P-III-1 Lesník Jehlička II

Úlohu vyřešíme metodou, která se často používá pro geometrické problémy. Představíme si přímku p rovnoběžnou s osou x, která se pohybuje ve směru osy y (od záporných čísel ke kladným – tomuto směru budeme říkat zezdola nahoru) a budeme simulovat průnik této přímky se zadanými úsečkami. Za tímto účelem budeme používat následující datové struktury:

- Seznam úseček S protínajících v daném okamžiku přímku p, setříděný dle x-ové souřadnice těchto průsečíků. V tomto seznamu budeme potřebovat rychle vyhledávat, proto ho budeme ukládat jako vyhledávací strom. Za zmínku stojí, že si v této datové struktuře nepamatujeme souřadnice průsečíků s přímkou p. Tyto souřadnice se mění, kdykoliv se přímka p posune, a jejich upravování by bylo příliš pomalé. Místo toho si pamatujeme pouze pořadí těchto průsečíků, které se mění, jen když p projde jedním z průsečíků úseček. Navíc přitom stačí upravit pouze pořadí úseček, které tímto průsečíkem procházejí. Souřadnice průsečíků s p dopočítáváme až v průběhu vyhledávání z aktuální y-ové souřadnice p. Pro určení, mezi kterými dvěma úsečkami se pak nachází daný bod přímky p, můžeme použít metodu binárního vyhledávání, která vyžaduje výpočet y-ové souřadnice pouze pro $\mathcal{O}(\log N)$ úseček, se kterými porovnáváme.
- Frontu událostí U, které ovlivňují průnik p s úsečkami:
  - Spodní konce úseček, které jsou celé nad p.
  - Vodorovné úsečky. Ty potřebují ošetřit speciálně.
  - Průsečík každé dvojice úseček sousedících v S, jestliže existuje a leží nad p.
  - Horní konce úseček, které jsou nad p.

  Frontu událostí reprezentujeme haldou. Události porovnáváme dle jejich y-ových souřadnic, v případě shody podle pořadí v předešlém seznamu.

Na začátku je přímka p v $-\infty$, tedy seznam S je prázdný a fronta U obsahuje horní a dolní konce všech úseček. Pro zjednodušení některých operací přidejme do S dvě falešné úsečky rovnoběžné s osou y, v $-\infty$ a $+\infty$ (nebo libovolných bodech ležících nalevo a napravo od všech ostatních úseček). V průběhu simulace odebíráme z U nejmenší událost a upravujeme S a U dle ní. Program končí ve chvíli, kdy přímka p dorazí do $+\infty$, tedy když U je prázdná.

Vždy, když do S přidáme prvek, přidáme též do U událost průniku se sousedními úsečkami, pokud průnik existuje a leží nad p.

Popišme nyní detailněji, jak zpracováváme událost způsobenou úsečkou u. Události se stejnou y-ovou souřadnicí zpracováváme v tomto pořadí:

- *Spodní konec úsečky u:* Přidáme u do S tak, aby průniky jednotlivých úseček byla stále uspořádané.
- *Vodorovná úsečka $u = (x_1, y) - (x_2, y)$:* Vypíšeme průsečíky s úsečkami z S, jejichž x-ová souřadnice na přímce p leží v intervalu $\langle x_1, x_2 \rangle$.
- *Průsečík dvou úseček v bodě (x, y):* smažeme z S všechny úsečky procházející bodem (x, y) a dáme si je stranou. Vypíšeme (x, y).
- S je nyní setříděno podle průsečíků s přímkou p (může se změnit jen pořadí těch úseček, které se protínají na p, ale ty jsme dali stranou). Do S nyní zatřídíme úsečky, které jsme dali stranou.
- *Horní konec úsečky u:* Nechť u leží mezi úsečkami u_1 a u_2. Odstraníme u z S. Navíc, pokud se u_1 a u_2 protínají nad p, přidáme jejich průsečík do U.

Je třeba ještě dořešit jeden technický detail: *zaokrouhlovací chyby.* Pokud by způsobily záměnu pořadí dvou událostí v U či dvou úseček v S, mohlo by to vést ke špatnému výsledku. Proto je nutné všechna porovnání dělat přesně. Toho lze dosáhnout více způsoby, jednou z možností je počítat souřadnice průsečíku jako zlomky.

Paměťová složitost našeho algoritmu je lineární – jak U, tak S vždy obsahují $\mathcal{O}(N)$ prvků. Jaká je jeho časová složitost? Nechť P je počet průsečíků úseček. Každá operace s U či S zabere čas $\mathcal{O}(\log N)$. Spočítejme nyní počet těchto operací pro každou z událostí:

- *Spodní či horní konec úsečky:* Těchto událostí je $\mathcal{O}(N)$ a vyžadují konstantní počet operací s U a S. Každá z nich způsobí přidání nanejvýš dvou událostí do U.
- *Průsečík, v němž se protíná k úseček:* Těchto událostí je P a vyžadují $\mathcal{O}(k)$ operací s S. Každý průsečík způsobí přidání nanejvýš dvou a odebrání alespoň $k - 1$ událostí z U.

Počet událostí přidaných do U (včetně inicializace) je $\mathcal{O}(N + P)$ a počet odebraných událostí je tedy také $\mathcal{O}(N + P)$. Počet operací s S a U je omezen počtem událostí přidaných či odebraných z U a je $\mathcal{O}(N + P)$. Časová složitost proto je $\mathcal{O}((N + P) \log N)$. Zmiňme, že existuje i rychlejší (a podstatně složitější) řešení s časovou složitostí $\mathcal{O}(N \log N + P)$.*

```
#include <stdio.h>
#include <stdlib.h>
#include <set>
#include <queue>
#include <algorithm>
using namespace std;

// Největší společný dělitel dvou čísel
long long gcd(long long a, long long b) { return (b==0) ? a : gcd(b, a%b); }
```

* Viz B. Chazelle, H. Edelsbrunner: An Optimal Algorithm for Intersecting Line Segments in the Plane. Journal of the ACM **39**, 1992.

```
// Zlomky a práce s nimi
struct ratio {
    long long nom, den;                          // čitatel a jmenovatel
    ratio () { nom=0; den=1; }                   // jednička
    ratio(long long n) { nom=n; den=1; }         // celé číslo -> zlomek
    ratio(long long n, long long d) {            // úprava zlomku na základní tvar
        long long g = gcd(llabs(den), llabs(nom));
        nom=n/g; den=d/g;
        if (den<0) { nom=-nom; den=-den; }
     }
    ratio& operator =(long long b)  { nom=b; den=1; return *this; }
    ratio operator +(ratio b) const { return ratio(nom*b.den+b.nom*den, den*b.den); }
    ratio operator -(ratio b) const { return ratio(nom*b.den-b.nom*den, den*b.den); }
    ratio operator *(ratio b) const { return ratio(nom*b.nom, den*b.den); }
    ratio operator /(ratio b) const { return ratio(nom*b.den, den*b.nom); }
    bool operator  <(ratio b) const { return (*this-b).nom < 0; }
    bool operator ==(ratio b) const { return nom==b.nom && den==b.den; }
    bool operator <=(ratio b) const { return (*this-b).nom <= 0; }
    float f() const { return (nom+0.0)/den; }   // konverze na float
};

ratio p(-900000);                                // zametací přímka

// Úsečka ze vstupu
struct usecka {
    ratio x1, y1, x2, y2;                        // souřadnice konců
    bool prunik(usecka pr) const {               // má přímka průnik s jinou?
        ratio r = intersect(pr);
        return !(r<pr.y1 || pr.y2<r || r<y1 || y2<r);
    }
    ratio intersect(usecka pr) const {           // y-ová souřadnice průniku
        return (pr.dir() == dir()) ? 1000000 :
               (x(ratio(0)) - pr.x(ratio(0))) / (pr.dir() - dir());
    }
    ratio x(ratio y) const { return x1+dir()*(y-y1); } // x-ová souřadnice pro y=p
    ratio dir()const{ return (x2-x1)/(y2-y1); } // směrnice
    // porovnání dvou přímek (v pořadí jako na přímce y=p)
    bool operator < (const usecka &pr) const {
        if (x(p) < pr.x(p)) return true;
        if (pr.x(p) < x(p)) return false;
        return dir() < pr.dir();
    }
};

// Událost
struct event{
    ratio y;                                     // kdy nastává
    int type;   // typ: 0 začatek, 1 křížení, 2 vodorovná přímka, 3 konec
    usecka u;
    bool operator < (const event &e) const {
        if (y<e.y) return false;
        if (e.y<y) return true;
        return type > e.type;
    }
```

```
};

priority_queue <event> U;                          // fronta událostí
set <usecka> S;                                    // zadané úsečky
typedef set <usecka>::iterator iter;               // iterátor přes úsečky
int n;                                             // kolik je úseček
usecka prk[1000000];                               // pomocné pole na změněné přímky

// Pokud se a s b proniká nad zametací přimkou, vytvoříme událost.
void pronika(usecka a, usecka b) {
    event f;
    if (a.prunik(b)) {
        f.y = a.intersect(b); f.u = b; f.type = 1;
        if (p < f.y) U.push(f);
    }
}

void pridej(usecka u) {
    iter it = S.insert(u).first;                   // iterátor na přidaný prvek
    it--; pronika(*it, u);
    it++; it++; pronika(u, *it);
}

int main() {
    int x1, y1, x2, y2, i;
    FILE *f = fopen("lesnik.in", "r"), *fo=fopen("lesnik.out", "w");
    fscanf(f, "%i", &n);
    for (i=0; i<n; i++){
        usecka u;
        fscanf(f, "%i%i%i%i" , &x1, &y1, &x2, &y2);
        if (y2 < y1) { swap(x1, x2); swap(y1, y2); }
        if (y1 == y2) {
            if (x2 < x1) swap(x1, x2);
            u.x1 = x1; u.y1 = y1; u.x2 = x2; u.y2 = y2;
            event e; e.u = u; e.type = 2; e.y = u.y1;
            U.push(e);
        } else {
            event e;
            u.x1 = x1; u.y1 = y1; u.x2 = x2; u.y2 = y2;
            e.u = u;
            e.type = 0; e.y = u.y1; U.push(e);
            e.type = 3; e.y = u.y2; U.push(e);
        }
    }

    event ev; ev.y = 1000000; U.push(ev);          // zarážky
    usecka u;
    u.y2 = u.x1 = u.x2 = 1000000; u.y1 = -1000000; S.insert(u);
    u.y1 = u.x1 = u.x2 = -1000000; u.y2 = 1000000; S.insert(u);

    while (1) {
        event e = U.top(), f;
        ratio y = e.y;                             // nová poloha zametací přimky
        if (y == ratio(1000000)) break;            // zarážka => konec
        while (y == e.y && e.type == 0) {          // nové přímky
```

```
            pridej(e.u);
            U.pop(); e=U.top();                 // další událost
        }

        int k = 0;
        while (y == e.y && e.type == 1) {       // průsečiky
            iter it = S.find(e.u), it2;
            if (it != S.end()) {
                // smažeme vše, co se mění
                ratio x = it->x(y);
                fprintf(fo, "%f %f\n", x.f(), y.f());
                while (it->x(y) == x) it--;
                it++;
                while (it->x(y) == x) {
                    it2 = it; it2++;
                    prk[k++] = *it;             // a zapamatujeme si, co se mění
                    S.erase(it);
                    it = it2;
                }
            }
            U.pop(); e = U.top();
        }

        while (p == e.y && e.type == 2) {       // vodorovné přímky
            usecka u;
            u.y1 = -1000000; u.y2 = 1000000;
            u.x1 = u.x2 = e.u.x1;
            iter i = S.lower_bound(u);
            for (iter i = S.lower_bound(u); i->x(y) <= e.u.x2; i++)
                fprintf(fo, "%f %f\n", i->x(y).f(), y.f());
            U.pop(); e = U.top();
        }

        p=y;                                    // posuneme zametací přímku

        for (i=0; i<k; i++)                     // vrátíme změněné
            pridej(prk[i]);

        while (y == e.y && e.type == 3) {       // konce přímek
            iter it = S.find(e.u), it2 = it;
            it++; it2--;
            pronika(*it, *it2);
            S.erase(e.u);
            U.pop(); e = U.top();
        }
    }
    return 0;
}
```

P-III-2 Kouzelník Pecivális

Hledáme nejdelší souvislou podposloupnost čísel, jejichž součet je násobkem K, jinými slovy modulo K je roven nule. Pokud by ona podposloupnost měla začínat na začátku zadané posloupnosti, mohli bychom si jednoduše při čtení počítat průběžný

součet modulo K od začátku až do aktuálního místa a ve chvíli, kdy by byl roven nule, si vždy poznamenat zatím nejlepší nalezený konec.

Zbývá se jen vypořádat s možností začínat na libovolném místě. Budeme postupovat obdobně – počítat si průběžně součet modulo K a pamatovat si zatím nejlepší nalezenou podposloupnost. Pokud jsme na pozici j a součet $a_1 + a_2 + \ldots + a_j \equiv s \pmod{K}$ a zároveň známe nějaký index $i \leq j$ takový, že $a_1 + a_2 + \ldots + a_{i-1} \equiv s \pmod{K}$, pak zřejmě $a_i + a_{i+1} + \ldots + a_j \equiv 0 \pmod{K}$. Je-li i zároveň nejmenší možné, lepší (tedy delší) podposloupnost končící na pozici j nemůže existovat a už se jen podíváme, zda je tato delší než nejlepší zatím nalezená.

A jak najít pro každé j nejmenší takový index i? Stačí si uvědomit, že na příslušné pozici jsme již byli a znali součet $a_1 + a_2 + \ldots + a_{i-1} \pmod{K}$. Pokud tedy narazíme na index j takový, že $a_1 + a_2 + \ldots + a_{j-1} \equiv s \pmod{K}$ a $a_1 + a_2 + \ldots + a_{j'-1} \not\equiv s \pmod{K}$ pro všechna $j' < j$, tak si hodnotu j uložíme do pomocného pole velikosti K na index s; neexistenci indexu $j' < j$ poznáme tak, že s-tá hodnota v uvažovaném pomocném poli dosud nebyla inicializována.

Algoritmus postupně prochází všech N prvků posloupnosti na vstupu a u každého stráví konstantně dlouhou dobu (test existence indexu $j' < j$, uložení do pomocného pole, výpočet rozdílu $j - i$ a případné nahrazení nově nalezené optimální hodnoty). To trvá $\mathcal{O}(N)$, ovšem ještě potřebujeme inicializovat pomocné pole, takže celková časová složitost vyjde $\mathcal{O}(N + K)$. Pamatovat si budeme jen možné indexy i posloupností pro každý z K možných součtů s a několik dalších pomocných (konstantně velkých) údajů – paměťová složitost tedy bude $\mathcal{O}(K)$.

```
#include <stdio.h>

#define K_MAX 50000

int N, K;                // hodnoty N a K ze vstupu
int zacatky[K_MAX];      // nejmenší možné indexy i s daným součtem s modulo K
int a, soucet = 0;       // pomocné proměnné - načtení vstupu a mezisoučet
int nej_zacatek = 0;     // dosud nejlepší nalezené řešení
int nej_delka = 0;

int main(void) {
    scanf("%d%d", &N, &K);
    zacatky[0] = 1;
    for (int i = 1; i <= N; i++) {
        scanf("%d", &a);
        soucet = (soucet + a) % K;
        if (zacatky[soucet]) {
            if (i - zacatky[soucet] + 1 > nej_delka) {
                nej_zacatek = zacatky[soucet];
                nej_delka = i - nej_zacatek + 1;
            }
        } else
            zacatky[soucet] = i+1;
    }
    if (nej_delka)
        printf("%d %d\n", nej_zacatek, nej_zacatek + nej_delka - 1);
```

```
    else
        printf("Nelze zaklínat.\n");
    return 0;
}
```

P-III-3 Stackal

Tato úloha je podobná příkladu ve studijním textu, pouze místo výskytů dvou znaků počítá výskyty čtyř znaků. Můžeme o ní tedy uvažovat obdobně. Ihned se nabízí řešení pomocí čtyř zásobníků v lineárním čase: každé písmeno budeme ukládat do jeho zásobníku a na konci vybíráním po čtveřicích ověříme, že všechny zásobníky obsahovaly stejný počet písmen. Také můžeme jeden zásobník ušetřit tím, že si budeme pamatovat jen rozdíly počtů $a - b$, $a - c$ a $a - d$.

Existuje ovšem i dvouzásobníkové řešení, které do jednoho zásobníku zakóduje všechna čtyři počítadla a druhý zásobník používá jako pomocný. To si předvedeme.

Nejprve si rozmysleme, jak bude fungovat jedno počítadlo. Počet budeme ukládat po číslicích, ovšem ve dvojkové soustavě. Chceme-li počítadlo zvýšit o jedničku, budeme postupovat podle klasického algoritmu pro sčítání čísel po cifrách. Číslo budeme procházet po číslicích zprava doleva (od nejnižšího řádu k nejvyššímu) a přepisovat přitom jedničky na nuly. Jakmile narazíme na první nulu, přepíšeme ji na jedničku a zastavíme se (například $\mathbf{110011 + 1 = 110100}$). Kdyby číslo skončilo dříve, domyslíme si vlevo od něj další nulu ($\mathbf{111 + 1 = 0111 + 1 = 1000}$).

Jak tento postup naprogramovat pomocí zásobníků? Počítadlo uložíme na zásobník tak, aby nejpravější cifra ležela na vrcholu zásobníku. Můžeme tedy postupně odebírat jednotlivé číslice zprava, přepisovat je a odkládat do pomocného zásobníku. Až budeme hotovi, přesuneme je z pomocného zásobníku zpět do hlavního. Ve Stackalu to můžeme zapsat následovně:

```
procedure zvys(var a:stack of 0..1);    { zvyš počítadlo v zásobníku 'a' o 1 }
var b: stack of 0..1;                   { pomocný zásobník }
begin
   repeat
      if empty(a) then x := 0           { vlevo od čísla si domyslíme jedničky }
                  else x := pop(a);
      push(b, (x+1) mod 2);             { 0->1, 1->0 }
   until x=0;                           { zarazíme se o první nulu }
   while not empty(b) do                { přesypeme číslice zpět }
      push(a, pop(b));
end;
```

Čtyři dvojková počítadla nyní můžeme uložit „přes sebe“ na jeden zásobník. Každá položka zásobníku bude obsahovat čtyřbitové číslo, jehož i-tý bit bude představovat příslušnou cifru i-tého počítadla. Při zvyšování jednoho počítadla tedy budeme přepisovat jen příslušné bity a ostatní bity (patřící jiným počítadlům) zachováváme.

K práci s bity se velice hodí bitové operace `and` a `xor`. Výraz a `and` b dává přirozené číslo, v jehož dvojkovém zápisu je na i-tém místě jednička právě tehdy,

když je jednička na i-tém místě v číslech a i b. Podobně a `xor` b má jedničky tam, kde byla jednička buď v a, nebo v b, ale ne v obou. Proto a `and` 8 dává nulu právě tehdy, když je na čtvrté pozici zprava v čísle a nula (8 je dvojkově **1000**), zatímco a `xor` 8 číslici na této pozici změní z nuly na jedničku a naopak. (Také bychom se bez těchto operací mohli obejít a čísla kódovat a dekódovat tabulkami, konec konců musíme ošetřit jen 16 možností, ale to by bylo poněkud pracné.)

Celý program tedy bude vypadat takto:

```
program abcd;
var a, b: stack of 0..15;              { zásobník s počítadly, pomocný zásobník }
    c: char;                           { právě zpracovávaný znak }
    m, x: 0..15;                       { pomocné proměnné }

begin
   while read(c) do begin
      case c of                        { který bit odpovídá načtenému znaku? }
         'a': m=1;
         'b': m=2;
         'c': m=4;
         'd': m=8;
         end;
      repeat                           { zvýšení o jedničku }
         if empty(a) then x := 0
                     else x := pop(a);
         push(b, x xor m);
      until (x and m) = 0;
      while not empty(b) do            { kopírujeme z 'b' zpět do 'a' }
         push(a, pop(b));
      end;

   m := 1;                             { zkontrolujeme, zda se počítadla rovnají }
   while not empty(a) do begin
      x := pop(a);
      if (x<>0) and (x<>15) then       { jsou všechny bity 0 nebo všechny 1? }
         m := 0;
      end;
   write(m);
end.
```

Program používá pouze dva zásobníky a spotřebuje paměť $\mathcal{O}(\log N)$, protože každé číslo menší nebo rovné N má ve dvojkové soustavě nejvýše $\lfloor \log_2 N \rfloor + 1$ číslic. Délka čísla také omezuje počet opakování cyklu repeat pro každý znak, takže časová složitost není horší než $\mathcal{O}(N \log N)$. Ukážeme ovšem, že je lineární.

Rozbor stačí provést pro jedno počítadlo (tedy pro naši ukázkovou proceduru `zvys`), protože jednotlivá počítadla se navzájem neovlivňují a závěrečné porovnání trvá pouze $\mathcal{O}(\log N)$. Navíc se stačí zaměřit na cyklus repeat v této proceduře, neboť přesouváním číslic z pomocného zásobníku v cyklu while zjevně trávíme nejvýše tolik času, jako když jsme je tam ukládali.

Uvažujme, co se stane, když proceduru `zvys` zavoláme N-krát po sobě na zpočátku nulové počítadlo. V každém průchodu cyklu repeat přepíšeme jednu číslici

ve dvojkovém zápisu počítadla, tudíž časová složitost je přímo úměrná tomu, kolikrát došlo ke změně číslice.

Sledujme, jak se v průběhu výpočtu mění celkový počet jedniček v počítadle. Když v cyklu přepíšeme **0** na **1**, jedna jednička přibude, v opačném případě jedna ubude. Navíc ihned po přepsání **0** na **1** se cyklus zastaví, takže se za celou dobu výpočtu počet jedniček zvýší nejvýše o N. Počet jedniček ovšem také nikdy neklesne pod nulu, takže operací, které ho snižují, může být rovněž nejvýše N. Všech operací je proto $\mathcal{O}(N)$.