }
    v: integer;                        { aktuální vrchol }
    graf: array[1..MAXN] of vrchol;
    nr: integer;
    rozpracovane: array[1..MAXN] of integer; { seznam rozpracovaných vrcholů }
    nh: integer;                       { počet hotových vrcholu }

{ odebere hranu ze seznamu }
procedure odeber (hrana: phrana);
begin
  hrana^.dal^.pr := hrana^.pr;
  hrana^.pr^.dal := hrana^.dal;
  hrana^.dal := nil;
  hrana^.pr := nil;
end;
```

```
{ přidá hranu na začátek seznamu }
procedure pridej (var seznam: hrana; hrana: phrana);
begin
  hrana^.dal := seznam.dal;
  hrana^.pr := @seznam;
  seznam.dal := hrana;
  hrana^.dal^.pr := hrana;
end;

{ vrátí true, pokud je seznam prázdný }
function prazdny (var seznam: hrana): boolean;
begin
  prazdny := (seznam.dal = @seznam);
end;

{ označí hranu daným číslem, přeřadí ji do seznamu použitých hran, a případně přiřadí
  vrchol, z nejž hrana vede, do seznamu rozpracovaných hran }
procedure oznac_hranu (hrana: phrana; cislo: integer);
var vrchol: integer;
begin
  vrchol := hrana^.z;
  hrana^.tah := cislo;

  odeber (hrana);
  if prazdny (graf[vrchol].nepouzite) then
    inc (nh);

  if prazdny (graf[vrchol].pouzite) and (graf[vrchol].stupen <> 2) then
    begin
      inc (nr);
      rozpracovane[nr] := vrchol;
    end;
  pridej (graf[vrchol].pouzite, hrana);
end;

{ vrátí true pokud je přechod z hrany Z na hranu K zakázaný }
function zakazano (z, k: phrana): boolean;
var vrchol: integer;
begin
  if z = nil then
    zakazano := false
  else
    begin
      vrchol := z^.k;
      if k^.z <> vrchol then
        writeln('Chyba');
      zakazano := (graf[vrchol].zak_z = z) and (graf[vrchol].zak_do = k);
    end;
end;

{ vybere nepoužitou hranu z vrcholu, na kterou je povoleno přejít z dané hrany }
function vyber_hranu (vrchol: integer; hrana: phrana): phrana;
var ah: phrana;
begin
  ah := graf[vrchol].nepouzite.dal;

  while (ah <> @graf[vrchol].nepouzite) and zakazano (hrana, ah) do
    ah := ah^.dal;

  if ah = @graf[vrchol].nepouzite then
    vyber_hranu := nil
  else
    vyber_hranu := ah;
end;
```

```
{ označuje tah z nepoužitých hran, začínající hranou PRVNI, číslem CISLO.
  Pokud PRVNI_TAH je true, tah může skončit, jakmile dorazí na hranu, která
  naváže na hranu PRVNI; jinak musí využít všechny hrany z vrcholu PRVNI^.z }
procedure oznac_tah (prvni: phrana; cislo: integer; prvni_tah: boolean);
var ahrana, prhrana: phrana;
    avrchol: integer;
begin
  prhrana := nil;
  ahrana := prvni;

  repeat
    oznac_hranu (ahrana, cislo);
    if prhrana <> nil then
      prhrana^.tah_d := ahrana;
    ahrana^.tah_p := prhrana;

    prhrana := ahrana;
    avrchol := ahrana^.k;
    ahrana := vyber_hranu (avrchol, ahrana);
  until (ahrana = nil)
        or (prvni_tah and (prhrana^.k = prvni^.z)
            and not zakazano (prhrana, prvni));

  prhrana^.tah_d := prvni;
  prvni^.tah_p := prhrana;
  if prvni^.z <> prhrana^.k then
    writeln('Chyba');
end;

{ pokud spojení tahu přes hrany H1 a H2 nevytvoří zakázaný přechod,
  spojí je a vrátí true, jinak vrátí false }
function spoj (h1, h2: phrana): boolean;
var p1, p2: phrana;
begin
  p1 := h1^.tah_p;
  p2 := h2^.tah_p;
  if zakazano (p1, h2) or zakazano (p2, h1) then
    spoj := false
  else
    begin
      p1^.tah_d := h2;
      h2^.tah_p := p1;
      p2^.tah_d := h1;
      h1^.tah_p := p2;
      spoj := true;
    end;
end;

{ spojí A-tý tah v daném vrcholu s předchozími }
procedure spoj_tahy (vrchol: integer);
var stara, nova, navic, ah: phrana;
    t: boolean;
begin
  ah := graf[vrchol].pouzite.dal;
  stara := nil;
  nova := nil;
  navic := nil;
  while ah <> @graf[vrchol].pouzite do
    begin
      if (ah^.tah = a) and (nova = nil) then
        nova := ah
      else if (ah^.tah < a) and (stara = nil) then
        stara := ah
      else
        navic := ah;
```

```
      ah := ah^.dal;
    end;
  if not spoj (stara, nova) then
    begin
      if navic^.tah = a then
        t := spoj (stara, navic)
      else
        t := spoj (nova, navic);
      if not t then
        writeln ('Chyba');
    end;
end;

{ najde tah z nepoužitých hran začínající v daném vrcholu, a označí jeho hrany daným číslem }
procedure najdi_tah (cislo, vrchol: integer);
var prvni: phrana;
begin
  prvni := vyber_hranu (vrchol, nil);
  oznac_tah (prvni, cislo, cislo = 1);
end;

{ najde hranu z vrcholu U do vrcholu V }
function najdi_hranu (u, v: integer): phrana;
var ah: phrana;
begin
  ah := graf[u].nepouzite.dal;
  while ah^.k <> v do
    ah := ah^.dal;
  najdi_hranu := ah;
end;

{ načte graf ze souboru fin. Pokud graf obsahuje vrcholy stupně 1, vrátí false, jinak true }
function nacti_graf (var fin: text): boolean;
var m, i, u, v: integer;
    h, d: phrana;
begin
  readln (fin, n, m);

  for i := 1 to n do
    with graf[i] do
      begin
        stupen := 0;
        pouzite.dal := @pouzite;
        pouzite.pr := @pouzite;
        nepouzite.dal := @nepouzite;
        nepouzite.pr := @nepouzite;
      end;

  for i := 1 to m do
    begin
      readln (fin, u, v);
      new (h);
      h^.z := u;
      h^.k := v;
      h^.pr := nil;
      h^.dal := nil;
      h^.tah := 0;
      h^.tah_p := nil;
      h^.tah_d := nil;
      inc (graf[u].stupen);
      pridej (graf[u].nepouzite, h);
    end;

  for i := 1 to n do
    begin
```

```
      readln (fin, u, v);
      h := najdi_hranu (u, i);
      d := najdi_hranu (i, v);
      graf[i].zak_z := h;
      graf[i].zak_do := d;
      { hranu iv přesuneme na začátek seznamu, takže bude vždy první vybrána do tahu }
      odeber (d);
      pridej (graf[i].nepouzite, d);
    end;

  nacti_graf := true;
  for i := 1 to n do
    if graf[i].stupen < 2 then
      nacti_graf := false;
end;

{ vypíše eulerovský tah v grafu do souboru FOUT }
procedure vypis_tah (var fout: text);
var prvni, ah: phrana;
begin
  prvni := graf[1].pouzite.dal;
  ah := prvni^.tah_d;
  write (fout, prvni^.z);
  while ah <> prvni do
    begin
      write (fout, ' ', ah^.z);
      ah := ah^.tah_d;
    end;
end;

var fin, fout: text;

begin
  assign (fin, 'okruh.in');
  reset (fin);
  assign (fout, 'okruh.out');
  rewrite (fout);

  if not nacti_graf (fin) then
    begin
      writeln (fout, 'Okruzni jizda neexistuje.');
      exit;
    end;

  a := 1;
  najdi_tah(a, 1);

  while nr <> 0 do
    begin
      v := rozpracovane[nr];
      dec (nr);
      inc (a);
      najdi_tah (a, v);
      spoj_tahy (v);
    end;

   if nh = n then
     vypis_tah (fout)
   else
     writeln (fout, 'Okruzni jizda neexistuje.');
end.
```

P-I-4 Grafomat

Pro nalezení nejkratší cesty mezi vrcholy v a w použijeme podobně jako v Příkladu 1 prohledávání do šířky z vrcholu v, tentokrát si ale každý označený vrchol zapamatujeme, po které hraně značku přijal (pokud takových hran bude víc, vybere si libovolnou z nich).

Jakmile prohledávání dospěje do cílového vrcholu w, budeme postupovat pozpátku proti směru zapamatovaných hran a přitom vyznačovat nalezenou cestu.

Tento algoritmus funguje, ale je-li cesta krátká, tráví zbytečně mnoho času tím, že i po nalezení vrcholu w v prohledávání pokračuje, až projde všechny dosažitelné vrcholy grafu. Jak ale prohledávání zastavit? Jakmile se vrchol v dozví, že w bylo nalezeno (dorazí do něj vytyčování cesty), začne se z něj šířit ještě jedna vlna značek, která bude mít za úkol dostihnout tu první, zastavit ji a sama při tom zmizet. Aby to fungovalo, potřebujeme, aby druhá vlna postupovala rychleji než ta první, a jelikož ji nemůžeme zrychlit, namísto toho vše ostatní dvakrát zpomalíme.

Časovou složitost spočítáme následovně: pokud je délka hledané cesty l, dojde první vlna do w po $2l$ taktech, vytyčování se vrátí do v po dalších $2l$ taktech, takže druhá vlna je oproti té první opožděna o $4l$ taktů. V čase t od spuštění algoritmu tedy první vlna dorazí do vzdálenosti $t/2$, zatímco druhá do $t-4l$. Proto se potkají, až bude $t/2 = t-4l$, čili $t = 8l$. Časová složitost je tedy lineární v l.

Program je přímočarou implementací tohoto algoritmu:

```
var x: 0..2;                            { vstupní značky }
    y: 0..1 = 0;                        { výstupní značky }
    s: 0..2 = 0;                        { stav prohledávání: 1=dorazila první vlna, 2=už i druhá }
    b: 0..3 = 0;                        { kterou hranou jsme značku přijali, je-li s>0 }
    p: 0..1 = 0;                        { počítadlo pro zpomalování }
    i: 1..3 = 1;                        { pomocná proměnná }
begin
   p := 1-p;
   if (s=1) and ((S[1].s=2) or (S[2].s=2) or (S[3].s=2)) then
      s := 2                            { druhá vlna běží plnou rychlostí }
   else if p=1 then                     { ostatní části programu běží zpomaleně }
      case s of
         0: begin
               if x=1 then s := 1;      { začátek první vlny }
               for i := 1 to 3 do       { pokračování první vlny }
                  if S[i].s = 1 then begin
                     s := 1;
                     b := i;
                  end;
               if s=0 then stop;        { stále se nic neděje }
            end;
         1: begin
               if x=2 then y := 1       { první vlna dorazila do cíle => jdeme zpět }
               for i := 1 to 3 do       { zpětný průchod }
                  if (S[i].y = 1) and (S[i].b = P[i]) then y := 1;
               if (y=1) and (x=1) then s := 2;  { zpětný průchod dorazil do počátku }
            end;
         2: stop;
      end;
end.
```