68. ročník Matematické olympiády – 2018/2019

Řešení úloh domácího kola kategorie P

P-I-1 Bourání města

Úlohu budeme popisovat v řeči teorie grafů.* V této terminologii se každé křižovatce říká *vrchol* a každé ulici *hrana.* Počet ulic vycházejících z dané křižovatky označujeme jako *stupeň* daného vrcholu. Celý Kocourkov je *neorientovaný graf.* Naším úkolem je nalézt *podgraf* zadaného grafu obsahující všechny vrcholy a vybraných K hran takový, že stupně všech jeho vrcholů jsou sudé. V řešení budeme také používat pojmy *cesta* (posloupnost různých vrcholů grafu taková, že každé dva po sobě jdoucí jsou spojeny hranou) a *komponenta souvislosti* (maximální podgraf takový, že mezi každými dvěma jeho vrcholy vede cesta). Dále *paritou* čísla myslíme zbytek po vydělení tohoto čísla dvěma (tedy je-li liché, nebo sudé).

Začněme s jednoduššími podúlohami. Platí-li $M = K$, tedy nemůžeme mazat žádné hrany, stačí pouze v čase $\mathcal{O}(M + N)$ ověřit, zda jsou stupně všech vrcholů grafu sudé. Platí-li $M \leq 20$, stačí vygenerovat všechny podgrafy grafu a ověřit, zda nějaký obsahuje všechny speciální hrany a zároveň jsou všechny stupně jeho vrcholů sudé. Časová složitost tohoto postupu je $\mathcal{O}((M+N)\cdot 2^M)$, neboť možných podgrafů je stejně jako možností, jak vybrat nějakou podmnožinu hran, což je 2^M.

Nyní vyřešme podúlohu s $K \leq 1$. Je-li $K = 0$, jistě bude vyhovovat, když vrátíme prázdný graf, neboli zbořím celé město. Případ $K = 1$ je již zajímavější: nechť uv je hrana, kterou nesmíme smazat. Povšimněme si, že existuje-li v grafu kromě hrany uv ještě jiná cesta mezi vrcholy u a v, můžeme za náš podgraf zvolit tuto posloupnost hran společně s hranou uv. Každý vrchol totiž v takovém případě bude mít stupeň 2, nebo 0.

Pokud po odebrání hrany mezi u a v již mezi nimi neexistuje žádná cesta, naše úloha nemá řešení, což nahlédneme s pomocí následujícího pozorování. Platí, že součet stupňů všech vrcholů libovolného grafu je roven dvojnásobku počtu hran tohoto grafu. To proto, že každou hranu v tomto součtu započítáme dvakrát. V libovolném grafu tedy musí být součet stupňů jeho vrcholů sudé číslo. Jenže v našem případě, kdy u a v jsou po odebrání hrany uv v rozdílných komponentách původního grafu, chceme pro komponentu obsahující u najít její podgraf takový, že všechny jeho stupně jsou sudé kromě stupně vrcholu u, který má být lichý. Potom by ale i součet stupňů vrcholů této komponenty byl lichý, což je ve sporu se zmíněným pozorováním. Pro vyřešení případu $K \leq 1$ tedy stačí ověřit, zda jsou u a v po odebrání hrany uv ve stejné komponentě, což lze zařídit v čase $\mathcal{O}(M + N)$ např. prohledáváním do hloubky.

Pro případ obecného K si nejprve povšimněme, že hrany, které nesmíme bořit, nejsou moc zajímavé. Pro každý vrchol nás totiž zajímá jen parita počtu nezbořitelných hran, se kterými vrchol sousedí, neboť ta nám určuje paritu, kterou musí mít

* viz například *https://ksp.mff.cuni.cz/kucharky/grafy/*

počet zbylých nezbořených hran, aby měl vrchol sudý stupeň. Řešíme vlastně úlohu, kdy na vstupu je graf tvořený zbylými hranami a pro každý vrchol požadujeme, aby jeho výsledný stupeň byl lichý, nebo sudý; chceme odebrat nějaké hrany tak, aby všechny tyto požadavky byly splněny.

Zpracujeme nyní zvlášť každou komponentu souvislosti tohoto grafu. Je-li součet požadovaných parit stupňů v nějaké komponentě lichý, z již učiněného pozorování víme, že úloha nemá řešení. Zbývá pouze nějaké řešení najít, je-li součet parit v dané komponentě sudý.

To lze udělat podobně jako pro případ $K = 1$ tak, že budeme postupně hledat cesty mezi dvojicemi vrcholů s lichým stupněm ve stejné komponentě. Každou hranu na nalezené cestě pak vždy do podgrafu přidáme, pokud v něm není, a naopak odebereme v opačném případě. Paritu počtu sousedících hran tak změníme jen pro dva krajní vrcholy každé cesty. Takové řešení má časovou složitost $\mathcal{O}(K \cdot (M + N))$. Pro případ $K = 1$ tedy toto řešení stále běží v lineárním čase.

Nakonec vysvětlíme řešení, které dostalo plný počet bodů. Pustíme na danou komponentu algoritmus prohledávání do hloubky s počátkem v libovolném vrcholu. Při průchodu grafem si některé hrany budeme barvit; barevné hrany pak po skončení algoritmu budou tvořit kýžený podgraf. Pokaždé, když algoritmus prohledávání do hloubky naposledy opouští nějaký vrchol, ověříme, zda je požadovaná parita tohoto vrcholu stejná jako jeho parita v podgrafu určeném barevnými hranami. Pokud tomu tak není, obarvíme hranu spojující daný vrchol s jeho otcem, tedy vrcholem, z kterého jsme do aktuálního vrcholu přišli. Aktuální vrchol následně označíme za vyřešený – jeho parita v barevném podgrafu je nyní stejná jako požadovaná parita. Povšimněme si, že po obarvení hrany se v barevném podgrafu změní i parita otce. Tento vrchol jsme ale ještě neoznačili za vyřešený, a jeho paritu tedy později případně opravíme.

Tímto způsobem jsme dostali podgraf takový, že všechny vrcholy kromě toho, kde začalo prohledávání (ten totiž nemá otce), mají požadovanou paritu počtu sousedících hran. Nahlédneme, že i tento počáteční vrchol bude mít v našem podgrafu požadovanou paritu. Požadovaná parita tohoto vrcholu je určena zbylými požadovanými paritami a předpokladem, že součet všech těchto parit je sudý. Zároveň parita tohoto vrcholu ve vytvořeném podgrafu je obdobně určena paritami zbylých vrcholů (které jsou stejné jako jejich požadované parity) a podmínkou, že součet všech těchto parit je sudý. Z toho už ale plyne, že požadovaná parita tohoto vrcholu je vždy stejná jako jeho parita v podgrafu, který vrátí algoritmus.

Celý algoritmus se dá jednoduše implementovat s pomocí prohledávání do hloubky a jeho časová i paměťová složitost je tak $\mathcal{O}(M + N)$.

```
#include <bits/stdc++.h>

using namespace std;
int N, M, K;
vector<bool> odd;  // pro každý vrchol, zda s ním sousedí liše nezbořitelných cest
vector<vector<int>> G;     // pro každý vrchol seznamy id hran, které z něj vedou
vector<pair<int,int>> E;   // seznam všech nezbořitelných hran
vector<bool> keep;         // pro každou nedůležitou hranu, zda ji ponecháme
vector<bool> visited;      // zda jsme už vrchol navštívili v DFS
```

```
// vrchol na druhém konci dané hrany
inline int other(int e, int v)
{
    return (E[e].first == v) ? E[e].second : E[e].first;
}

// vrací, zda v podstromu vede nahoru hrana, kterou je potřeba spárovat
bool dfs(int v)
{
    assert(!visited[v]);
    visited[v] = true;
    for (auto e : G[v]) {
        int s = other(e, v);
        // pokud jsme vrchol ještě nenavštívili, spustíme DFS a podíváme se,
        // zda přebytek vyjde lichý (využíváme zkráceného vyhodnocování: pokud
        // už jsme vrchol navštívili, dfs(s) se neprovede)
        if (!visited[s] && dfs(s))
            odd[v] = !odd[v], keep[e] = true;
    }

    return odd[v];
}

int main(int argc, char *argv[])
{
    scanf("%d %d %d", &N, &M, &K);
    odd.resize(N);
    G.resize(N);
    visited.resize(N);
    keep.resize(M - K);

    for (int i = 0; i < M; i++) {
        int a, b;
        scanf("%d %d", &a, &b);
        a--, b--;
        if (i < K) {
            // Nezbořitelné hrany rovnou zahazujeme, jen podle nich
            // přepočítáváme parity ve vrcholech
            odd[a] = !odd[a], odd[b] = !odd[b];
        } else {
            E.push_back({a, b});
            G[a].push_back(i - K), G[b].push_back(i - K);
            // Posouváme id, aby první nedůležitá hrana měla id 0
        }
    }

    for (int i = 0; i < N; i++) {
        if (!visited[i] && dfs(i)) {
            // Na vrchol jsme pustili DFS a to nám vrátilo lichý přebytek, což
            // nutně znamená, že v komponentě je lichý počet lichých vrcholů
            printf("-1\n");
            return 0;
        }
    }

    int destroy = M - K;
    for (int e = 0; e < keep.size(); e++)
        destroy -= keep[e];
```

```
    printf("%d\n", destroy);
    for (int e = 0; e < keep.size(); e++)
        if (!keep[e])
            printf("%d %d\n", E[e].first + 1, E[e].second + 1);
}
```

P-I-2 Bonbóny

Místo počtů bonbónů, které Anička a Honzík dostanou za každý úsek, stačí uvažovat jejich rozdíl – zajímá nás, zda existuje výlet, na němž tento rozdíl bude nenulový. Úlohu si tedy můžeme v řeči teorie grafů přeformulovat takto: Máme zadán orientovaný graf s ohodnocenými hranami a zajímá nás, zda v něm existuje uzavřený sled s nenulovým součtem ohodnocení.

Začněme nejprve jednodušším případem, kdy mezi každými dvěma vrcholy vede orientovaná cesta (graf je tedy silně souvislý). Vyberme si libovolný vrchol v_0 jako počáteční, udělejme z něj prohledávání do hloubky, a na prohledávaných cestách si průběžně počítejme součet ohodnocení. Takto v lineárním čase pro každý vrchol v najdeme cestu P_v z v_0 do v a určíme součet s_v ohodnocení hran této cesty. Prohledáním do hloubky proti směru hran obdobně pro každý vrchol v najdeme cestu Z_v z v do v_0 a určíme součet s'_v ohodnocení hran této cesty. Zjevně pokud $s'_v \neq -s_v$ pro nějaký vrchol v, pak spojení P_v a Z_v je uzavřený tah se součtem ohodnocení $s_v + s'_v \neq 0$, a můžeme vypsat odpovídající výlet.

Předpokládejme tedy, že $s'_v = -s_v$ pro každý vrchol v. Uvažme nyní libovolnou hranu uv s ohodnocením h. Pokud $h \neq s_v - s_u$, pak spojení P_u, uv a Z_v je uzavřený tah se součtem ohodnocení $s_u + h + s'_v = s_u + h - s_v \neq 0$, a znovu můžeme vypsat odpovídající výlet. Ověření této podmínky pro všechny hrany nám zabere také lineární čas.

Zbývá případ, kdy ohodnocení každé hrany uv je rovno $s_v - s_u$. Pak tvrdíme, že součet ohodnocení hran každého uzavřeného sledu je 0: Uvažme sled $v_1 v_2 \dots v_k$, kde $v_k = v_1$. Pak součet ohodnocení hran tohoto sledu je $(s_{v_2} - s_{v_1}) + (s_{v_3} - s_{v_2}) + \dots + (s_{v_k} - s_{v_{k-1}}) = s_{v_k} - s_{v_1} = 0$.

Výše popsaný postup nám tedy umožňuje najít uzavřený tah s nenulovým součtem ohodnocení, nebo rozhodnout, že neexistuje v případě, že graf je silně souvislý; a časová složitost je lineární. Každý uzavřený tah je nutně obsažen v komponentě silné souvislosti, v obecnosti tedy stačí úlohu vyřešit zvlášť v každé komponentě. Jak komponenty silné souvislosti najít?

Nejprve si vrcholy grafu setřídíme tak, aby pro každé dvě různé komponenty silné souvislosti K_1 a K_2 platilo následující tvrzení: existuje-li nějaká cesta z K_1 do K_2, pak nějaký vrchol K_1 je v pořadí za všemi vrcholy K_2. Takové uspořádání dostaneme například prohledáváním do hloubky tak, že každý vrchol přidáme poté, co z něj ukončíme průchod: Povšimněme si, že z K_2 nevede cesta do K_1, protože K_1 a K_2 jsou různé komponenty silné souvislosti. Uvažme první vrchol v komponenty K_1, který prohledávání navštíví. Pokud jsme před v v prohledávání navštívili nějaký vrchol z cesty z K_1 do K_2 (včetně K_2), museli jsme již dokončit průchod všemi vrcholy K_2, jelikož z K_2 nevede cesta do K_1; proto všechny vrcholy K_2 jsou

v pořadí před v. Jinak se během prohledávání z v dostaneme do K_2 a než se do v vrátíme, celé K_2 prohledáme; opět tedy do pořadí přidáme všechny vrcholy K_2 předtím, než dokončíme prohledávání z v a přidáme v do pořadí.

Nyní se podívejme na poslední vrchol w v pořadí; nechť K je jeho komponenta silné souvislosti. Všechny vrcholy, které jsou z něj dosažitelné cestami proti směru hran, musí patřit do K: Kdyby patřily do nějaké jiné komponenty silné souvislosti K', pak by z K' vedla cesta do K a dle předchozího odstavce by nějaký vrchol K' musel být za w. Můžeme tedy vrcholy komponenty K určit prohledáváním proti směru hran z w. Poté vrcholy K odebereme a úvahu opakujeme pro poslední vrchol zbývající v pořadí. Takto najdeme komponenty silné souvislosti v lineárním čase.

Časová i paměťová složitost celého postupu je tedy lineární, $\mathcal{O}(N + M)$.

```
#include <cstdio>
#include <list>
#include <vector>
using namespace std;

struct hrana
{
  int id; // číslo hrany
  int delta; // rozdíl počtu bonbónů
  int odkud, kam; // spojené vrcholy
};

struct vrchol
{
  bool zpracovany;     // zda je vrchol v dříve zpracované komponentě
  bool akt_komponenta; // zda je vrchol v aktuálně zpracovávané komponentě
  bool navstiveny;     // zda jsme vrchol navštívili v aktuálním průchodu
  hrana *P_posl;       // poslední hrana na cestě P_v do tohoto vrcholu v
  int s_v;             // součet ohodnocení hran na cestě P_v
  hrana *Z_prv;        // první hrana na cestě Z_v z tohoto vrcholu v
  int sp_v;            // Součet ohodnocení hran na cestě Z_v
  list<hrana *> vstupuji, vychazi;  // příchozí a odchozí hrany
  vrchol(void) : zpracovany(false), akt_komponenta(false), navstiveny(false),
                 P_posl(0), s_v(0), Z_prv(0), sp_v(0), vstupuji(), vychazi()
    { }
};

// Seznam hran a vrcholů grafu
static vector<hrana> hrany;
static vector<vrchol> vrcholy;

// Smazání značek navštívenosti ve zpracovávané komponentě k
static void smaz_znacky(const list<int> &k)
{
  for (int v : k)
    vrcholy[v].navstiveny = false;
}

// Nalezení cest P_v a Z_v.
static void najdi_P_v(int v, int delta = 0, hrana *posl = 0)
{
  if (!vrcholy[v].akt_komponenta || vrcholy[v].navstiveny)
    return;
```

```
  vrcholy[v].navstiveny = true;
  vrcholy[v].s_v = delta;
  vrcholy[v].P_posl = posl;

  for (hrana *e : vrcholy[v].vychazi)
    if (e != posl)
      najdi_P_v(e->kam, delta + e->delta, e);
}

static void najdi_Z_v(int v, int delta = 0, hrana *posl = 0)
{
  if (!vrcholy[v].akt_komponenta || vrcholy[v].navstiveny)
    return;
  vrcholy[v].navstiveny = true;
  vrcholy[v].sp_v = delta;
  vrcholy[v].Z_prv = posl;

  for (hrana *e : vrcholy[v].vstupuji)
    if (e != posl)
      najdi_Z_v(e->odkud, delta + e->delta, e);
}

// Vypíše cestu P_v nebo Z_v
static void vypis_P_v(int v)
{
  hrana *e = vrcholy[v].P_posl;

  if (e)
    {
      vypis_P_v(e->odkud);
      printf("%d ", e->id);
    }
}

static void vypis_Z_v(int v)
{
  hrana *e = vrcholy[v].Z_prv;

  if (e)
    {
      printf("%d ", e->id);
      vypis_Z_v(e->kam);
    }
}

/* Ověří, zda v komponentě k existuje uzavřený tah s nenulovým součtem ohodnocení.
   Pokud ano, vypíše ho a vrátí true, jinak vrátí false. */
static bool nenulovy_tah(const list<int> &k)
{
  for (int v : k)
    vrcholy[v].akt_komponenta = true;

  int v0 = k.front();
  najdi_P_v(v0);
  smaz_znacky(k);
  najdi_Z_v(v0);
  smaz_znacky(k);

  for (int v : k)
    if (vrcholy[v].s_v + vrcholy[v].sp_v != 0)
```

```
      {
        vypis_P_v(v);
        vypis_Z_v(v);
        printf("\n");
        return true;
      }

  for (int v : k)
    for (hrana *e : vrcholy[v].vychazi)
      {
        int u = e->kam;
        if (vrcholy[u].akt_komponenta
            && vrcholy[v].s_v + e->delta + vrcholy[u].sp_v != 0)
          {
            vypis_P_v(v);
            printf("%d ", e->id);
            vypis_Z_v(u);
            printf("\n");
            return true;
          }
      }

  for (int v : k)
    vrcholy[v].akt_komponenta = false;

  return false;
}

/* Provede prohledání do hloubky a vrátí vrcholy v opačném pořadí dle
   konců jejich průchodů. */
static void poradi_dle_prohledani(int v, list<int> &poradi)
{
  if (vrcholy[v].navstiveny)
    return;
  vrcholy[v].navstiveny = true;

  for (hrana *e : vrcholy[v].vychazi)
    poradi_dle_prohledani(e->kam, poradi);

  poradi.push_front(v);
}

static void poradi_dle_prohledani(list<int> &poradi)
{
  int n = vrcholy.size();
  for (int v = 0; v < n; v++)
    poradi_dle_prohledani(v, poradi);
  for (int v = 0; v < n; v++)
    vrcholy[v].navstiveny = false;
}

/* Najde komponentu silné souvislosti obsahující v. */
static void najdi_komponentu(int v, list<int> &k)
{
  if (vrcholy[v].zpracovany)
    return;

  k.push_back(v);
  vrcholy[v].zpracovany = true;
```

```
  for (hrana *e : vrcholy[v].vstupuji)
    najdi_komponentu(e->odkud, k);
}

/* Pro každou komponentu silné souvislosti spustí hledání tahu
   s nenulovým součtem ohodnocení. */
static bool nenulovy_tah(void)
{
  list<int> poradi;
  poradi_dle_prohledani(poradi);

  for (int v : poradi)
    if (!vrcholy[v].zpracovany)
      {
        list<int> k;
        najdi_komponentu(v, k);
        if (nenulovy_tah(k))
          return true;
      }

  return false;
}

int main(void)
{
  int n, m;
  scanf("%d%d", &n, &m);
  vrcholy.resize(n);
  hrany.resize(m);

  for (int id = 1; id <= m; ++id)
    {
      int f, t, a, b;
      scanf("%d%d%d%d", &f, &t, &a, &b);
      f--; t--;

      hrana *e = &hrany[id - 1];
      e->id = id;
      e->odkud = f;
      e->kam = t;
      e->delta = a - b;

      vrcholy[f].vychazi.push_back(e);
      vrcholy[t].vstupuji.push_back(e);
    }

  if (!nenulovy_tah())
    printf("0\n");
  return 0;
}
```

P-I-3 Zahrádka

Jako jednoduché řešení se nabízí projít všechny trojice bodů, pro každou z nich spočítat obsah jimi tvořeného trojúhelníka a vypsat ten s nejmenším obsahem. Počet trojic, a tedy i časová složitost takového algoritmu, je $\mathcal{O}(n^3)$.

Můžeme zkusit postupovat o něco sofistikovaněji. Procházejme pouze dvojice

bodů A a B, tvořících hranu AB hledaného trojúhelníka. Máme-li zafixovanou hranu, obsah trojúhelníka závisí pouze na jeho výšce; mezi zbylými body tedy stačí najít bod C, který je nejblíž k přímce AB. Kdybychom to ale dělali průchodem přes všechny body, nepomůžeme si, časová složitost by opět byla $\mathcal{O}(n^3)$. Můžeme ale udělat užitečné pozorování. Představme si přímku p kolmou na AB a podívejme se na kolmé průměty všech bodů na p; tyto průměty si označme b_1, b_2, … v pořadí postupně na p. Body A i B se promítají na ten samý bod b_i. Průmět bodu C je k němu nejblíž ze všech ostatních průmětů, je tedy roven b_{i-1} nebo b_{i+1}.

To nám naznačuje následující postup. Budeme postupně otáčet přímkou p a udržovat si kolmé průměty bodů na tuto přímku. Kdykoliv se dva body A a B promítnou na ten samý bod b (tedy přímka p je kolmá na úsečku AB), podíváme se na oba body C, jejichž průměty jsou hned před či za b na p, a spočítáme si obsah trojúhelníka ABC. Poté, co se přímkou p otočíme o 180°, máme zaručeno, že v nějakém okamžiku p byla rovnoběžná s hranou optimálního trojúhelníka, a proto jsme započítali jeho obsah. Navíc pro každou dvojici bodů takto počítáme obsah nejvýše dvou trojúhelníků, dohromady tedy počítáme jen $\mathcal{O}(n^2)$ obsahů.

Kdybychom ale skutečně počítali v každém okamžiku přesné souřadnice průmětů na p, postup by byl příliš pomalý; každý takový přepočet zabere lineární čas a časová složitost by se tím zhoršila na alespoň kubickou. Povšimněme si ale, že si stačí udržovat pořadí průmětů bodů na p. Toto pořadí se mění podstatně méně: pouze ve chvíli, kdy je přímka p kolmá na nějakou úsečku AB, se pořadí průmětů bodů A a B na p prohodí.

Tím dostáváme výsledné řešení. Nejprve si spočtěme směrnice přímek mezi každými dvěma body ve vstupu a dvojice bodů si setřiďme podle těchto směrnic. Dále si určeme pořadí dle průmětů na vodorovnou přímku p. Pak postupně procházejme dvojice bodů A a B dle rostoucích směrnic přímek AB; pro každou dvojici spočtěme obsah trojúhelníka tvořeného A, B a předchozím či následujícím bodem v pořadí průmětů, a pak prohoďme pořadí bodů A a B v průmětu. Časová složitost tohoto postupu je $\mathcal{O}(n^2 \log n)$, nejpomalejší částí je setřízení dvojic bodů dle směrnice. Paměťová složitost je $\mathcal{O}(n^2)$.

Paměťovou složitost jde ještě vylepšit: Nemusíme si počítat směrnice všech přímek předem, stačí je mít určené pro všechny dvojice bodů, jejichž průměty aktuálně sousedí. Z těchto směrnic potřebujeme umět vybrat tu nejmenší, budeme si je proto udržovat v haldě. Časová složitost zůstává stejná, ale paměťová se zlepší na $\mathcal{O}(n)$.

```
#include <cstdio>
#include <climits>
#include <list>
#include <vector>
#include <algorithm>
using namespace std;

struct bod
{
  // souřadnice
  int x, y;
```

```
  // body nalevo a napravo v průmětu na aktuální přímku
  bod *vlevo, *vpravo;

  /* místo v haldě reprezentující směrnici přímky procházející tímto
     bodem a bodem napravo od něj v průmětu  */
  int v_halde;

  // zda na začátku leží průmět aktuálního bodu nalevo od bodu NEZ
  bool operator<(const bod &nez)
    {
      if (x < nez.x)
        return true;

      if (x > nez.x)
        return false;

      return y > nez.y;
    }
};

/* Zda je směrnice přímky (a, a->vpravo) menší než směrnice přímky (b, b->vpravo) */
static bool drivejsi(const bod *a, const bod *b)
{
  int ax = a->vpravo->x - a->x;
  int ay = a->vpravo->y - a->y;
  int bx = b->vpravo->x - b->x;
  int by = b->vpravo->y - b->y;

  return ax * by > bx * ay;
}

// vstup
static vector<bod> body;

// halda směrnic přímek
static vector<bod *> halda;

// přesune směrnici přímky (b, b->vpravo) na pozici p v haldě
static void presun(bod *b, int p)
{
  halda[p] = b;
  b->v_halde = p;
}

/* oprava haldy probubláním prvku na pozici p nahoru, je-li menší
   než jeho otec  */
static bool probublej_nahoru(int &p)
{
  if (p == 0)
    return false;

  int otec = (p + 1) / 2 - 1;
  bod *bp = halda[p];
  bod *botec = halda[otec];
  if (drivejsi(botec, bp))
    return false;

  presun(botec, p);
  presun(bp, otec);
  p = otec;
  return true;
```

```
}

/* oprava haldy probubláním prvku na pozici p dolů, je-li větší než
   některý z jeho synů  */
static bool probublej_dolu(int p)
{
  int s1 = 2 * p + 1;
  int s2 = 2 * p + 2;
  int n = halda.size();

  if (s1 >= n)
    return false;

  bod *bp = halda[p];
  bod *bs = halda[s1];
  int s = s1;

  if (s2 < n && drivejsi(halda[s2], bs))
    {
      bs = halda[s2];
      s = s2;
    }

  if (drivejsi(bs, bp))
    {
      presun(bp, s);
      presun(bs, p);
      p = s;
      return true;
    }

  return false;
}

// přidá směrnici směrnici přímky (b, b.vpravo) do haldy
static void pridej_do_haldy(bod &b)
{
  int p = halda.size();
  b.v_halde = p;
  halda.push_back(&b);

  while (probublej_nahoru(p))
    continue;
}

// odebere směrnici směrnici přímky (b, b.vpravo) z haldy
static void odeber_z_haldy(bod &b)
{
  int p = b.v_halde;
  if (p < 0)
    return;
  b.v_halde = -1;

  bod *posl = halda.back();
  halda.pop_back();
  if (p >= (int) halda.size())
   return;

  presun(posl, p);
```

```
  while (probublej_dolu(p))
    continue;
}

// vrátí dvojnásobek obsahu trojúhelníka s vrcholy a, b, c
static int obsah(bod *a, bod *b, bod *c)
{
  int ax = a->x - c->x;
  int ay = a->y - c->y;
  int bx = b->x - c->x;
  int by = b->y - c->y;

  return abs(ax * by - bx * ay);
}

int main(void)
{
  int n;
  scanf("%d", &n);

  body.resize(n);
  for (int i = 0; i < n; i++)
    scanf("%d%d", &body[i].x, &body[i].y);

  // setřídíme body dle jejich x-ové souřadnice
  sort(body.begin(), body.end());
  body[0].vlevo = 0;
  for (int i = 0; i < n - 1; i++)
    {
      body[i].vpravo = &body[i + 1];
      body[i + 1].vlevo = &body[i];
    }
  body[n - 1].vpravo = 0;
  body[n - 1].v_halde = -1;

  // naninicalizujeme haldu směrnicemi bodů, jejichž průměty sousedí
  for (int i = 0; i < n - 1; i++)
    pridej_do_haldy(body[i]);

  bod *sa, *sb, *sc;
  int min_obsah = INT_MAX;

  while (!halda.empty())
    {
      // nalezneme sousední body a a b v průmětu, jejichž směrnice je minimální
      bod *a = halda.front();
      bod *b = a->vpravo;

      /* započítáme obsahy trojúhelníků s hranou ab a vrcholem c, který s nimi
         sousedí v průmětu */
      bod *c1 = a->vlevo;
      bod *c2 = b->vpravo;

      if (c1)
        {
          int o1 = obsah(a, b, c1);
          if (o1 < min_obsah)
            {
              min_obsah = o1;
              sa = a; sb = b; sc = c1;
```

```
            }
          }

        if (c2)
          {
            int o2 = obsah(a, b, c2);
            if (o2 < min_obsah)
              {
                min_obsah = o2;
                sa = a; sb = b; sc = c2;
              }
          }

        // z haldy odebereme směrnice přímek procházejících body a či b
        odeber_z_haldy(*a);
        odeber_z_haldy(*b);
        if (c1)
          odeber_z_haldy(*c1);

        // prohodíme pořadí průmětů bodů a a b
        a->vpravo = c2;
        if (c2)
          c2->vlevo = a;
        a->vlevo = b;
        b->vpravo = a;
        b->vlevo = c1;
        if (c1)
          c1->vlevo = b;

        /* do haldy přidáme směrnice přímek procházejících a či b a body,
           které s nimi sousedí v průmětu.  */
        if (c1 && c1 < b)
          pridej_do_haldy(*c1);
        if (c2 && a < c2)
          pridej_do_haldy(*a);
      }

    printf("%d %d; %d %d; %d %d\n", sa->x, sa->y, sb->x, sb->y, sc->x, sc->y);
    return 0;
}
```

P-I-4 Dva lupiči

a) Nechť C je celková cena věcí v pytli. Zjevně v libovolném (a tedy i optimálním) řešení hromádka alespoň jednoho z lupičů bude mít cenu alespoň $C/2$. Oproti tomu v řešení, které dá vše prvnímu lupiči, bude mít dražší hromádka cenu C, tedy pouze dvakrát větší. Toto řešení je tedy 2-kompetitivní.

b) Zkusme přirozený algoritmus, který nově vytaženou věc vždy přidělí tomu lupiči, jehož hromádka má aktuálně nižší cenu (mají-li oba stejně, pak libovolnému z nich). Nechť na konci algoritmu mají hromádky ceny C_1 a C_2, kde zjevně $C = C_1 + C_2$. Bez újmy na obecnosti můžeme předpokládat $C_1 \leq C_2$ (jinak lupiče prohodíme), hodnota nalezeného řešení je tedy C_2.

Nechť M je cena nejdražšího předmětu. Rozmysleme si, že v průběhu rozdělování dle přirozeného algoritmu se ceny hromádek vždy liší nejvýše o M: Zjevně

je to pravda na začátku, kdy mají obě nulovou cenu. V průběhu algoritmu vždy přidáváme předmět nějaké ceny c na levnější hromádku. Pokud tato hromádka zůstává levnější, rozdíl cen hromádek se tím jen zmenší. Pokud se přidáním předmětu hromádka stane dražší než druhá, rozdíl cen výsledných hromádek je nejvýše $c \leq M$. Speciálně, na konci běhu algoritmu platí $C_2 \leq C_1 + M$.

Nechť OPT je cena dražší hromádky v optimálním řešení. Jak jsme vypozorovali v minulé podúloze, $C/2 \leq \mathrm{OPT}$, a tedy $C \leq 2 \cdot \mathrm{OPT}$. Optimální řešení musí také někomu přidělit nejdražší předmět, a proto $M \leq \mathrm{OPT}$.

Nyní stačí tyto nerovnosti zkombinovat. Máme

$$2C_2 \leq C_2 + C_1 + M = C + M \leq 3 \cdot \mathrm{OPT},$$

a tedy $C_2 \leq \frac{3}{2} \cdot \mathrm{OPT}$. Náš algoritmus je tedy (3/2)-kompetitivní.

c) Uvažme chování libovolného on-line algoritmu na vstupech (cenách věcí postupně vytahovaných z pytle) 1 1 a 1 1 2. Tyto vstupy se shodují v prvních dvou prvcích, algoritmus se tedy na nich zachová stejně. První předmět bez újmy na obecnosti přidělí prvnímu lupiči. Pokud druhý předmět také dá prvnímu lupiči, pak dražší hromádka na konci prvního vstupu bude mít cenu 2, optimální řešení by ale dalo každému lupiči 1; uvažovaný algoritmus by tedy nemohl být lepší než 2-kompetitivní.

Řekněme tedy, že algoritmus druhý předmět dá druhému lupiči. Ve druhém ze vstupů se pak musí rozhodnout, komu dát poslední předmět ceny 2; ať už se rozhodne jakkoliv, cena dražší hromádky bude 3. Optimální algoritmus, který zná celý vstup dopředu, ale může dát oba předměty ceny 1 prvnímu lupiči a předmět ceny 2 druhému, a jeho dražší hromádka bude tedy mít cenu 2. Uvažovaný algoritmus tedy nemůže být lepší než 3/2-kompetitivní.