66. ročník Matematické olympiády – 2016/2017

Řešení úloh domácího kola kategorie P

P-I-1 Na dostizích

Jako *permutaci* označíme libovolné uspořádání čísel 1 až N, v němž se každé číslo vyskytuje právě jednou. Úloha nám zadá dvě permutace N čísel a chce po nás spočítat počet dvojic čísel (a, b) takových, že se v obou permutacích nachází ve stejném pořadí (takové dvojice budeme nazývat *zajímavé*).

Vyzkoušíme všechny dvojice

Přímočaré řešení by bylo projít všechny dvojice čísel od 1 do N a pro každou dvojici se podívat, zda se v obou permutacích nachází ve stejném pořadí. Pořadí se dá zjistit tak, že pro každou dvojici (a, b) projdeme každou permutaci, zapíšeme si, na které z čísel a, b jsme narazili dříve, a podíváme se, zda jsme si u obou permutací zapsali to samé číslo. Dvojic je $\mathcal{O}(N^2)$ a pro každou z nich uděláme $\mathcal{O}(N)$ práce (projdeme obě permutace), tedy řešení má složitost $\mathcal{O}(N^3)$, což stačilo na 5 bodů.

Zrychlíme testování dvojic

Předchozí řešení můžeme vylepšit, pokud si všimneme, že zbytečně pořád procházíme permutace znovu. Můžeme je projít jednou a pro každou z nich si u každého čísla zapsat, na které pozici se nachází. Když pak chceme zjistit, zda jsou čísla a, b v obou permutacích ve stejném pořadí, máme už pořadí zjištěné, takže ho jednoduše porovnáme. Tak na každou dvojici spotřebujeme jen $\mathcal{O}(1)$ práce, a celé řešení tedy bude mít složitost $\mathcal{O}(N^2)$, čímž oproti předchozímu získáme další dva body.

Lépe než $\mathcal{O}(N^2)$ to už nedokážeme, pokud budeme chtít otestovat každou dvojici zvlášť. Pokud tedy chceme plných 10 bodů, musíme vymyslet jiný přístup.

Co kdyby jedna permutace byla $1, 2, 3, \ldots, N$?

Představme si chvilku, že jedna z permutací jsou čísla seřazená podle velikosti. Pak vlastně chceme zjistit, kolik dvojic čísel je ve druhé permutaci ve správném (tj. podle velikosti) pořadí.

Pokud víte, jak se počítá počet inverzí permutace, můžete přeskočit následující část. (Hledané číslo je rozdíl počtu všech dvojic a počtu inverzí).

Každou dvojici $a < b$ takovou, že a se v permutaci vyskytuje dříve než b, nazveme *dobrou*. Nyní si ukážeme, jak spočítat počet dobrých dvojic. Použijeme k tomu Mergesort:* Představíme si, že chceme Mergesortem setřídit naši permutaci, přičemž během celého algoritmu si budeme udržovat globální čítač dobrých dvojic.

Představme si, že právě sléváme dvě části X a Y. Z předchozích fází Mergesortu víme, že každá z těchto částí je již setříděná a všechny dobré dvojice v rámci X

---

* Popis Mergesortu (a plno dalších zajímavých věcí) můžete najít v této kapitole Kuchařky KSP: *https://ksp.mff.cuni.cz/viz/kucharky/trideni*.

resp. Y jsme již započítali. A také víme, že všechny prvky X byly v původní permutaci před všemi prvky Y. V této fázi chceme slít části X a Y do jedné a započítat všechny dobré dvojice takové, že jeden jejich prvek je z X a druhý z Y. Protože se všechny prvky X vyskytují před všemi prvky Y, tak každý prvek X tvoří dobrou dvojici se všemi prvky Y, které jsou větší. To znamená, že při slévání stačí pokaždé, když zatřiďujeme prvek X, k čítači přičíst aktuální počet ještě nezatříděných prvků Y (protože to jsou právě ty prvky Y, které jsou větší, než právě zatřiďovaný prvek X).

Po doběhnutí tohoto algoritmu budeme mít v čítači počet dobrých dvojic, tedy odpověď na naši otázku. Ke každému kroku Mergesortu jsme přidali jen konstantně mnoho operací, tedy upravený Mergesort a i celý algoritmus poběží v $\mathcal{O}(N \log N)$.

Co když žádná permutace není $1, 2, 3, \ldots, N$?

Tak zkusíme situaci převést na předchozí případ. Mějme permutaci P. Pak jako *inverzní* permutaci (značíme P^{-1}) označíme takovou permutaci, že pokud je na a-té pozici P číslo b, tak na b-té pozici P^{-1} bude číslo a. Například je-li $P = (2314)$, pak $P^{-1} = (3124)$. Všimněte si, že pokud se na permutace díváme jako na pole, tak $P^{-1}[P[i]] = i$.

Na vstupu dostaneme dvě permutace P, Q. Nyní si rozmyslíme, že počet zajímavých dvojic vzhledem k P, Q je stejný jako počet dobrých dvojic v $Q^{-1}[P[\ldots]]$.

Nechť je a, b zajímavá dvojice (bez újmy na obecnosti je a před b). Vezměme takové i, j, že $P[i] = a$ a $P[j] = b$ (a tedy $i < j$). Protože a je před b i v Q, tak to znamená, že ve Q^{-1} bude na a-té pozici menší číslo než na b-té. A to znamená, že v $Q^{-1}[P[i]] = Q^{-1}[a] < Q^{-1}[b] = Q^{-1}[P[j]]$, tedy prvky na i-té resp. j-té pozici v $Q^{-1}[P[\ldots]]$ jsou dobré, pokud jsou prvky na i-té a j-té pozici v P zajímavé vzhledem k P, Q.

Nechť naopak a, b jsou dobré prvky v $Q^{-1}[P[\ldots]]$ (bez újmy na obecnosti můžeme předpokládat, že $a < b$). Opět vezmeme takové i, j, že $Q^{-1}[P[i]] = a$ a $Q^{-1}[P[j]] = b$ (a tedy $i < j$). A označme $P[i] = u$, $P[j] = v$. Víme, že $Q^{-1}[u] = a$ a $Q^{-1}[v] = b$, což z definice Q^{-1} znamená, že $Q[a] = u$ a $Q[b] = v$. A protože $a < b$, tak u leží v Q před v, což je ale stejné pořadí, v jakém jsou v P, tedy prvky na i-té a j-té pozici v P jsou vzhledem k P, Q zajímavé, pokud jsou prvky na i-té a j-té pozici v $Q^{-1}[P[\ldots]]$ dobré.

Ukázali jsme obě dvě implikace, a tedy prvky na i-té a j-té pozici v P jsou vzhledem k P, Q zajímavé právě tehdy, pokud jsou prvky na i-té a j-té pozici v $Q^{-1}[P[\ldots]]$ dobré. To speciálně znamená, že dobrých prvků v $Q^{-1}[P[\ldots]]$ je stejný počet jako zajímavých vzhledem k P, Q.

Níže naleznete program implementující toto řešení.

Řešení pomocí intervalových stromů

Pokud umíte naprogramovat intervalový (stačí Fenwickův) strom,* mohli jste se obejít bez myšlenky *skládání* permutací. Dobré dvojice budeme počítat přes jejich

* I ty najdete v kuchařce KSP: *https://ksp.mff.cuni.cz/viz/kucharky/intervalove-stromy*.

dřívější prvek v P. Podívejme se tedy na první prvek v P a najděme takové i, že $Q[i] = P[1]$. Potom $P[1]$ tvoří dobré dvojice právě s těmi prvky, které jsou ve Q za ním, tedy se všemi $Q[j]$, kde $j > i$. To znamená, že za $P[1]$ započítáme $N - i$ dobrých prvků. S $P[2]$ můžeme udělat stejnou úvahu, akorát pokud se $P[2]$ nachází ve Q před $P[1]$ (tj. $P[2] = Q[k]$, kde $k < i$), tak nesmíme započítat prvek $Q[i]$. V dalším kroku si musíme dát pozor na $Q[i]$ a $Q[k]$. A tak dále.

Naivní přístup, kdy bychom pokaždé prošli všechny již zpracované pozice, by opět trval $\mathcal{O}(N^2)$. Ale to dokážeme zlepšit. Založme si pomocné pole $done[\ldots]$, které bude na začátku nulové, a pokaždé, když zpracujeme nějaký prvek $Q[i]$ (který odpovídal právě zpracovávanému prvku P), tak nastavíme $done[i] = 1$. Potom pokud nyní zpracováváme $Q[k]$, tak počet prvků, které nechceme započítat, je součet všech prvků pole $done[\ldots]$ od pozice k až na konec. Ale intervalové součty s updaty prvků umí intervalové stromy nebo Fenwickův strom v $\mathcal{O}(\log N)$ (obě operace), a tedy kdybychom místo pole $done[\ldots]$ použili nějakou takovou datovou strukturu, dostaneme se opět na čas $\mathcal{O}(N \log N)$.

```
#include <bits/stdc++.h>
using namespace std;

#define SIZE 123456
int n;
int K[SIZE], P[SIZE], Q[SIZE], tmp[SIZE];
long long int pocet = 0;

void mergesort(int l, int r) {
    if (r-l <= 1)
        return;
    int m = (l+r)/2;
    mergesort(l, m);
    mergesort(m, r);
    int i = l, j = m, k = 0;
    while (k < r-l) {
        if (j < r && (i == m || K[j] < K[i])) {
            tmp[k++] = K[j++];
        } else {
            tmp[k++] = K[i++];
            pocet+=(r-j);
        }
    }
    for (int i = 0; i < r-l; i++)
        K[l+i] = tmp[i];
}

int main() {
    scanf("%d", &n);
    int d;
    for (int i = 0; i < n; i++) {
        scanf("%d", &d);
        P[i] = d-1;
    }
    for (int i = 0; i < n; i++) {
        scanf("%d", &d);
        Q[d-1] = i; // Ve skutečnosti už Q^-1
```

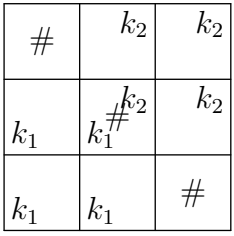
```
    }

    for (int i = 0; i < n; i++)
        K[i] = Q[P[i]];

    mergesort(0, n);
    printf("%lld\n", pocet);
    return 0;
}
```

P-I-2 Rekonstrukce školy

Kdyby učitel nesměl žádnou překážku rozbít, úlohu lze jednoduše vyřešit libovolným algoritmem pro hledání cesty, například prohledáváním do šířky: Postupně hledáme a označujeme políčka, na něž se učitel může dostat po $0, 1, 2, \ldots$ krocích, dokud buď nedojdeme do třídy, nebo nezjistíme, že jsme navštívili všechna dostupná políčka. Pamatujeme-li si, odkud jsme na dostupná políčka přišli, není pak problém vypsat cestu učitele. Každé políčko navštívíme nejvýše jednou, časová složitost tedy bude úměrná jejich počtu $\mathcal{O}(RS)$.

Jednoduché řešení tedy je vyzkoušet všechny možnosti, kterou překážku učitel rozbije (těch může být až lineárně mnoho), tuto překážku odebrat a aplikovat postup z předchozího odstavce. Výsledná časová složitost takového řešení je $\mathcal{O}(R^2S^2)$.

Zkusme tento postup vylepšit. Uvažme nějakou možnou cestu učitele z kabinetu do třídy. Nechť k_1 je první pozice na této cestě, v níž učitel rozbije nějakou překážku p, a nechť k_2 je poslední pozice na cestě, v níž se učitel překrývá s překážkou p (je případně možné, že $k_1 = k_2$). Jako C_1 označme část cesty před k_1 a jako C_2 označme část cesty za k_2. Ani C_1 ani C_2 tedy nerozbíjí žádnou překážku. Jak vypadá část C cesty mezi k_1 a k_2? Jestliže $k_1 = k_2$, pak C je prázdná. Jestliže k_1 sousedí s k_2 horizontálně nebo vertikálně, pak bez újmy na obecnosti můžeme předpokládat, že C se skládá pouze z jednoho kroku. Jestliže k_1 a k_2 sousedí diagonálně a alespoň jedna ze dvou cest délky 2 mezi nimi nerozbije žádnou jinou překážku, pak C může být taková cesta délky 2. Je ale možná i situace na následujícím obrázku, kde prostřední překážka je p a žádná z cest délky 2 možná není:



V tomto případě je C nějaká cesta, která nerozbíjí žádnou překážku (ani p).

Označme dvě pozice učitele, na nichž nepřekrývá žádnou překážku, jako propojené, jestliže mezi nimi existuje cesta učitele nerozbíjející žádnou překážku. Které dvojice pozic jsou navzájem propojené si můžeme snadno předpočítat v čase $\mathcal{O}(RS)$: Skupiny navzájem propojených pozic si budeme číslovat. Projdeme postupně všechny pozice a jestliže se aktuálně uvažovaná pozice zatím nenachází v žádné skupině,

postupem popsaným v prvním odstavci nalezneme všechny z ní dostupné pozice a označíme si je číslem nově vytvořené skupiny. Na konci tohoto postupu jsou dvě pozice propojené právě tehdy, když mají stejné číslo skupiny.

Nechť počáteční pozice učitele je ve skupině 1 a cílová ve skupině 2 (je-li učitel na začátku ve stejné skupině jako jeho cílová třída, pak mezi nimi existuje cesta nerozbíjející žádnou překážku a postupujeme stejně jako v prvním odstavci). Přípustná cesta učitele rozbíjející překážku p existuje, jestliže existují pozice k_1 a k_2 učitele rozbíjející pouze překážku p takové, že k_1 sousedí s pozicí ve skupině 1, k_2 sousedí s pozicí ve skupině 2 a buď mezi k_1 a k_2 existuje cesta délky nejvýše 2 nerozbíjející žádnou jinou překážku, nebo k_1 i k_2 sousedí s navzájem propojenými pozicemi nerozbíjejícími žádnou překážku.

Pro každou překážku p stačí otestovat pouze konstantně mnoho pozic k_1 a k_2, a zbylé podmínky lze díky předpočítaným číslům skupin navzájem propojených pozic ověřit v konstantním čase. Jelikož překážek je nejvýše RS, v čase $\mathcal{O}(RS)$ tedy dokážeme nalézt překážku, po jejímž rozbití se učitel dokáže dostat do třídy (případně rozhodnout, že taková překážka neexistuje). Tuto překážku odstraníme a postupujeme jako v prvním odstavci, výsledná časová složitost tedy bude $\mathcal{O}(RS)$. Jelikož si potřebujeme pamatovat pouze konstantní množství informace pro každou pozici, paměťová složitost je také $\mathcal{O}(RS)$.

```
#include <cstdio>
#include <cstdlib>
#include <list>
#include <vector>

using namespace std;

static char mapa[2001][2001];
static char smer_prichodu[2001][2001];
static int skupina[2001][2001];
static int R, S;

struct pozice
{
  int r, s;
  pozice(void) { }
  pozice(int _r, int _s) : r(_r), s(_s) { }
};

static const int smer[4][2] = {{0,1},{0,-1},{1,0},{-1,0}};
static const char smer_pismeno[4] = {'P', 'L', 'D', 'N'};

static char
pismeno_na_opacny(char c)
{
  switch (c)
    {
    case 'P' : return 1;
    case 'L': return 0;
    case 'D': return 3;
    case 'N': return 2;
    default: abort();
    }
```

```
}

static bool
pripustna(pozice const& p, int znic)
{
  if (p.r < 0 || p.s < 0 || p.r > R - 2 || p.s > S - 2)
    return false;
  int nint = 0;
  nint += (mapa[p.r][p.s] == '#');
  nint += (mapa[p.r + 1][p.s] == '#');
  nint += (mapa[p.r][p.s + 1] == '#');
  nint += (mapa[p.r + 1][p.s + 1] == '#');

  return nint <= znic;
}

static void
hledej_cesty_z(pozice const& z, int sk)
{
  list<pozice> fronta;

  fronta.push_back(z);
  skupina[z.r][z.s] = sk;
  smer_prichodu[z.r][z.s] = 0;
  while (!fronta.empty())
    {
      pozice a = fronta.front();
      fronta.pop_front();

      for (int sm = 0; sm < 4; sm++)
        {
          pozice n(a.r + smer[sm][0], a.s + smer[sm][1]);
          if (!pripustna(n, 0) || skupina[n.r][n.s])
            continue;

          skupina[n.r][n.s] = sk;
          smer_prichodu[n.r][n.s] = smer_pismeno[sm];
          fronta.push_back(n);
        }
    }
}

static void
vypis_cestu(pozice const &d)
{
  vector<char> cesta;
  vector<char>::reverse_iterator ci;
  pozice a = d;
  char c;

  while ((c = smer_prichodu[a.r][a.s]) != 0)
    {
      cesta.push_back(c);
      int sm = pismeno_na_opacny(c);
      a.r += smer[sm][0];
      a.s += smer[sm][1];
    }

  for (ci = cesta.rbegin(); ci != cesta.rend(); ++ci)
```

```
    printf("%c", *ci);
  printf("\n");
}

static bool
dostupna(pozice const &k, int sk)
{
  for (int sm = 0; sm < 4; sm++)
    {
      pozice pk(k.r + smer[sm][0], k.s + smer[sm][1]);
      if (pripustna(pk, 0)
          && skupina[pk.r][pk.s] == sk)
        return true;
    }

  return false;
}

static bool
odstranitelna_pres(pozice const &k1, pozice const &k2, int zac_sk, int kon_sk)
{
  if (!dostupna(k1, zac_sk)
      || !dostupna(k2, kon_sk))
    return false;

  int dr = k1.r - k2.r;
  int ds = k1.s - k2.s;
  if (abs(dr) + abs(ds) < 2)
    return true;

  int minr, maxr, mins, maxs;

  minr = min(k1.r, k2.r);
  maxr = max(k1.r + 1, k2.r + 1);
  mins = min(k1.s, k2.s);
  maxs = max(k1.s + 1, k2.s + 1);
  int nint = 0;
  for (int r = minr; r <= maxr; r++)
    for (int s = mins; s <= maxs; s++)
      if (mapa[r][s] == '#')
        nint++;

  if (nint < 3)
    return true;

  for (int sm = 0; sm < 4; sm++)
    {
      pozice pk1(k1.r + smer[sm][0], k1.s + smer[sm][1]);
      if (!pripustna(pk1, 0))
        continue;

      if (dostupna(k2, skupina[pk1.r][pk1.s]))
        return true;
    }

  return false;
}

static bool
odstranitelna(pozice const &p, int zac_sk, int kon_sk)
```

```
{
  for (int r1 = -1; r1 <= 0; r1++)
    for (int s1 = -1; s1 <= 0; s1++)
      {
        pozice k1(p.r + r1, p.s + s1);
        if (!pripustna(k1, 1))
          continue;

        for (int r2 = -1; r2 <= 0; r2++)
          for (int s2 = -1; s2 <= 0; s2++)
            {
              pozice k2(p.r + r2, p.s + s2);
              if (pripustna(k2, 1)
                  && odstranitelna_pres(k1, k2, zac_sk, kon_sk))
                return true;
            }
      }

  return false;
}

int main(void)
{
  pozice zacatek, konec;
  bool zacatek_nastaven = false;
  bool konec_nastaven = false;

  scanf("%d%d", &R, &S);
  for (int r = 0; r < R; r++)
    {
      scanf("%s", mapa[r]);
      for (int s = 0; s < S; s++)
        {
          if (mapa[r][s] == '#')
            continue;

          if (mapa[r][s] == 'K' && !zacatek_nastaven)
            {
              zacatek.r = r;
              zacatek.s = s;
              zacatek_nastaven = true;
            }
          if (mapa[r][s] == 'T' && !konec_nastaven)
            {
              konec.r = r;
              konec.s = s;
              konec_nastaven = true;
            }

          mapa[r][s] = '.';
        }
    }

  int pocet_skupin = 1;
  hledej_cesty_z(zacatek, 1);
  if (skupina[konec.r][konec.s] == 1)
    {
      vypis_cestu(konec);
```

```
      return 0;
    }

  for (int r = 0; r < R - 1; r++)
    for (int s = 0; s < S - 1; s++)
      {
        pozice a(r, s);
        if (!skupina[r][s] && pripustna(a, 0))
          hledej_cesty_z(a, ++pocet_skupin);
      }

  int kon_sk = skupina[konec.r][konec.s];
  for (int r = 0; r < R; r++)
    for (int s = 0; s < S; s++)
      if (mapa[r][s] == '#')
        {
          if (odstranitelna(pozice(r, s), 1, kon_sk))
            {
              mapa[r][s] = '.';
              for (int r1 = 0; r1 < R; r1++)
                for (int s1 = 0; s1 < S; s1++)
                  skupina[r1][s1] = 0;
              hledej_cesty_z(zacatek, 1);
              vypis_cestu(konec);
              return 0;
            }
        }

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

P-I-3 Mapa

Představme si, že stojíme v Kocourkově. Nejprve si pro každé město m určíme úhel β_m ve stupních mezi směrem k tomuto městu a východem (směrem kladné x-ové osy na mapě), měřeným po směru hodinových ručiček. Zajímá nás, zda existuje úhel α takový, že v intervalu $\langle \alpha - 90, \alpha + 90 \rangle$ není žádný z úhlů β_m; pootočíme-li mapu o tento úhel, bude Kocourkov nejvíc vpravo.

Jednoduché řešení je tedy následující: úhly β_m si setřídíme podle velikosti a jedním průchodem zjistíme, zda mezi dvěma po sobě jdoucími úhly β_i a β_{i+1} je rozdíl více než 180. Pokud takové dva úhly najdeme, jako výsledek můžeme vrátit jejich aritmetický průměr $(\beta_i + \beta_{i+1})/2$, jinak řešení neexistuje. Musíme si ale dát pozor na to, že hledaná mezera by mohla být i mezi posledním a prvním prvkem této posloupnosti. Popsané řešení má časovou složitost $\mathcal{O}(N \log N)$, jelikož tak dlouho nám zabere třídění posloupnosti úhlů.

Toto řešení můžeme ještě vylepšit. Povšimněme si, že hledaný prázdný interval $\langle \alpha - 90, \alpha + 90 \rangle$ nemůže celý ležet uvnitř kratšího intervalu, například $I_i = \langle 90 \cdot (i-1), 90 \cdot i \rangle$ pro libovolné přirozené číslo i. Z každého takového intervalu I_i tedy řešení může ovlivnit pouze největší a nejmenší úhel, který v něm leží. Všechny úhly v naší posloupnosti padnou do jednoho z intervalů $I_1, \ldots, I_4$, a minima a maxima z těchto intervalů snadno zjistíme jedním průchodem v čase $\mathcal{O}(N)$. Vyřešit

úlohu pro výsledných nejvýše 8 úhlů je pak triviální, dostáváme tedy řešení s časovou složitostí $\mathcal{O}(N)$. Načítaná čísla si ani nemusíme ukládat, paměťová složitost je tedy konstantní.

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

struct interval
{
  double nejmensi, nejvetsi;
};

int main(void)
{
  interval intervaly[4];

  /* Inicializujeme na hodnoty větší/menší než libovolné přípustné. */
  for (int i = 0; i < 4; i++)
    {
      intervaly[i].nejmensi = 370;
      intervaly[i].nejvetsi = -10;
    }

  int n, kx, ky, x, y;
  scanf("%d%d%d", &n, &kx, &ky);

  /* Zvlášť ošetřeme speciální případ, že Kocourkov je jediné město. */
  if (n == 1)
    {
      printf("0\n");
      return 0;
    }

  for (int i = 0; i < n - 1; i++)
    {
      scanf("%d%d", &x, &y);
      x -= kx;
      y -= ky;

      double uhel = (180 * atan2 (y, x) / M_PI);
      if (uhel < 0)
        uhel += 360;

      int ino = (int) (uhel / 90);
      intervaly[ino].nejmensi = min(intervaly[ino].nejmensi, uhel);
      intervaly[ino].nejvetsi = max(intervaly[ino].nejvetsi, uhel);
    }

  vector<double> posloupnost;
  for (int i = 0; i < 4; i++)
    {
      /* Ignorujeme prázdné intervaly. */
      if (intervaly[i].nejmensi > 360)
        continue;

      /* Když interval obsahuje jen jedno číslo, přidáme ho dvakrát, což nevadí. */
      posloupnost.push_back(intervaly[i].nejmensi);
```

```
      posloupnost.push_back(intervaly[i].nejvetsi);
    }
  /* Abychom nemuseli zvlášť ošetřovat mezeru mezi posledním a prvním číslem. */
  posloupnost.push_back(posloupnost[0] + 360);

  for (auto i = posloupnost.begin(); i + 1 != posloupnost.end(); ++i)
    {
      double rozdil = *(i + 1) - *i;
      if (rozdil > 180)
        {
          double alpha = (*(i + 1) + *i) / 2;
          /* Aby výsledek byl v rozmezí [0,360). */
          if (alpha >= 360)
            alpha -= 360;
          printf("%.2f\n", alpha);
          return 0;
        }
    }

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

P-I-4 Stromochod

a) K otestování, zda je vrcholů sudý počet, postačí lehce upravit příklad ze studijního textu. Budeme procházet strom do hloubky a udržovat proměnnou, která říká, zda jsme zatím prošli sudý nebo lichý počet vrcholů. Kdykoliv budeme opouštět vrchol, proměnnou znegujeme. Jelikož kořen opouštíme naposledy, podle stavu proměnné poznáme, jak máme odpovědět. Časová složitost programu bude lineární, neboť každý vrchol navštívíme třikrát.

```
type vrchol = record
   stav: 0..3;                { Kolikrát jsme ve vrcholu byli }
   ok: boolean;               { V kořeni výstup, jinde nevyužito }
end;

var lichy: boolean;           { Potkali jsme zatím lichý počet vrcholů? }

begin
   lichy := false;
   repeat
      V.stav := V.stav + 1;
      case V.stav of
         1: if ex_l then jdi_l;
         2: if ex_p then jdi_p;
         3: lichy := not lichy;
            if ex_o then jdi_o
            else begin
               V.ok := not lichy;
               halt;
            end;
      end;
   until false;
end.
```

b) Nyní máme otestovat, zda počet vrcholů je mocnina dvojky. To je totéž, jako že počet vrcholů lze opakovaně dělit dvěma beze zbytku, až nakonec vyjde jednička.

Tuto myšlenku můžeme realizovat značkami ve vrcholech: nejprve budou označeny všechny vrcholy. Pak každou druhou značku zrušíme a zkontrolujeme, že jich byl sudý počet. Pak zrušíme každou druhou ze zbývajících značek a tak dále. Skončíme buďto je-li v nějakém kroku počet vrcholů lichý (tehdy odpovíme negativně), nebo zbude-li jediná značka v kořeni (odpovíme pozitivně).

Program se bude poměrně přímočaře řídit touto myšlenkou, pouze si zjednodušíme práci tím, že značku reprezentujeme hodnotou `false`, takže všechny vrcholy jsou už na začátku výpočtu implicitně označeny.

```
type vrchol = record
   stav: 0..3;                { Kolikrát jsme ve vrcholu byli }
   skrt: boolean;             { Byla už značka škrtnuta? }
   ok: boolean;               { V kořeni výstup, jinde nevyužito }
end;

var lichy: boolean;           { Potkali jsme zatím lichý počet vrcholů? }
    pocet: 0..2;              { Kolik jsme potkali značek? 0, 1, 2 či víc }

procedure pruchod;
begin
   repeat
      V.stav := V.stav + 1;
      case V.stav of
         1: if ex_l then jdi_l;
         2: if ex_p then jdi_p;
         3: if not V.skrt then begin
               { Vrchol je stále označený }
               if lichy then V.skrt := true;
               lichy := not lichy;
               if pocet < 2 then pocet := pocet + 1;
            end;
            V.stav := 0;      { Reset stavu pro příští průchod }
            if ex_o then jdi_o else exit;
      end;
   until false;
end;

begin
   repeat
      lichy := false;
      pocet := 0;
      pruchod;
      if lichy then begin
         { Pokud byl označených vrcholů lichý počet, skončíme }
         if pocet=1 then V.ok := true;  { Jediný označený => úspěšně }
                    else V.ok := false; { Více takových => neúspěšně }
         halt;
      end;
      { Jinak pokračujeme dalším průchodem }
   until false;
end.
```

Zbývá si rozmyslet, jakou má tento program časovou složitost. Každé projití značek ve stromu s N vrcholy nás stojí čas $\mathcal{O}(N)$. Strom přitom projdeme $\mathcal{O}(\log N)$-krát, čímž tedy strávíme celkem čas $\mathcal{O}(N \log N)$.

c) Nakonec chceme testovat, zda je strom úplný. To znamená, že na první až k-té hladině (pro vhodné $k \geq 0$) má každý vrchol právě dva syny a na $(k+1)$-ní hladině leží všechny listy.

Program bude postupovat po hladinách. Na počátku označí kořen, což je jediný vrchol na první hladině. V každém dalším průchodu nalezne všechny označené vrcholy a značky posune do jejich synů. Přitom si bude udržovat tři booleovské proměnné S_0, S_1 a S_2. Proměnná S_i bude indikovat, zda už jsme viděli nějaký vrchol s i syny. Na konci průchodu se rozhodneme takto:

- Pokud $S_1 = \mathit{true}$, zastavíme se a strom zamítneme (žádný úplný strom neobsahuje vrchol s jedním synem).
- Pokud $S_0 = \mathit{true}$ a $S_2 = \mathit{false}$, zastavíme se a strom přijmeme (narazili jsme na poslední hladinu, kde jsou samé listy).
- Pokud $S_0 = \mathit{false}$ a $S_2 = \mathit{true}$, pokračujeme dál (hladina obsahovala samé vrcholy se dvěma syny).
- V ostatních případech se zastavíme a zamítneme (směs vrcholů různých typů).

```
type vrchol = record
   stav: 0..3;                { Kolikrát jsme ve vrcholu byli }
   znacka: boolean;           { Leží vrchol v aktuální hladině? }
   ok: boolean;               { V kořeni výstup, jinde nevyužito }
end;

var s0, s1, s2: boolean;      { Viděli jsme vrchol s 0, 1, 2 syny? }

procedure pruchod;
begin
   repeat
      V.stav := (V.stav + 1) mod 3;  { Automaticky resetujeme pro příští průchod }
      case V.stav of
         1: { Do vrcholu jsme vstoupili shora. Nejprve aktualizujeme s0, s1, s2. }
            if ex_l and ex_p then s2 := true
            else if not ex_l and not ex_p then s0 := true
            else s1 := true;
            { Je-li označen, předáváme značky do synů a vracíme se nahoru }
            if V.znacka then begin
               V.znacka := false;
               if ex_l then begin jdi_l; V.znacka := true; jdi_o; end;
               if ex_p then begin jdi_p; V.znacka := true; jdi_o; end;
               V.stav := 2;
            end
            { Není-li označen, procházíme obvyklým způsobem }
            else if ex_l then jdi_l;
         2: if ex_p then jdi_p;
         0: if ex_o then jdi_o else exit;
      end;
   until false;
end;
```

```
begin
   V.znacka := true;          { Na počátku označen právě kořen }
   repeat
      s0 := false; s1 := false; s2 := false;
      pruchod;
      if s1             then begin V.ok := false; halt; end;
      if s0 and not s2 then begin V.ok := true;  halt; end;
      if s0 or not s2  then begin V.ok := false; halt; end;
   until false;
end.
```

Stanovíme časovou složitost. V úplném binárním stromu platí, že každá hladina obsahuje dvakrát víc vrcholů než ta předchozí, takže na i-té hladině leží 2^i vrcholů. V prvních k hladinách tedy $2^0 + 2^1 + \ldots + 2^{k-1}$ vrcholů. Tento součet je roven $2^k - 1$ – buďto můžeme použít vzorec pro součet geometrické řady, nebo si všimnout, jak jednotlivá čísla vypadají zapsaná ve dvojkové soustavě (2^i je jednička a za ní i nul; součet je tedy číslo z k jedniček).

Spustíme-li náš algoritmus na úplný strom s h hladinami, v prvním průchodu projde první hladinu, v dalším první dvě hladiny, a tak dále, až v posledním všech h hladin. Podle předchozího výpočtu tedy v k-tém průchodu navštíví $2^k - 1 < 2^k$ vrcholů. Ve všech průchodech proto nejvýše $2^1 + 2^2 + \ldots + 2^h$, což je dvojnásobek celkového počtu vrcholů. Časová složitost programu tudíž činí $\mathcal{O}(N)$.