
P-II-1 Tulipány

Budeme řešit o něco obecnější úlohu: dovolíme si předepsat, zda má na n-té pozici být tulipán, a pokud ano, zda skupina končící na n-té pozici má mít sudou či lichou délku (všechny ostatní skupiny musí mít délku lichou); do zadání si tedy přidáme ještě jeden parametr p, který může nabývat hodnot *prázdný*, *sudá*, či *lichá*.

Řekněme, že bychom uměli úlohu vyřešit pro prvních $n-1$ čtverečků a každou hodnotu parametru p; nejmenší počty tulipánů v jednotlivých případech označíme $t(n-1, \textit{prázdný})$, $t(n-1, \textit{sudá})$ a $t(n-1, \textit{lichá})$ (v některých z těchto případů nemusí existovat řešení, v tom případě je odpovídající počet tulipánů ∞). Rozmysleme si nyní, jak určit počet tulipánů $t(n, p)$ pro celý vstup.

Jestliže na n-tém čtverečku už je tulipán, nemůže toto políčko být prázdné a

$$t(n, \textit{prázdný}) = \infty.$$

Skupina končící na n-tém čtverečku bude mít lichou délku, jestliže skupina končící na čtverečku $n-1$ má délku sudou nebo jestliže na čtverečku $n-1$ není tulipán, a bude mít sudou délku, jestliže skupina končící na čtverečku $n-1$ má délku lichou. Proto dostáváme

$$\begin{aligned} t(n, \textit{lichá}) &= \min(t(n-1, \textit{prázdný}), t(n-1, \textit{sudá})), \\ t(n, \textit{sudá}) &= t(n-1, \textit{lichá}). \end{aligned}$$

Jestliže je n-tý čtvereček prázdný, pak můžeme na něj vysadit tulipán (a změnit tak paritu poslední skupiny), nebo ho nechat prázdný (což je ale možné pouze pokud čtvereček $n-1$ je prázdný nebo jím končí lichá skupina). Dostáváme tedy následující vztahy:

$$\begin{aligned} t(n, \textit{prázdný}) &= \min(t(n-1, \textit{prázdný}), t(n-1, \textit{lichá})), \\ t(n, \textit{lichá}) &= 1 + \min(t(n-1, \textit{prázdný}), t(n-1, \textit{sudá})), \\ t(n, \textit{sudá}) &= 1 + t(n-1, \textit{lichá}). \end{aligned}$$

Hodnoty $t(n-1, p)$ můžeme obdobně spočítat z hodnot $t(n-2, p)$, atd. Stačí si tedy tyto hodnoty spočítat postupně od $t(1, p)$ do $t(n, p)$ a vrátit minimum z $t(n, \textit{prázdný})$ a $t(n, \textit{lichá})$. Vzhledem k tomu, že si vždy stačí pamatovat tři čísla z předchozího kroku, paměťová složitost je konstantní. Program provede pouze jeden průchod nad vstupem, časová složitost je tedy lineární, $\mathcal{O}(n)$.

```
#include <cstdio>
#include <algorithm>
using namespace std;

#define INF 1000000

int main (void)
{
  int min_prazdny = 0, min_suda = INF, min_licha = INF;
  char a;

  while (scanf("%c", &a) == 1 && (a == 'T' || a == 'P'))
    {
      int prazdny, suda, licha;

      if (a == 'T')
        {
          prazdny = INF;
          licha = min(min_prazdny, min_suda);
          suda = min_licha;
        }
      else
        {
          prazdny = min(min_prazdny, min_licha);
          licha = 1 + min(min_prazdny, min_suda);
          suda = 1 + min_licha;
        }
      min_prazdny = prazdny;
      min_suda = suda;
      min_licha = licha;
    }

  printf("%d\n", min(min_prazdny, min_licha));
  return 0;
}
```

P-II-2 Ohrada

Nejprve si rozmysleme, že pro $n = 5$ má úloha vždy řešení. To je zjevné, jestliže všech 5 bodů leží na konvexním obalu (pak lze vybrat libovolnou čtveřici bodů). Jestliže na konvexním obalu leží 4 body $p_1, \ldots, p_4$, pak pátý bod p_5 leží buď uvnitř trojúhelníka $p_1p_2p_3$, nebo uvnitř trojúhelníka $p_1p_4p_3$; v prvním případě vrátíme čtveřici $p_1p_4p_3p_5$, ve druhém čtveřici $p_1p_2p_3p_5$. Poslední možnost je, že na konvexním obalu leží jen tři body p_1, p_2 a p_3. Bez újmy na obecnosti můžeme předpokládat, že bod p_5 leží uvnitř trojúhelníka $p_1p_2p_4$ (ostatní případy, kdy leží uvnitř trojúhelníka $p_2p_3p_4$ či $p_1p_3p_4$, jsou symetrické). Bez újmy na obecnosti také můžeme předpokládat, že p_5 leží ve stejné polorovině přímky p_3p_4 jako p_2 (případ, kdy leží ve stejné polorovině jako p_1, je symetrický). Pak můžeme vrátit čtveřici $p_2p_3p_4p_5$.

Pro $n \geq 6$ stačí aplikovat výše popsaný postup pro pět ze zadaných bodů s nejmenší x-ovou souřadnicí a také vždy dostaneme řešení. Nalezení těchto pěti bodů nám zabere lineární čas a konstantní paměť, rozbor případů z minulého odstavce provedeme v konstantním čase (nebo můžeme prostě vyzkoušet všechny čtveřice z těchto pěti bodů). Časová složitost je tedy $\mathcal{O}(n)$, paměťová $\mathcal{O}(1)$.

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

struct bod
{
  int x, y;

  bod(int _x, int _y) : x(_x), y(_y) { }

  bod operator- (const bod &b) const { return bod(x - b.x, y - b.y); }

  /* 2D vektorový součin. */
  int operator* (const bod &b) const { return x * b.y - y * b.x; }
};

static int sgn(int x)
{
  return (x > 0) - (x < 0);
}

/* Vrací, zda je vektor B nalevo či napravo od polopřímky dané vektorem A. */
static int orientace(const bod &a, const bod &b)
{
  return sgn(a * b);
}

/* Otestuje, zda bod U je uvnitř úhlu daného vektory A a B. */
static bool uvnitr_uhlu(const bod &a, const bod &b, const bod &u)
{
  int or_ab = orientace(a, b);

  return (or_ab == orientace(a, u) && or_ab == -orientace(b, u));
}
```

```
/* Otestuje, zda bod U je uvnitř trojúhelníka ABC. */
static bool uvnitr(const bod &a, const bod &b, const bod &c, const bod &u)
{
  return (uvnitr_uhlu(b - a, c - a, u - a) && uvnitr_uhlu(a - b, c - b, u - b));
}

/* Otestuje, zda je bod U uvnitř konvexního čtyřúhelníka s vrcholy C. */
static bool uvnitr(const vector<bod> &c, const bod &u)
{
  assert(c.size() == 4);
  /* Stačí testovat tři trojice vrcholů -- sjednocení libovolných
     tří trojúhelníků pokryje celý čtyřúhelník. */
  for (int i = 0; i < 3; i++)
    if (uvnitr(c[i], c[(i+1) % 4], c[(i+2) % 4], u))
      return true;

  return false;
}

/* Otestuje, zda jsou body v mn v konvexní poloze. */
static bool konvexni(const vector<bod> &mn)
{
  assert(mn.size() == 4);

  for (int i = 0; i < 4; i++)
    if (uvnitr(mn[i], mn[(i+1) % 4], mn[(i+2) % 4], mn[(i+3) % 4]))
      return false;

  return true;
}

static bool mensi_x_souradnice(const bod &b1, const bod &b2)
{
  return b1.x < b2.x;
}

static void vypis(const vector<bod> &hranice)
{
  for (const bod &b : hranice)
    printf("%d %d\n", b.x, b.y);
}

int main(void)
{
  int n;
  vector<bod> stromy;

  scanf("%d", &n);

  for (int i = 0; i < n; i++)
    {
      int x, y;

      scanf("%d%d", &x, &y);
      if (i < 5)
        stromy.emplace_back(x, y);
      else
        {
          vector<bod>::iterator m = max_element(stromy.begin(), stromy.end(),
                                                mensi_x_souradnice);
```

```
          if (x < m->x)
            {
              m->x = x;
              m->y = y;
            }
        }
    }

  if (n == 4)
    {
      if (konvexni(stromy))
        vypis(stromy);
      else
        printf("nelze\n");
      return 0;
    }

  assert(stromy.size() == 5);
  for (int i = 0; i < 5; i++)
    {
      vector<bod> bez_iteho;
      for (int j = 0; j < 5; j++)
        if (j != i)
          bez_iteho.emplace_back(stromy[j]);
      if (konvexni(bez_iteho) && !uvnitr(bez_iteho, stromy[i]))
        {
          vypis(bez_iteho);
          return 0;
        }
    }

  abort();  // Sem se program nemůže dostat.
}
```

P-II-3 Úřady

Nechť Z je množina měst, do nichž nevchází žádný kanál. Když začneme v libovolném městě a půjdeme proti směru kanálů tak dlouho, jak to půjde, nutně skončíme v Z (nemůžeme se zacyklit, jelikož se vždy přesouváme do města s vyšší nadmořskou výškou). To znamená, že z měst v množině Z se dá po proudu dostat do libovolného jiného města. Jestliže tedy mezi městy v Z nevedou cesty, stačí vypsat množinu Z.

Můžeme proto předpokládat, že mezi městy $a, b \in Z$ vede cesta. Řekněme, že bychom smazali město a (a všechny sousedící cesty a kanály) a v takto modifikované síti našli nějaké řešení U. Tvrdíme, že U je řešením i pro původní síť: Jelikož do b nevstupuje žádný kanál, do b se musí dát dostat z U po cestách, a přidáním cesty ba dostáváme, že se z U dá dostat po cestách i do a.

Řešení tedy vždy existuje a nalezneme ho opakováním výše popsaného postupu (který se zastaví nejpozději ve chvíli, kdy zbývá už jen jedno město). Jak ho implementovat efektivně? Budeme si pamatovat seznam cest spojujících města, z nichž nevychází žádný kanál. Vždy když smažeme město a, projdeme si všechna města do nichž vede z a kanál, a ověříme, zda do nich nyní žádný kanál nevede. Pokud ano, projdeme všechny sousední cesty, a pokud do jejich druhého konce také nevede kanál, přidáme je do seznamu. Během celého výpočtu takto každou cestu ověřujeme nejvýše dvakrát (jednou pro každý její konec), a tak nám to celkově zabere lineární čas. Ostatní operace (mazání vrcholů a incidentních hran) nám také zaberou pouze lineární čas. Výsledná časová i paměťová složitost jsou tudíž lineární $\mathcal{O}(n + k + c)$.

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

struct vrchol
{
  bool smazany;
  int pocet_vchazejicich_kanalu;
  list<int> vychazejici_kanaly;
  list<int> cesty;

  vrchol(void) : smazany(false), pocet_vchazejicich_kanalu(0) { }

  bool zdroj(void) { return pocet_vchazejicich_kanalu == 0; }
};

static vector<vrchol> sit;
static list<pair<int,int> > cesty_mezi_zdroji;

/* Přidá do seznamu cesty mezi nově vzniklým zdrojem a ostatními
   ještě nesmazanými zdroji. */
static void aktualizuj_cesty_mezi_zdroji(int novy_zdroj)
{
  for (int a : sit[novy_zdroj].cesty)
    if (!sit[a].smazany && sit[a].zdroj())
      cesty_mezi_zdroji.emplace_back(novy_zdroj, a);
}
```

```
int main(void)
{
  int n, k, c;
  scanf("%d%d%d", &n, &k, &c);
  sit.resize(n);

  for (int i = 0; i < k; i++)
    {
      int d, z;
      scanf("%d%d", &d, &z);
      d--; z--;
      sit[z].vychazejici_kanaly.push_back(d);
      sit[d].pocet_vchazejicich_kanalu++;
    }
  for (int i = 0; i < c; i++)
    {
      int a, b;
      scanf("%d%d", &a, &b);
      a--; b--;
      sit[a].cesty.push_back(b);
      sit[b].cesty.push_back(a);
      if (sit[a].zdroj() && sit[b].zdroj())
        cesty_mezi_zdroji.emplace_back(a, b);
    }

  while (!cesty_mezi_zdroji.empty())
    {
      pair<int,int> cesta = cesty_mezi_zdroji.back();
      cesty_mezi_zdroji.pop_back();

      int a = cesta.first;
      int b = cesta.second;

      if (sit[a].smazany || sit[b].smazany)
        continue;

      sit[a].smazany = true;
      for (int d : sit[a].vychazejici_kanaly)
        {
          assert(!sit[d].smazany);
          sit[d].pocet_vchazejicich_kanalu--;
        }

      for (int d : sit[a].vychazejici_kanaly)
        if (sit[d].zdroj())
          aktualizuj_cesty_mezi_zdroji(d);
    }

  for (int i = 0; i < n; i++)
    if (!sit[i].smazany && sit[i].pocet_vchazejicich_kanalu == 0)
      printf("%d ", i + 1);
  printf("\n");

  return 0;
}
```

P-II-4 Doprava

Zjevně jediné rozhodnutí, které můžete udělat, je zda a po kolika dnech si slevovou kartu koupíte. Nechť S_d označuje algoritmus „zakoupit slevovou kartu po d dnech“ a N algoritmus „slevovou kartu si nekoupím“.

Oproti tomu optimální řešení si slevovou kartu může koupit pouze na začátku (nemá smysl před její koupí ještě nějaký čas jezdit dráže). Vyhodí-li vás tedy za n dní, cena optimálního řešení je

$$\min(2n, a+n) = \begin{cases} 2n & \text{jestliže } n \leq a, \\ a+n & \text{jestliže } a \leq n. \end{cases}$$

Uvažujme nejprve srovnání s algoritmem S_d. Každý den před dnem d včetně se poměr mezi tím, kolik zaplatí S_d a optimální řešení zvětšuje nebo zůstává stejný. Po zakoupení slevové karty se poměr zvýší, dále už se bude jen snižovat. Nejhorší případ pro řešení S_d tedy je, že vás vyhodí přesně po d-tém dni, kdy utratíte $2d+a$ korun. Kompetitivita vůči optimálnímu řešení tedy je

- $\frac{2d+a}{2d} = 1 + \frac{a}{2d}$, jestliže $d \leq a$;
- $\frac{2d+a}{a+d} = 2 - \frac{a}{a+d}$, jestliže $a \leq d$.

V prvním případě je nejlepší mít d co největší, ve druhém co nejmenší. Nejlepší kompetitiní poměr tedy má algoritmus S_a, který je 3/2-kompetitivní.

Algoritmus N má po $n \geq a$ dnech poměr $\frac{2n}{a+n} = 2 - \frac{2a}{a+n}$, což je pro velké n libovolně blízko 2; je tedy horší než S_a. Žádný algoritmus lepší než 3/2-kompetitivní tedy neexistuje.