P-III-1 Bizoní rezervace

Zadány jsou alespoň tři přímky v obecné poloze a hledáme konvexní obal všech jejich průsečíků. V řešení budeme využívat standardních pojmy a postupy z oblasti geometrických algoritmů. Nebudeme je zde detailně odvozovat, všechny je najdete přehledně popsané v Programátorské kuchařce KSP.*

Jednoduché řešení

Povšimněme si, že průsečíků N přímek je pouze $N \cdot (N-1)/2 < N^2$. Můžeme pro každou dvojici přímek určit jejich průsečík a na závěr spočítat jejich konvexní obal. Průsečík dvou přímek zvládneme najít v konstantním čase a konvexní obal K bodů v čase $\mathcal{O}(K \log K)$. Celková asymptotická časová složitost tohoto přímočarého algoritmu je $\mathcal{O}(N^2 \log(N^2))$, což je rovno $\mathcal{O}(N^2 \log N)$, neboť $\log(N^2) = 2 \log N$.

Rychlejší řešení

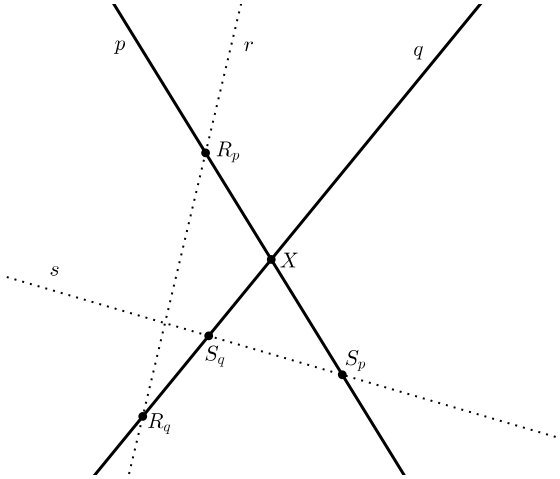
Nejprve popíšeme jednoduše znějící algoritmus a až poté ukážeme jeho správnost. Začneme seřazením všech přímek vzestupně podle jejich směrnic, tedy podílu $-B_i/A_i$. V případě, že $A_i = 0$ (jedná se o přímku rovnoběžnou s osou x), uvažujeme směrnici nekonečnou a zařadíme takovou přímku na konec. Následně spočteme průsečík každých dvou přímek, které jsou sousední v našem pořadí a navíc uvažujeme, že první přímka sousedí s tou poslední (uspořádání je cyklické). Na závěr vezmeme všechny tyto průsečíky a nalezneme jejich konvexní obal standardním algoritmem zmíněným výše. Nalezený konvexní obal popisuje hledanou ohradu s nejmenším obvodem, která obsahuje také všechny ostatní průsečíky přímek.

Z uvedeného algoritmu není na první pohled vůbec jasné, proč by měl fungovat. Budeme muset ukázat, že průsečík dvou přímek, které nejsou sousední v cyklickém seznamu přímek uspořádaném podle směrnic, leží v nalezeném konvexním obalu. Uvažme jednu takovou dvojici přímek p, q a jejich průsečík X. Protože tyto přímky nejsou sousední, tak pokud budeme procházet cyklický seznam přímek postupně od přímky p do přímky q, nalezneme mezi nimi alespoň jednu jinou přímku. Vezměme libovolnou takovou přímku r. Podobně procházíme-li seznamem od přímky q do přímky p, nalezneme nějakou přímku s. Označme průsečíky přímky r s přímkami p, q postupně R_p, R_q a průsečíky přímky s s přímkami p, q postupně S_p, S_q (každé dvě přímky se protínají v jednom bodě a všechny průsečíky jsou ze zadání různé).

Dále ukážeme, že průsečík X přímek p a q leží na přímce mezi dvěma jinými průsečíky dvou zadaných přímek. Pro spor předpokládejme, že tomu tak není a zaměřme se na polopřímky XR_p a XR_q. Pokud by bod S_p ležel na polopřímce XR_p a zároveň bod S_q ležel na polopřímce XR_q, pak by v našem cyklickém uspořádání přímek podle směrnic byla přímka s také na cestě od přímky p do přímky q, což je

* *https://ksp.mff.cuni.cz/kucharky/geometrie/*

ve sporu s její volbou. Jeden z bodů S_p, S_q musí tedy ležet na polopřímce opačné a bod X leží buď na úsečce R_pS_p nebo na úsečce R_qS_q, a proto ho nebylo potřeba uvažovat při tvorbě konvexního obalu.



Popsaný algoritmus se skládá ze dvou kroků a jak řazení N přímek dle směrnic, tak i hledání konvexního obalu N bodů lze implementovat s asymptotickou složitostí $\mathcal{O}(N \log N)$. Paměťová složitost našeho algoritmu je lineární v N.

```
#include <cstdio>
#include <algorithm>

#define MAX_N 1000000
int A[MAX_N], B[MAX_N], C[MAX_N];
int N;

/*
 * Porovná směrnice A[i] / B[i] a vrátí true, pokud směrnice
 * přímky p je menší nez směrnice přímky q. Abychom nemuseli
 * převádět celá čísla na double a ztrácet tím přesnost, upravíme
 * zkoumanou nerovnost A[p] / B[p] < A[q] / B[q] do tvaru, kde
 * se pouze násobí. Při násobení je třeba být opatrný na přetečení.
 * Poznamenejme, že přímka s B[i] = 0 bude zařazena na konec.
 */
bool porovnej_smernice(int p, int q) {
  return (long long)A[p] * B[q] < (long long)A[q] * B[p];
}

// Indexy přímek uspořádané vzestupně podle jejich směrnic.
int primky[MAX_N];

/*
 * Hledáme průsečík přímek
 * p: A[p] * x + B[p] * y + C[p] = 0
 * q: A[q] * x + B[q] * y + C[q] = 0
 * Řešíme soustavu dvou lineárních rovnic o dvou neznámých.
 */
```

```
void prusecik_primek(int p, int q, double &x, double &y) {
  x = (double)(B[q] * C[p] - B[p] * C[q]) / (B[q] * A[p] - B[p] * A[q]);
  y = (double)(A[q] * C[p] - A[p] * C[q]) / (A[q] * B[p] - A[p] * B[q]);
}

// Průsečíky přímek sousedících v cyklickém uspořádání.
double prusecik_X[MAX_N], prusecik_Y[MAX_N];

/*
 * Následuje standardní algoritmus na výpočet konvexního obalu bodů v rovině.
 * Nejprve uspořádáme všechny body vzestupně podle souřadnice x.
 * Poté je procházíme v tomto pořadí a průběžně tvoříme horní a dolní
 * půl-obaly a nakonec sloučíme obě části do jednoho obalu.
 *
 * Funkci v programu volame jedinkrát, a proto pro jednoduchost načítáme
 * vstup i ukládáme výstup do globálních polí.
 *   Vstup: prusecik_X, prusecik_Y
 *   Výstup: obal délky delka_obalu obsahuje indexy průsečíků
 */
int obal[MAX_N], delka_obalu;
int horni[MAX_N], delka_horni, dolni[MAX_N], delka_dolni;

/* Primárně porovnáváme podle y, sekundárně podle x. */
bool porovnej_pruseciky(int a, int b) {
  if (prusecik_X[a] != prusecik_X[b])
    return prusecik_X[a] < prusecik_X[b];
  return prusecik_Y[a] < prusecik_Y[b];
}

int pruseciky[MAX_N];

/* Třetí složka vektorového součinu vektoru PQ a PR. */
long long vektorovy_soucin(int P, int Q, int R) {
  return (prusecik_X[Q] - prusecik_X[P]) * (prusecik_Y[R] - prusecik_Y[P])
       - (prusecik_Y[Q] - prusecik_Y[P]) * (prusecik_X[R] - prusecik_X[P]);
}

void konvexni_obal() {
  for (int i = 0; i < N; ++i)
    pruseciky[i] = i;
  std::sort(pruseciky, pruseciky + N, porovnej_pruseciky);

  delka_horni = delka_dolni = 0;

  for (int i = 0; i < N; ++i) {
    while (
      delka_horni > 1 &&
      vektorovy_soucin(i, horni[delka_horni - 1], horni[delka_horni - 2]) <= 0
    )
      delka_horni--;
    while (
      delka_dolni > 1 &&
      vektorovy_soucin(i, dolni[delka_dolni - 1], dolni[delka_dolni - 2]) >= 0
    )
      delka_dolni--;
    horni[delka_horni++] = i;
    dolni[delka_dolni++] = i;
  }
```

```
  delka_obalu = 0;
  for (int i = 0; i < delka_horni; ++i)
    obal[delka_obalu++] = horni[i];
  for (int i = delka_dolni - 2; i >= 1; --i)
    obal[delka_obalu++] = dolni[i];
}

int main(int argc, char *argv[]) {

  // Načteme vstup.
  scanf("%d", &N);
  for (int i = 0; i < N; ++i)
    scanf("%d%d%d", A + i, B + i, C + i);

  // Uspořádáme přímky vzestupně dle jejich směrnic.
  for (int i = 0; i < N; ++i)
    primky[i] = i;
  std::sort(primky, primky + N, porovnej_smernice);

  // Spočítáme průsečíky každých dvou přímek sousedních směrnic.
  for (int i = 0; i < N - 1; ++i)
    prusecik_primek(i, i + 1, prusecik_X[i], prusecik_Y[i]);
  prusecik_primek(N - 1, 0, prusecik_X[N - 1], prusecik_Y[N - 1]);

  // Spočítáme kondexní obal.
  konvexni_obal();

  // Vypíšeme výsledek.
  for (int i = 0; i < delka_obalu; ++i)
    printf("%.6lf %.6lf\n", prusecik_X[obal[i]], prusecik_Y[obal[i]]);

  return 0;
}
```

P-III-2 Hostina

Předpokládejme nejprve, že nikdo nepracuje přes půlnoc. Nechť t_1 je nejmenší čas, v němž jednomu ze zaměstnanců Z skončí pracovní doba. Hostimír nutně musí rozdat koláčky nejpozději v čase t_1, jinak by nepohostil zaměstnance Z. Na druhou stranu, určitě se mu vyplatí rozdat koláčky co nejpozději (tím může pohostit navíc další zaměstnance, kteří začínají později). Je tedy pro něj nejvýhodnější poprvé koláčky rozdat v čase t_1. Obdobnou úvahou dospějeme k tomu, že podruhé by měl rozdat koláčky v nejmenším čase t_2, v němž skončí pracovní doba nějakému zaměstnanci, který začal pracovat až po čase t_1, atd.

V proměnné t si budeme udržovat čas, v němž Hostimír naposledy rozdal koláčky (na začátku $t = -1$). Stačí si tedy zaměstnance setřídit vzestupně dle konce jejich pracovní doby a postupně je procházet; pokaždé, když narazíme na zaměstnance, který začal pracovat po čase t, rozdáme další dávku koláčků v okamžiku, kdy mu skončí pracovní doba (a hodnotu t nastavíme na tento čas). Časová složitost tohoto algoritmu je dominována složitostí třídění, tedy $\mathcal{O}(n \log n)$.

Co když ale někdo přes půlnoc pracuje? Kdybychom znali nějaký čas t_0, v němž Hostimír rozdá koláčky, pak ze vstupu můžeme vyhodit všechny zaměstnance pracující v čase t_0 a na zbylé aplikovat výše popsaný algoritmus (čas t_0 hraje roli půlnoci – můžeme například všechny pracovní doby přepočítat na jejich vzdálenost od času t_0).

Mohli bychom tedy vyzkoušet všechny možné časy t_0 – stačí si rozmyslet, že opět stačí jako t_0 volit konce pracovních dob zaměstnanců, je tedy pouze n možností. Z nich si vybereme tu, která potřebuje nejméně dávek koláčků. Tím dostáváme algoritmus s časovou složitostí $\mathcal{O}(n^2 \log n)$, který snadno vylepšíme na $\mathcal{O}(n^2)$, uvědomíme-li si, že je zbytečné konce pracovních dob třídit pokaždé znova – stačí si jednou určit jejich cyklické pořadí.

Jak by šel tento algoritmus ještě vylepšit? Uvědomme si, že algoritmus z předchozího odstavce zbytečně počítá stejné věci opakovaně. Jestliže Hostimír rozdá koláčky v nějakém čase t_1, následující čas t_2, v němž znovu rozdá koláčky (tj. první konec pracovní doby zaměstnance, který začal pracovat po čase t_1), je vždy stejný nezávisle na vybraném počátečním čase t_0 (samozřejmě s výjimkou případu, že bychom na čas t_0 narazili dříve než na konec pracovní doby tohoto zaměstnance, v tomto případě již jsou všichni zaměstnanci pohoštěni).

Předpočítejme si tedy pro každý relevantní čas t_1 (opět stačí uvažovat konce pracovních dob) odpovídající čas t_2; postupujeme přitom podobně jako ve druhém odstavci, pouze při určení tohoto času nepřenastavíme t na t_2, ale na nejbližší další konec pracovní doby po t_1. Výsledky si uložíme do pole `nasledujici`, které je indexované čísly zaměstnanců; prvek `nasledujici[`i`]` tedy odpovídá času, v němž končí pracovní doba zaměstnance i. Jako hodnotu `nasledujici[`i`].zamestnanec` si uložíme číslo zaměstnance, jehož konec pracovní doby je t_2. Kromě toho si jako hodnotu `nasledujici[`i`].doba` uložíme počet milisekund mezi časy t_1 a t_2 (tedy $t_2 - t_1$ je-li $t_2 > t_1$, nebo $m - t_1 + t_2$, je-li $t_2 < t_1$, tj. nastane-li mezi časy t_1 a t_2 půlnoc).

Nyní bychom mohli algoritmus z třetího odstavce přeformulovat takto: projdeme všechny zaměstnance a vyzkoušíme jako t_0 volit konce jejich pracovních dob. Nechť zkoušíme jako t_0 volit jako konec pracovní doby zaměstnance i_0. Pak inicializujeme $i = i_0$ a opakovaně provádíme $i =$`nasledujici[`i`].zamestnanec`, dokud součet dob uložených v navštívených prvcích pole `nasledujici` nedosáhne alespoň délky dne m. Počet opakování odpovídá počtu časů, v nichž Hostimír rozdá koláčky.

Může se zdát, že jsme si moc nepomohli, přímočará implementace popsaného algoritmu opět má časovou složitost $\mathcal{O}(n^2)$. Nicméně počet iterací pro každé i_0 můžeme určit i efektivněji. Nejprve si postupně předpočítejme, kam a za jakou dobu se z každého i dostaneme po 2, 4, 8, …, 2^k iteracích, kde $k = \lfloor \log_2 n \rfloor$. To zvládneme v čase $\mathcal{O}(n \log n)$ – abychom zjistili, kam se z i dostaneme po 2^{t+1} iteracích, stačí zjistit, kam se dostaneme po 2^t iteracích a kam se odtamtud dostaneme po dalších 2^t iteracích, a tyto hodnoty již máme předpočítané. S těmito předpočítanými informacemi můžeme nutný počet iterací pro každé i zjistit v čase $\mathcal{O}(\log n)$ půlením intervalů: Nejprve zjistíme, zda se po 2^k iteracích dostaneme na dobu alespoň m, a pokud ne, o těchto 2^k iterací se posuneme. Poté totéž opakujeme pro 2^{k-1}, …, 4, 2, 1 iterací. Celková časová i paměťová složitost výsledného algoritmu je $\mathcal{O}(n \log n)$.

```
#include <stdio.h>
#include <stdlib.h>

#define MAXN 1000000
#define MAXLOG 20

static int n, m;
static struct zamestnanec
{
  int a, b;
} zamestnanci[MAXN];

static struct
{
  int zamestnanec, doba;
} nasledujici[MAXLOG][MAXN];

static int cas_mezi(int f, int t)
{
  return (t - f + m) % m;
}

static int pracovni_doba(const struct zamestnanec *z)
{
  return cas_mezi(z->a, z->b);
}

static int dle_konce(const void *z1, const void *z2)
{
  const struct zamestnanec *zz1 = z1;
  const struct zamestnanec *zz2 = z2;

  if (zz1->b == zz2->b)
    return pracovni_doba(zz1) - pracovni_doba(zz2);
  return zz1->b - zz2->b;
}

static int v_intervalu(struct zamestnanec *interval, int cas)
{
  if (interval->a <= interval->b)
    return interval->a <= cas && cas <= interval->b;

  return cas >= interval->a || cas <= interval->b;
}

int main (void)
{
  int i, t, e, doba;

  scanf("%d%d", &n, &m);
  for (i = 0; i < n; i++)
    scanf("%d%d", &zamestnanci[i].a, &zamestnanci[i].b);
  qsort(zamestnanci, n, sizeof(struct zamestnanec), dle_konce);

  // Pro zjednodušení: končí-li více zaměstnanců ve stejnou dobu, stačí
  // uvažovat toho z nich s nejkratší pracovní dobou.
  e = 1;
  for (i = 1; i < n; i++)
    if (zamestnanci[i].b != zamestnanci[i - 1].b)
      zamestnanci[e++] = zamestnanci[i];
  n = e;
```

```
  e = 0;
  doba = 0;
  for (t = 0; t < n; t++)
    {
      while (v_intervalu(zamestnanci + e, zamestnanci[t].b))
        {
          int ne = (e + 1) % n;
          doba += cas_mezi(zamestnanci[e].b, zamestnanci[ne].b);
          e = ne;

          if (doba >= m)
            {
              /* Speciální případ: všichni pracují ve stejný čas (konec pracovní
                 doby zaměstnance t). */
              printf("1\n");
              return 0;
            }
        }
      nasledujici[0][t].zamestnanec = e;
      nasledujici[0][t].doba = doba;

      doba -= cas_mezi(zamestnanci[t].b, zamestnanci[(t + 1) % n].b);
    }

  int log = 0;
  while ((1 << log) <= n)
    log++;
  for (i = 1; i < log; i++)
    for (t = 0; t < n; t++)
      {
        int m = nasledujici[i - 1][t].zamestnanec;
        nasledujici[i][t].zamestnanec = nasledujici[i - 1][m].zamestnanec;
        nasledujici[i][t].doba = nasledujici[i - 1][t].doba
                               + nasledujici[i - 1][m].doba;
      }

  int min_potreba = n;
  for (t = 0; t < n; t++)
    {
      int potreba = 1;
      doba = 0;
      e = t;
      for (i = log - 1; i >= 0; i--)
        if (doba + nasledujici[i][e].doba < m)
          {
            doba += nasledujici[i][e].doba;
            potreba += (1 << i);
            e = nasledujici[i][e].zamestnanec;
          }

      if (potreba < min_potreba)
        min_potreba = potreba;
    }

  printf("%d\n", min_potreba);
  return 0;
}
```

P-III-3 Stromochod

Procházení stromu

Nejprve si pořídíme sadu funkcí na přerušované procházení stromu. Budeme chtít navštívit všechny vrcholy zvoleného podstromu, a přitom si kdykoliv moci odskočit do jiného podstromu.

Všechny vrcholy uspořádáme v *pre-orderu,* to znamená, že kořen bude první, pak všechny vrcholy v jeho levém podstromu a nakonec všechny vrcholy v pravém podstromu. Vrcholy uvnitř podstromů jsou opět uspořádány pre-orderem.

V procházeném podstromu budeme udržovat právě jeden vrchol označený jako *aktivní.* Chceme umět následující operace:

- *Init* zavoláme v kořeni podstromu a kořen se stane aktivním.
- *Enter* zavoláme v kořeni podstromu a stromochod se přesune do aktivního vrcholu.
- *Leave* zavoláme v aktivním vrcholu a stromochod se vrátí do kořene podstromu.
- *Next* zavoláme v aktivním vrcholu, stromochod se přesune do následníka tohoto vrcholu v pre-orderu a učiní ho aktivním; kdyby už žádný následník neexistoval, zastaví se v kořeni podstromu a oznámí neúspěch.
- *Reset* zavoláme v aktivním vrcholu, chceme-li procházení ukončit předčasně.

Strom tedy procházíme posloupností *Init, Next, Next,* ..., přičemž si kdykoliv můžeme operací *Leave* odskočit mimo podstrom a pomocí *Enter* se opět vrátit.

Operace budou využívat značek ve vrcholech stromu. Především budeme udržovat `stav` vrcholu: aktivní vrchol bude ve stavu 1, vrcholy na cestě z kořene k aktivnímu vrcholu budou označeny 2 nebo 3 podle toho, jestli cesta pokračuje doleva nebo doprava. Všechny ostatní vrcholy budou mít stav 0. Také si budeme pamatovat booleovskou proměnnou `koren`, podle které poznáme, že se nacházíme v kořeni podstromu.

```
procedure Init;
begin
   { Postačí nastavit značky a vrátit se }
   V.koren := true;
   V.stav := 1;
end;

procedure Enter;
begin
   { Projdeme po cestě zaznamenané ve stavech }
   while V.stav <> 1 do begin
      if V.stav=2 then jdi_l
                  else jdi_p;
   end
end;
```

```
procedure Leave;
begin
   { Vracíme se nahoru, než narazíme na kořen }
   while not V.koren do jdi_o;
end;

function Next: boolean;
begin
   if ex_l then begin            { Existuje-li levý syn, je následníkem }
      V.stav := 2;
      jdi_l;
   end else if ex_r then begin   { Zkusíme pravého syna }
      V.stav := 3;
      jdi_p;
   end else begin                { Jsme v listu => vracíme se nahoru }
      repeat
         V.stav := 0;
         jdi_o;
      until V.koren or ((V.stav=2) and ex_p);
      if (V.stav=2) and ex_p then begin
         V.stav := 3;            { Vrchol s pravým synem => jdeme doprava }
         jdi_p;
      end else begin             { Přišli jsme zprava do kořene => končíme }
         V.stav := 0;
         V.koren := false;
         Next := false;
         exit;
      end;
   end;
   V.stav := 1;                  { Aktivujeme nový vrchol }
   Next := true;
end;

procedure Reset;
begin
   { Projdeme se z aktivního vrcholu do kořene a smažeme značky }
   while not V.koren do begin
      V.stav := 0;
      jdi_o;
   end;
   V.stav := 0;
   V.koren := false;
end;
```

Rozebereme časovou složitost operací. *Init* zvládneme v konstantním čase. *Enter, Leave* a *Reset* stojí lineárně se vzdáleností mezi kořenem a aktivním vrcholem. *Next* může stát až lineárně s hloubkou podstromu, ale všechna volání *Next* dohromady v celém podstromu trvají lineárně s počtem vrcholů v podstromu, neboť každý vrchol navštívíme nejvýše třikrát.

Jednoduché řešení

Dokonalou vyváženost stromu můžeme testovat následovně. Budeme procházet všechny vrcholy stromu (prohledáním do hloubky ze studijního textu, nebo třeba právě popsaným mechanismem). V každém vrcholu chceme porovnat velikosti podstromů.

Kdyby nám stačilo ověřit, že podstromy jsou úplně stejně velké, stačilo by je procházet „paralelně“ – tedy střídat kroky v obou podstromech – a zkontrolovat, zda obě procházení skončí současně. Nejprve bychom tedy zavolali *Init* na obou podstromech a pak opakovaně *Enter*, *Next* a *Leave* v obou podstromech na střídačku.

Máme-li připustit i jedničkový rozdíl ve velikostech, pak v případech, kdy jedno procházení skončilo dříve než druhé, ověříme, že druhé udělá nejvýše jeden krok navíc.

Následující program se řídí tímto principem. Jen se navíc musíme vypořádat s tím, že levý či pravý syn může úplně chybět. Také si všimněte, že procházení stromu ve vnějším cyklu nijak neinterferuje s vnitřními cykly, protože vnitřní cykly navštěvují pouze vrcholy pod aktivním vrcholem vnějšího cyklu.

```
var next_l, next_p, next_l_old: boolean;

begin
   Init;
   V.ok := false;

   repeat
      { next_l, next_p budou říkat, která prohledávání ještě pokračují }
      if ex_l then next_l := true; begin jdi_l; Init; jdi_o; end
              else next_l := false;
      if ex_p then next_p := true; begin jdi_p; Init; jdi_o; end
              else next_p := false;

      while next_l or next_p do begin
         { Jeden krok v levém podstromu }
         next_l_old := next_l;
         if next_l then begin
            jdi_l; Enter;
            if Next then begin
               Leave;
               if not next_p then halt;  { Pravý skončil o 2 dřív }
            end else
               next_l := false;          { Levý právě skončil }
            jdi_o;
         end;

         { A teď totéž napravo }
         if next_p then begin
            jdi_p; Enter;
            if Next then begin
               Leave;
               if not next_l_old then halt;
            end else
               next_p := false;
            jdi_o;
```

```
        end;
      end;
   until not Next;
   V.ok := true;
end;
```

Jaká je časová složitost tohoto řešení? Postupně navštívíme n vrcholů, pro každý z nich projdeme všech nejvýše n vrcholů v podstromech a každý krok procházení trvá až lineárně s hloubkou podstromu, což může být až $\mathcal{O}(n)$. Celkem tedy $\mathcal{O}(n^3)$.

Malé vylepšení s velkými důsledky

Všimneme si, že předchozí algoritmus je tak pomalý pouze ve velmi specifických případech (mohou-li vůbec nastat). Dokážeme, že kdykoliv je strom dokonale vyvážený, přijdeme na to mnohem rychleji.

Nejprve nahlédneme, že každý dokonale vyvážený strom má logaritmickou hloubku. Vskutku: pokud se vydáme po libovolné cestě z kořene do listu, velikost podstromu, který leží pod námi, se s každým krokem zmenší alespoň dvakrát. Nejvýše po $L = \lfloor \log_2 n \rfloor$ krocích tedy musíme dojít do listu.

Nyní to využijeme při výpočtu časové složitosti. Hladiny stromu očíslujeme od 0 do L zdola nahoru. Na i-té hladině leží nejvýše 2^{L-i} vrcholů, každý z nich je kořenem podstromu hloubky nejvýše i o maximálně 2^i vrcholech. V tomto podstromu každý *Next*, *Enter* i *Leave* trvá $\mathcal{O}(i)$, takže zkoumáním podstromu strávíme čas $\mathcal{O}(i \cdot 2^i)$. Zkontrolovat všechny podstromy zakořeněné na i-té hladině proto trvá $\mathcal{O}(2^{L-i} \cdot i \cdot 2^i) = \mathcal{O}(2^L \cdot i)$. Jelikož $2^L \leq n$ a $i \leq \log_2 n$, můžeme předchozí výraz omezit $\mathcal{O}(n \log n)$. Na všech L hladinách tedy trávíme dohromady čas $\mathcal{O}(n \log^2 n)$.

V nevyváženém stromu se nám nicméně může stát, že nějaký podstrom bude příliš hluboký, takže jeho kontrola bude mnohem pomalejší. Poradíme si tak, že podstromy budeme kontrolovat zdola nahoru (třeba upraveným prohledáváním do hloubky) a jakmile narazíme na nevyvážený podstrom, okamžitě skončíme. Pak si totiž můžeme být jistí, že levý i pravý pod-podstrom nevyváženého podstromu jsou vyvážené (jinak bychom chybu objevili dřív). Proto se i v nevyváženém případě hloubky od logaritmu liší nejvýše o 1.

Takto upravený algoritmus má složitost $\mathcal{O}(n \log^2 n)$ i v nejhorším případě.

Počítadla

Předchozí řešení lze ještě zrychlit, byť se tím trochu zkomplikuje. Vrcholy v podstromech začneme doopravdy počítat. Stromochod sice nemá proměnné pro neomezená čísla, ale můžeme čísla ukládat do vrcholů stromu ve dvojkové soustavě: každý vrchol si v proměnné `bit` zapamatuje jeden bit čísla. Bity budou po sobě následovat v pre-orderu; začneme nejnižším bitem a za nejvyšší bit přidáme hodnotu -1 jako značku konce čísla.

Číslo vyjadřující počet vrcholů v podstromu se přitom vždy vejde do vrcholů tohoto podstromu, neboť $x > \log_2 x$ pro všechna $x \geq 1$. Jen pozor na to, že pokud $x = 1$, nevejde se už za číslo koncová -1. Konec čísla tedy poznáme buď podle -1, nebo podle toho, že *Next* vrátil `false`.

Strom budeme procházet do hloubky. Kdykoliv se budeme vracet z nějakého vrcholu, spočítáme číslo vyjadřující počet jeho potomků. Ten získáme sečtením už spočítaných počtů potomků levého a pravého syna. Při té příležitosti také počty potomků porovnáme, abychom ověřili vyváženost.

Zkusíme napsat podprogramy pro zvýšení čísla o 1 a porovnání čísel na rovnost.

```
procedure BinaryInc;
var p: integer;
begin
   Init;
   p := 1;
   while (p = 1) and (V.bit >= 0) do begin
      p := p + V.bit;
      V.bit := p mod 2;
      p := p div 2;
      Next;
   end;
   if V.bit < 0 then begin
      V.bit := 1;
      Next;
      V.bit := -1;
   end;
   Reset;
end;
```

```
function BinaryCmp: boolean;  { Porovná číslo v levém a pravém podstromu }
var x, y: integer;
    vysledek: boolean;
    next_l, next_p: boolean;
begin
   if ex_l then begin jdi_l; Init; jdi_o; next_l := true; end else next_l := false;
   if ex_p then begin jdi_p; Init; jdi_o; next_p := true; end else next_p := false;
   repeat
      if next_l then begin
         jdi_l; Enter; x := V.bit;
         if Next then Leave else next_l := false;
         jdi_o;
      end else x := -1;
      if next_p then begin
         jdi_p; Enter; y := V.bit;
         if Next then Leave else next_p := false;
      end else y := -1;
      if (x<0) and (y<0) then begin BinaryCmp := true; break; end;
      if x <> y then begin BinaryCmp := false; break; end;
   until false;
   if next_l then begin jdi_l; Enter; Reset; jdi_o; end;
   if next_p then begin jdi_p; Enter; Reset; jdi_o; end;
end;
```

Potřebujeme nicméně poznat, že dvě čísla se liší o jedničku. Jednoduchý způsob, jak to zařídit, je zkombinovat porovnávání se sčítáním. Během porovnávání si budeme představovat, že jsme jedno z čísel zvýšili o jedničku, ale výsledek nebudeme nikam ukládat, pouze s ním porovnávat.

Zbývá umět sečíst čísla uložená v synech do jednoho čísla uloženého v otci. Nabízí se nejprve přičíst pravé číslo k levému a poté výsledek přesunout do otce. Samotné přesunutí není problém: stačí každý bit přesunout do jeho předka v pre-orderu, na což stačí do našeho procházecího mechanismu doplnit operaci *Prev* pro přesun do předchůdce fungující podobně jako *Next.* Mohlo by se ovšem stát, že výsledek se do levého podstromu nevejde. Proto raději nejprve přesuneme a poté sčítáme. Jeden sčítanec tedy bude uložen v kořeni a levém podstromu, druhý v pravém podstromu.

Spočítejme složitost. Podstrom na i-té hladině (opět číslujeme zespoda) má hloubku i, takže *Prev, Next* a spol. trvají $\mathcal{O}(i)$. Pracujeme přitom s čísly o logaritmickém počtu bitů vzhledem k velikosti podstromů, tedy $\mathcal{O}(\log 2^i) = \mathcal{O}(i)$. Jedna operace s čísly tedy trvá $\mathcal{O}(i^2)$.

Sečtením přes jednu hladinu dostaneme $\mathcal{O}(2^{L-i} \cdot i^2) = \mathcal{O}(n/2^i \cdot i^2)$, sečtením přes celý strom pak:

$$\mathcal{O}\left(\sum_{i=0}^{L}\left(\frac{n}{2^i}\cdot i^2\right)\right) = \mathcal{O}\left(n\cdot\sum_{i=0}^{L}\frac{i^2}{2^i}\right) \subseteq \mathcal{O}\left(n\log n\cdot\sum_{i=0}^{L}\frac{i}{2^i}\right).$$

Poslední inkluze platí díky tomu, že i je vždy nejvýše $\log n$. Suma na její pravé straně je přitom shora omezena konstantou.* Složitost celého algoritmu je tedy $\mathcal{O}(n \log n)$.

Mělká počítadla

Algoritmus s počítadly se překvapivě dá ještě zlepšit. Místo abychom bity počítadla ukládali v pre-orderu, uložíme je po hladinách. Tím pádem přístup k jednotlivým bitům k-bitového počítadla bude trvat pouze $\mathcal{O}(\log k)$, zatímco předtím byl lineární v hloubce podstromu, kde je počítadlo uloženo. Jedna aritmetická operace pak stojí $\mathcal{O}(k \log k)$.

Technické detaily popíšeme pouze zhruba. V podstromu budeme udržovat „zarážku“ nastavenou na určitou hladinu: všechny vrcholy na této hladině opatříme speciální značkou a operaci *Next* naučíme, aby se o značku zarazila. Kdykoliv nám dojde místo na počítadlo, posuneme zarážku o hladinu níže. Naopak při přesouvání čísla ze syna do otce potřebujeme posunout i zarážku. Přesun zarážky přitom stojí $\mathcal{O}(k)$, takže si ho můžeme dovolit provést konstanta-krát na jednu aritmetickou operaci.

Přepočítáme složitost. Na i-té hladině jsou počítadla i-bitová, takže jedna operace stojí $\mathcal{O}(\log i)$. V součtu přes celou hladinu dostaneme $\mathcal{O}(2^{L-i} \cdot i \log i)$. Sečtením přes všechny hladiny dostaneme:

$$\mathcal{O}\left(\sum_{i=0}^{L}\left(\frac{n}{2^i}\cdot i\log i\right)\right) = \mathcal{O}\left(n\cdot\sum_{i=0}^{L}\frac{i\log i}{2^i}\right) \subseteq \mathcal{O}\left(n\log\log n\cdot\sum_{i=0}^{L}\frac{i}{2^i}\right).$$

* To je nerovnost známá z rozboru budování haldy zdola nahoru. Důkaz najdete například v kapitole o haldách na *http://mj.ucw.cz/vyuka/ads/.*

Inkluze plyne z toho, že $i \leq \log n$, takže $\log i \leq \log \log n$. Suma na pravé straně je opět omezena konstantou, takže jsme složitost celého algoritmu odhadli výrazem $\mathcal{O}(n \log \log n)$.

Překvapení na závěr

Nápady účastníků ústředního kola nás inspirovaly k dalšímu řešení, které pracuje v lineárním čase a překvapivě se obejde bez porovnávání velikostí podstromů. Je založené na opakování *redukčních kroků*. Jeden takový krok buďto strom zamítne jako nevyvážený, nebo ho upraví na menší strom, který je vyvážený pravě tehdy, když byl vyvážený původní strom.

Nejprve dokážeme pomocné tvrzení o struktuře dokonale vyvážených stromů, které je připodobňuje k úplným binárním stromům z domácího kola.

Lemma: Dokonale vyvážený strom je buďto úplný, nebo z něj smazáním nejnižší hladiny úplný strom vznikne.

Důkaz: Mějme dokonale vyvážený strom o h hladinách, který není úplný. Hladině $h-1$ budeme říkat *koruna* stromu. Hladiny od kořene do koruny doplníme *virtuálními* vrcholy na úplný strom. Pokud v koruně neleží žádné virtuální vrcholy, nemohou ležet ani jinde, takže tvrzení lemmatu platí.

V opačném případě pokračujeme: Vrcholy koruny, které mají syny o hladinu níže, nazveme *pupeny*. Nyní označíme s nejhlubší vrchol stromu, pod kterým je alespoň jeden virtuální vrchol a alespoň jeden pupen. Vrchol s sám není ani virtuální, ani pupen. Všimneme si, že v žádném podstromu pod vrcholem s nemohou být současně virtuální vrcholy a pupeny, protože by s bylo ještě možné posunout níž.

Existuje tedy jeden podstrom (řekněme levý), v němž jsou virtuální vrcholy, ale nejsou pupeny. V opačném podstromu (řekněme pravém) se naopak nachází pupeny, ale ne virtuální vrcholy. Nyní se podívejme na hladiny těchto podstromů ležící (neostře) nad korunou. V pravém podstromu jsou všechny tyto hladiny plné a pod nejnižší z nich je alespoň jeden list z h-té hladiny. V levém naopak alespoň jeden vrchol chyběl (byl doplněn virtuálním) a na h-té hladině neleží žádné vrcholy. Proto se velikosti obou podstromů liší aspoň o 2, takže strom nemohl být vyvážený.

Nyní popíšeme redukční krok. Nejprve otestujeme, zda strom splňuje podmínku z lemmatu. To můžeme ověřit v čase $\mathcal{O}(n)$ jednoduchou úpravou algoritmu na testování úplnosti stromu z domácího kola. Algoritmus nám vyznačí poslední úplnou hladinu (korunu) a zkontroluje, zda pod ní leží pouze listy.

Poté strom „očešeme“: podíváme se na vrcholy koruny a pokud je vrchol koruny list, odstraníme ho. Pokud má nějaké syny, odstraníme jednoho ze synů. Očesání jistě můžeme provést v lineárním čase. Všimneme si, že tím celkově ze stromu ubylo přesně tolik vrcholů, kolik jich leželo v koruně. Také si všimneme, že z obou podstromů pod libovolným vnitřním vrcholem v musel ubýt stejný počet vrcholů – mezi v a korunou je totiž strom úplný. Očesaný strom je tedy dokonale vyvážený právě tehdy, když takový byl původní strom.

Nyní dokážeme, že očesáním ubyla alespoň čtvrtina všech vrcholů. Kdyby strom byl úplný, v koruně leží polovina všech vrcholů (± 1), takže bychom polovinu vrcholů

smazali. Není-li úplný, pod korunou leží nejvýše tolik vrcholů, kolik jich je ve zbytku stromu, takže koruna tvoří alespoň čtvrtinu vrcholů.

Opakováním redukčního kroku tedy získáváme stromy velké nejvýše $(3/4) \cdot n$, $(3/4)^2 \cdot n$, ..., až po strom konstantní velikosti, kde jistě umíme dokonalou vyváženost testovat v konstantním čase. Každým stromem strávíme čas lineární s jeho velikostí, takže celkově algoritmus běží v čase $\mathcal{O}(n \cdot \sum_i (3/4)^i)$. Tato suma je geometrická řada se součtem $1/(1 - 3/4) = 4$. Celková složitost tedy činí $\mathcal{O}(n)$.