P-III-4 Výlet

Abychom našli požadovaný výlet délky 4, stačí najít dvě hvězdy a a b, které mají dva společné sousedy v a w (hvězdy, do nichž vedou červí díry z a i z b); pak $a\ v\ b\ w$ je výlet délky 4.

Proto si postavíme dvourozměrné pole `soused`, v němž si postupně budeme pro každou dvojici hvězd určovat číslo nějaké hvězdy, která je s oběma spojena červími děrami (existuje-ji). To uděláme tak, že pro každou hvězdu v a pro každou dvojici hvězd a a b, do nichž vede červí díra z v, si do `soused`$[a][b]$ poznačíme v, pokud na této pozici zatím není nic uloženo. Pokud na pozici `soused`$[a][b]$ už je uloženo w, pak vypíšeme $a\ v\ b\ w$ a skončíme. Naopak, pokud tímto postupem projdeme každý vrchol a všechny dvojice jeho sousedů a vždy ukládáme na prázdné pozice v poli `soused`, pak úloha zjevně nemá řešení.

Jaká je časová složitost tohoto postupu? Na první pohled by se mohlo zdát, že je až $\mathcal{O}(n^3)$—procházíme n vrcholů a pro každý z nich procházíme dvojice z až $n-1$ sousedů. Nicméně si povšimněme, že do každého prvku pole `soused` budeme během celého postupu zapisovat nejvýše jednou, celkový počet zápisů je tedy nejvýše n^2. Jelikož v každém kroku popsaného postupu (kromě posledního) zapisujeme do pole `soused`, počet těchto kroků je tedy také nejvýše n^2, a proto je výsledná časová složitost $\mathcal{O}(n^2)$. Paměťová složitost je dominována velikostí pole `soused` a je tedy také $\mathcal{O}(n^2)$.

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

#define MAXN 10000
static list<int> cervi[MAXN];
static vector<int> soused[MAXN];

static int zaznamenej_souseda(int a, int b, int v)
{
  if (a > b)
    swap(a, b);

  int ret = soused[b][a];
  soused[b][a] = v;
  return ret;
}

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

  for (int i = 0; i < m; i++)
```

```
    {
      int a, b;
      scanf("%d%d", &a, &b);
      a--; b--;
      cervi[a].push_back(b);
      cervi[b].push_back(a);
    }

  for (int i = 0; i < n; i++)
    soused[i].resize(i, -1);

  for (int v = 0; v < n; v++)
    {
      list<int> &s = cervi[v];
      for (list<int>::iterator a = s.begin(); a != s.end(); ++a)
        {
          list<int>::iterator b = a;
          ++b;
          for (; b != s.end(); ++b)
            {
              int c = zaznamenej_souseda(*a, *b, v);
              if (c >= 0)
                {
                  printf("%d %d %d %d\n", *a + 1, c + 1, *b + 1, v + 1);
                  return 0;
                }
            }
        }
    }

  printf("0\n");
  return 0;
}
```

P-III-5 Důl

Nejprve si rozmysleme řešení jednodušší úlohy, kdy nemáme žádné omezení na délku pobytů horníků. V tom případě je úloha ekvivalentní určení počtu dobrých uzávorkování z n otevíracích a n zavíracích závorek (otevírací závorky odpovídají příchodům horníků, zavírací odchodům). Dobré uzávorkování vypadá tak, že na začátku je otevírací závorka, pak následuje dobré uzávorkování z i otevíracích a zavíracích závorek pro nějaké i tž. $0 \leq i \leq n-1$, pak zavírací závorka odpovídající první otevírací, a nakonec dobré uzávorkování z $n-i-1$ otevíracích a zavíracích závorek. Jelikož každá taková kombinace odpovídá právě jednomu dobrému uzávorkování, dostáváme pro počet z_n uzávorkování z n otevíracích a zavíracích závorek následující vzorec platný pro každé $n \geq 1$ (máme $z_0 = 1$):

$$z_n = \sum_{i=0}^{n-1} z_i z_{n-i-1}.$$

Můžeme tedy postupně počítat $z_1, \ldots, z_n$, pro určení každého z nich sčítáme nejvýše n členů předchozí sumy, zabere nám to tedy čas $\mathcal{O}(n^2)$.

Vraťme se nyní k úloze. Oproti předchozímu případu se mění následující: první horník může pracovat pouze m minut, a tedy musí platit $a_{2i+2} \leq a_1 + m$. Horníci, kteří přidou dřív, než první horník odejde, také musí odejít před časem a_{2i+2}, a proto pro ně doba v práci nepředstavuje žádné další omezení; můžeme si tedy pro ně vybrat libovolné dobré uzávorkování z i otevíracích a zavíracích závorek. Oproti tomu možná uzávorkování pro $n - i - 1$ zbývajících horníků mohou být dobou práce omezeny. Označme si proto jako u_i počet způsobů, jak dobře uzávorkovat posledních i horníků tak, aby pro ně platilo omezení na pracovní dobu. Dle výše popsané úvahy máme

$$u_n = \sum_{\substack{1 \leq i \leq n-1 \\ a_{2i+2} \leq a_1+m}} z_i u_{n-i-1},$$

a obdobně

$$u_m = \sum_{\substack{1 \leq i \leq m-1 \\ a_{2(n-m+i)+2} \leq a_{2(n-m)+1}+m}} z_i u_{m-i-1}$$

pro $1 \leq m \leq n$ (kde $u_0 = 1$). Můžeme si tedy v čase $\mathcal{O}(n^2)$ spočítat u_1, …, u_n dle tohoto vzorce a vypsat u_m. Všechny výpočty samozřejmě provádíme modulo 1 000 003. Paměťová složitost je lineární, pamatujeme si vstup a pole obsahující hodnoty z_i a u_i.

```
#include <cstdio>
#include <vector>
using namespace std;
#define MOD 1000003

static vector<int> zavorky;
static vector<int> casy;
static vector<int> suffixy;

static void predpocitej_zavorkovani(int n)
{
  zavorky.resize(n + 1);
  zavorky[0] = 1;
  for (int z = 1; z <= n; z++)
    {
      long long s = 0;

      for (int a = 0; a < z; a++)
        s += (long long) zavorky[a] * zavorky[z - a - 1];

      zavorky[z] = s % MOD;
    }
}

int main(void)
{
  int n, m;
  scanf("%d%d", &n, &m);
  casy.resize(2 * n);
  for (int i = 0; i < 2 * n; i++)
    scanf("%d", &casy[i]);
```

```
  predpocitej_zavorkovani(n);
  suffixy.resize(n + 1);
  suffixy[n] = 1;

  for (int p = n-1; p >= 0; p--)
    {
      long long s = 0;
      int cp = casy[2 * p];
      for (int a = 0; a + p < n; a++)
        {
          if (casy[2*(p+a) + 1] > cp + m)
            break;
          s += (long long) zavorky[a] * suffixy[p+a+1];
        }

      suffixy[p] = s % MOD;
    }

  printf("%d\n", suffixy[0]);
  return 0;
}
```

P-III-6 Wienerův index

Vzdálenost mezi atomy a a b označme jako $d(a, b)$. Řešme zobecněnou úlohu: Každému atomu a přiřaďme jako váhu přirozené číslo w_a, a počítejme součet $w_a w_b d(a, b)$ přes všechny dvojice atomů a a b. Původní úloha odpovídá případu, kdy všechny atomy mají váhu 1. Jakožto ω označme součet všech vah.

Podívejme se na poslední atom b na vstupu; ten je spojen vazbou s právě jedním atomem a. Představme si nyní, že váhu atomu a zvýšíme o w_b a atom b smažeme. Jak se tímto změní (zobecněný) Wienerův index? Ze součtu jsme ztratili členy $w_b w_c d(b, c)$ pro všechny atomy $c \neq b$; oproti tomu nám tam (díky zvýšení váhy atomu a) přibyly členy $w_b w_c d(a, c)$. Jelikož všechny cesty do b vedou přes a, máme $d(b, c) = d(a, c) + 1$. Proto jsme popsanou modifikací zmenšili Wienerův index o

$$w_b \sum_{c \neq b} w_c (d(b, c) - d(a, c)) = w_b \sum_{c \neq b} w_c = w_b(\omega - w_b).$$

Určíme-li tedy Wienerův index modifikované molekuly, přičtením členu $w_b(\omega - w_b)$ dostáváme výsledek. Wienerův index modifikované molekuly určíme rekurzivně opakováním tohoto postupu. Jelikož pro každý atom vykonáme jen konstantní množství práce (jeho odebrání, zvýšení váhy sousedního atomu a přičtení $w_b(\omega - w_b)$ k výsledku z rekurze), celková časová složitost je lineární $\mathcal{O}(n)$; stejná je i paměťová složitost.

```
#include <cstdio>
#include <vector>
using namespace std;
#define MOD 1000003

static vector<int> atomy;
static vector<int> vahy;
```

```
int main(void)
{
  int n;
  scanf("%d", &n);
  atomy.resize(n);
  vahy.resize(n, 1);

  for (int i = 1; i < n; i++)
    {
      int s;
      scanf("%d", &s);
      atomy[i] = s - 1;
    }

  for (int b = n - 1; b > 0; b--)
    vahy[atomy[b]] += vahy[b];

  long long s = 0;
  for (int b = 1; b < n; b++)
    s += (long long) vahy[b] * (n - vahy[b]);

  printf("%lld\n", s % MOD);
  return 0;
}
```