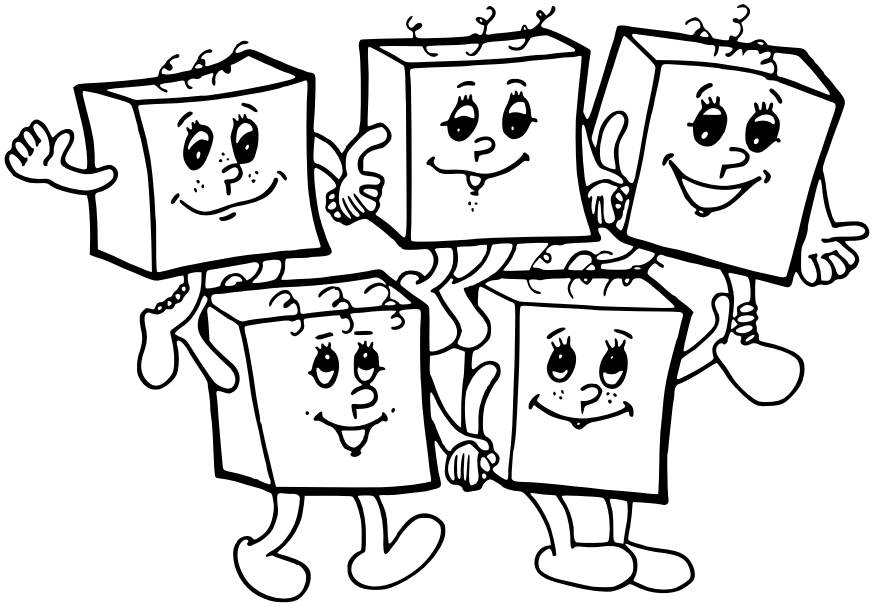
OLYMPIÁDA V INFORMATIKE
NA STREDNÝCH ŠKOLÁCH



dvadsiaty štvrtý ročník
školský rok 2008/09

riešenia celoštátneho kola
kategória A

1. súťažný deň

- **Olympiáda v informatike** je od školského roku 2006/07 samostatnou súťažou. Predchádzajúcich 21 ročníkov tejto súťaže prebiehalo pod názvom **Matematická olympiáda, kategória P** (programovanie).
- Oficiálnu **webstránku** súťaže nájdete na `http://oi.sk/`.

Riešenia kategórie A

A-III-1 Horár Jedlička II

Úlohu vyriešime metódou *zametania*, ktorá sa používa často pre geometrické problémy: Predstavíme si priamku p rovnobežnú s osou x, ktorá sa pohybuje v smere osi y (od záporných čísel ku kladným – tomuto smeru budeme hovoriť zdola nahor). Budeme udržiavať prienik tejto priamky so zadanými úsečkami. Pre tento účel budeme používať nasledovné dátové štruktúry:

- Zoznam úsečiek S pretínajúcich v danom okamihu priamku p, v poradí podľa x-ovej súradnice týchto priesečníkov. V tomto zozname budeme potrebovať rýchlo vyhľadávať, preto ho budeme realizovať ako (vyvažovaný) vyhľadávací strom. Za zmienku stojí, že si v tejto dátovej štruktúre nepamätáme súradnice priesečníkov s priamkou p. Tieto súradnice sa menia, keď sa priamka p posunie, a ich upravovanie by bolo príliš pomalé. Namiesto toho si pamätáme iba poradie týchto priesečníkov. Toto poradie sa mení iba keď p prejde jedným z priesečníkov úsečiek.

  Naviac vieme, že keď takáto situácia nastane, iba sa obráti poradie priesečníkov priamky p s úsečkami, ktoré sa v danom bode pretínali. Súradnice priesečníkov s p dopočítavame až v priebehu vyhľadávania z aktuálnej y-ovej súradnice p.

  Tiež si uvedomte, že pre ľubovoľný bod priamky p vieme v čase $O(\log N)$ zistiť, medzi ktorými dvoma úsečkami sa nachádza.

- Fronta udalostí U, ktoré ovplyvňujú prienik p s úsečkami:

  – Spodné konce úsečiek, ktoré sú celé nad p.
  – Priesečník každej dvojice úsečiek susediacich v S, ak existuje a leží nad p.
  – Vodorovné úsečky. Tie treba ošetriť špeciálne.
  – Horné konce úsečiek, ktoré sú nad p.

  Frontu udalostí môžeme realizovať napr. ako haldu. (Prípadne môžeme opäť použiť vyvažovaný binárny strom.) Udalosti porovnávame podľa ich y-ovej súradnice a v prípade zhody podľa poradia v predošlom zozname.

Na začiatku je priamka p v $-\infty$, teda zoznam S je prázdny a fronta U obsahuje horné a dolné konce všetkých úsečiek. Pre zjednodušenie niektorých operácií pridajme do S dve falošné úsečky rovnobežné s osou y, v $-\infty$ a $+\infty$ (alebo v ľubovoľných bodoch ležiacich naľavo a napravo od všetkých ostatných úsečiek). V priebehu simulácie odoberáme z U najmenšiu udalosť a upravujeme S a U podľa nej. Program sa končí vo chvíli, keď priamka p dorazí do $+\infty$, teda keď je U prázdna.

Vždy, keď do S pridáme prvok, pridáme tiež do U udalosť prieniku so susednými úsečkami, pokiaľ prienik existuje a leží nad p.

Popíšeme teraz detailnejšie, ako spracovávame udalosť spôsobenú úsečkou u. Udalosti s rovnakou y-ovou súradnicou spracovávame v tomto poradí:

- *Spodný koniec úsečky u:* Pridáme u do S tak, aby prieniky jednotlivých úsečiek boli stále usporiadané.
- *Vodorovná úsečka $u = (x_1, y) - (x_2, y)$:* Vypíšeme priesečníky s úsečkami z S, ktorých x-ová súradnica na priamke p je z intervalu $\langle x_1, x_2 \rangle$.
- *Priesečník dvoch úsečiek v bode x, y:* Zmažeme z S všetky úsečky prechádzajúce bodom (x, y) a dáme ich nabok. Vypíšeme (x, y).
- S je teraz utriedené podľa priesečníkov s priamkou p (môže sa zmeniť iba poradie tých úsečiek, ktoré sa pretínajú na p, ale tie sme dali nabok). Do S teraz zatriedime úsečky, ktoré sme dali nabok.
- *Horný koniec úsečky u:* Nech u leží medzi úsečkami u_1 a u_2. Odstránime u z S. Naviac, pokiaľ sa u_1 a u_2 pretínajú nad p, pridáme ich priesečník do U.

Treba ešte doriešiť tento technický detail: *zaokrúhľovacie chyby* – pokiaľ by spôsobili zámenu poradia dvoch udalostí v U alebo dvoch úsečiek v S, mohlo by to viesť k zlému výsledku. Preto je nutné všetky porovnania robiť presne. To sa dá dosiahnuť viacerými spôsobmi, jednou z možností je počítať súradnice priesečníkov ako zlomky.

Pamäťová zložitosť je lineárna – ako U tak S vždy obsahujú $O(N)$ prvkov. Aká je časová zložitosť? Nech P je počet priesečníkov úsečiek. Každá operácia s U či S zaberie čas $O(\log N)$. Spočítajme teraz počet týchto operácií pre každú z udalostí:

- *Spodný alebo horný koniec úsečky:* Týchto udalostí je $O(N)$ a vyžadujú konštantný počet operácií s U a S. Každá z nich spôsobuje pridanie nanajvýš dvoch udalostí do U.
- *Priesečník, v ktorom sa pretína k úsečiek:* Týchto udalostí je P a vyžadujú $O(k)$ operácií s S. Každý priesečník spôsobí pridanie nanajvýš dvoch a odobratie aspoň $k-1$ udalostí z U.

Počet udalostí pridaných do U (vrátane inicializácie) je $O(N+P)$ a počet odobraných udalostí je teda tiež $O(N+P)$. Počet operácií s S a U je obmedzený počtom udalostí pridaných či odobraných z U a je $O(N+P)$. Časová zložitosť je preto $O((N+P)\log N)$. Na záver uveďme, že existuje aj rýchlejšie (a podstatne zložitejšie) riešenie s časovou zložitosťou $O(N \log N + P)$ (Chazelle a Edeslbrunner, 1992).

Listing programu:

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

// Nejvetsi spolecny delitel dvou cisel
long long gcd(long long a, long long b) { return (b==0) ? a : gcd(b, a%b); }

// Zlomky a prace s nimi
struct fraction {
    long long nom, den;                                     // citatel a jmenovatel
    fraction () { nom=0; den=1; }                           // default je nula
    fraction(long long n) { nom=n; den=1; }                 // cele cislo -> zlomek
    fraction(long long n, long long d) {                    // uprava zlomku na zakladni tvar
        long long g = gcd(llabs(den), llabs(nom));
        nom=n/g; den=d/g;
        if (den<0) { nom=-nom; den=-den; }
    }
    fraction& operator =(long long b)  { nom=b; den=1; return *this; }
    fraction operator +(fraction b) const { return fraction(nom*b.den+b.nom*den, den*b.den); }
    fraction operator -(fraction b) const { return fraction(nom*b.den-b.nom*den, den*b.den); }
    fraction operator *(fraction b) const { return fraction(nom*b.nom, den*b.den); }
    fraction operator /(fraction b) const { return fraction(nom*b.den, den*b.nom); }
    bool operator  <(fraction b) const { return (*this-b).nom < 0; }
    bool operator ==(fraction b) const { return nom==b.nom && den==b.den; }
    bool operator !=(fraction b) const { return !(*this == b); }
    bool operator <=(fraction b) const { return (*this-b).nom <= 0; }
    double f() const { return (nom+0.0)/den; }              // konverze na realne cislo
};

fraction p(-900000);                                        // zametaci primka

// Usecka ze vstupu
struct usecka {
    fraction x1, y1, x2, y2;                                // souradnice koncu
    bool prunik(usecka pr) const {                          // ma usecka prunik s jinou?
        fraction r = intersect(pr);
        return !(r<pr.y1 || pr.y2<r || r<y1 || y2<r);
    }
    fraction intersect(usecka pr) const {                   // y-ova souradnice pruniku
        return (pr.dir() == dir()) ? 1000000 :
               (x(fraction(0)) - pr.x(fraction(0))) / (pr.dir() - dir());
    }
    fraction x(fraction y) const { return x1+dir()*(y-y1); }  // x-ova souradnice pro y=p
    fraction dir() const { return (x2-x1)/(y2-y1); }        // smernice
    bool operator < (const usecka &pr) const {              // porovnani usecek dle pruseciku s p
        if (x(p) != pr.x(p)) return x(p) < pr.x(p); else return dir() < pr.dir();
    }
```

```
};

// Udalost
struct event{
    fraction y;                                     // kdy nastava
    int type;   // typ: 0 zacatek, 1 krizeni, 2 vodorovna primka, 3 konec
    usecka u;
    bool operator < (const event &e) const {
        if (e.y != y) return e.y < y; else return type > e.type;
    }
};

priority_queue <event> U;                           // fronta udalosti
set <usecka> S;                                     // zadane usecky
typedef set <usecka>::iterator iter;                // iterator pres usecky
int n;                                              // kolik je usecek
usecka prk[1000000];                                // pomocne pole na zmenene primky

// Pokud se a s b pronika nad zametaci primkou, vytvorime udalost.
void pronika(usecka a, usecka b) {
    if (!a.prunik(b)) return;
    event f; 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;                    // iterator na pridany prvek
    it--; pronika(*it, u);
    it++; it++; pronika(u, *it);
}

int main() {
    int x1, y1, x2, y2, i;
    scanf("%i", &n);
    for (i=0; i<n; i++){
        usecka u;
        scanf("%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);           // zarazky
    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;
        fraction y = e.y;                           // nova poloha zametaci primky
        if (y == fraction(1000000)) break;          // zarazka => konec
        while (y == e.y && e.type == 0) {           // nove primky
            pridej(e.u);
            U.pop(); e=U.top();                     // dalsi udalost
        }

        int k = 0;
        while (y == e.y && e.type == 1) {           // pruseciky
            iter it = S.find(e.u), it2;
            if (it != S.end()) {
                // smazeme vse, co se meni
                fraction x = it->x(y);
                printf("%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 meni
                    S.erase(it);
                    it = it2;
                }
            }
            U.pop(); e = U.top();
        }

        while (p == e.y && e.type == 2) {       // vodorovne primky
            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++)
                printf("%f %f\n", i->x(y).f(), y.f());
            U.pop(); e = U.top();
        }

        p=y;                                    // posuneme zametaci primku
        for (i=0; i<k; i++) pridej(prk[i]);     // vratime zmenene
        while (y == e.y && e.type == 3) {       // konce primek
            iter it = S.find(e.u), it2 = it;
            it++; it2--;
            pronika(*it, *it2);
            S.erase(e.u);
            U.pop(); e = U.top();
        }
    }
    return 0;
}
```

A-III-2 Magické cestovanie

Našou úlohou je nájsť najdlhšiu súvislú podpostupnosť zadanej postupnosti nezáporných celých čísel, ktorej súčet je násobok K. Inými slovami, hľadáme najdlhšiu súvislú podpostupnosť, ktorej súčet dáva po delení číslom K zvyšok 0. V celom tomto riešení budeme všetky matematické operácie robiť modulo K, teda ak napríklad $K = 7$, tak súčet postupnosti $2, 4, 2$ je 1.

Predstavme si najskôr, že hľadaná podpostupnosť môže začínať len na začiatku. Potom by sa nám úloha riešila jednoducho – postupne by sme spracúvali čísla v postupnosti a počítali si ich súčet. Zakaždým, keď by sme dostali súčet 0, našli sme potenciálne riešenie. Keďže chceme najdlhšiu takú podpostupnosť, výsledkom by bolo posledné takéto miesto.

Zostáva už len toto riešenie zovšeobecniť na podpostupnosť začínajúcu na ľubovoľnom mieste. Na to stačí spraviť nasledujúce pozorovanie. Nech sme práve spracovali j-te číslo v postupnosti a vieme, že súčet prvých j čísel je d. Ak pre nejaké $i < j$ bol tiež súčet prvých i čísel rovný d, tak vieme, že podpostupnosť $a_{i+1}, \ldots, a_j$ musí mať súčet 0, a teda je možným riešením úlohy.

A to už je v podstate všetko, stačí si už len uvedomiť, že ak máme pre i viacero možností, chceme vybrať tú najmenšiu z nich – teda miesto, kde bol priebežne počítaný súčet čísel po prvý krát rovný d.

Aby sme takéto i vedeli efektívne nájsť, stačí si spraviť pole dĺžky K a v ňom si pre každý súčet $d, 0 \leq d < K$ zaznamenať, kde sa nachádzal prvý taký index i, pre ktorý sme dostali súčet d.

Na začiatku algoritmu musíme toto pole inicializovať – na všetky políčka si zapíšeme špeciálnu hodnotu (napr. -1), ktorá nám hovorí, že sa takýto súčet ešte nevyskytol. Následne stačí načítavať postupnosť, počítať priebežný súčet a zisťovať v poli, či sme už na daný súčet natrafili. Každý prvok postupnosti teda spracujeme v konštantnom čase. Celková časová zložitosť postupu je teda $O(N + K)$. Keďže si postupnosť nepotrebujeme pamätať, vystačíme si s pamäťou $O(K)$.

Malé cvičenie na záver: Vedeli by ste dokázať, že ak $N \geq K + 1$, tak určite existuje aspoň jedno riešenie?

Listing programu:

```
#include <cstdio>
using namespace std;
int N, K;                           // hodnoty N a K ze vstupu
int zacatky[50007];                 // nejmensi mozne indexy i s danym souctem s modulo K
int a, soucet = 0;                  // pomocne promenne - nacteni vstupu a mezisoucet
int nej_zacatek = 0, nej_delka = 0; // dosud nejlepsi nalezene reseni
```

```
int main() {
    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 zaklinat.\n");
}
```

A-III-3 Zásobníkové počítače

Najjednoduhším riešením tejto úlohy bolo použiť 4 zásobníky, každý pre jeden znak, a na konci zistiť, či sú všetky zásobníky rovnako zaplnené. Po troške rozmýšľania sa dalo spraviť riešenie s troma zásobníkmi, pričom si v nich pamätáme len rozdiely počtov znakov: $a - b$, $a - c$, $a - d$. Všetky tieto riešenia sa dali naprogramovať s lineárnou časovou zložitosťou.

Použiť tri zásobníky je zbytočný luxus – totiž platí, že úplne čokoľvek, čo sa dá spraviť s použitím K zásobníkov, sa dá spraviť s použitím len 2 zásobníkov. Ukážeme teraz dve riešenia, ktoré potrebujú iba dva zásobníky.

Kvadratické riešenie

Najskôr si načítame celý vstup do prvého zásobníka. Potom pomocou druhého zásobníka vymažeme z prvého zásobníka jeden znak a, jeden znak b, jeden znak c a jeden znak d. Toto budeme opakovať, až kým nebude prvý zásobník prázdny, alebo neminieme niektorý znak. Toto riešenie má zjavne kvadratickú časovú zložitosť.

Lineárne riešenie

Ukážeme si teraz riešenie, ktoré bude používať dva zásobníky a bude mať lineárnu časovú zložitosť. Prvý zásobník bude fungovať ako štyri počítadlá a druhý zásobník bude pomocný. Každé počítadlo bude počítať počet výskytov jedného znaku. Najskôr si popíšeme, ako bude fungovať jedno počítadlo a potom si povieme, ako spojiť 4 počítadlá do jediného zásobníka.

Počítadlá budú binárne (číslo si budeme pamätať v dvojkovej sústave). Keď budeme chcieť počítadlo zvýšiť o jedna, tak použijeme klasický algoritmus na sčítanie čísliel po cifrách. Budeme prechádzať po čísliciach od najnižších bitov k najvyšším a prepisovať jednotky na nuly. Keď narazíme na prvú nulu, tak ju zmeníme na jednotku a skončíme (napríklad $110011 + 1 = 110100$). Ak by číslo skončilo skôr ako narazíme na nulu, tak na jeho začiatok dopíšeme ešte jednu jednotku ($111 + 1 = 1000$).

Ako tento postup naprogramovať pomocou zásobníkov? Počítadlo uložíme do zásobníka tak, že najpravejšia (najnižší bit) cifra bude na vrchole zásobníka. Môžeme postupne odoberať jednotlivé číslice sprava, prepisovať ich a odkladať ich do pomocného zásobníka. Keď budeme hotoví, tak presunieme obsah pomocného zásobníka na hlavný zásobník. Kód by vyzeral nasledovne:

Listing programu:

```
procedure zvys(var a:stack of 0..1); { zvys pocitadlo v zasobniku  a  o 1 }
var b: stack of 0..1;                { pomocny zasobnik }
begin
   repeat
      if empty(a) then x := 0 else x := pop(a);  { ak treba, vlavo od cisla si domyslime nuly }
      push(b, (x+1) mod 2);                      { 0->1, 1->0 }
   until x=0;                                    { zastavime na prvej nule }
   while not empty(b) do push(a, pop(b));        { presypeme cisla spat }
end;
```

Ako uložiť 4 počítadlá do jedného zásobníka? Jedna možnosť je ukladať tam ich bity „na striedačku“. Druhá možnosť je použiť čísla z väčšieho rozsahu ako `0..1`. My ukážeme tú druhú z nich.

Do zásobníka budeme ukladať štvorbitové čísla, pričom i-te počítadlo bude zodpovedať i-temu najmenej významnému bitu každého z čísiel v zásobníku. Pri zvyšovaní počítadla budeme prepisovať len príslušné bity, ostatné bity (patriace iným počítadlám) necháme na pokoji.

Na prácu s bitmi sa nám budú hodiť operácie **and** a **xor**. Výraz a **and** b dáva prirodzené číslo, ktoré má v dvojkovom zápise na i-tom mieste 1 práve vtedy, ak aj a aj b majú na i-tom mieste 1. Podobne a **xor** b má jednotky tam, kde bola jednotka buď v a, alebo v b, ale nie v oboch naraz. Napríklad a **and** 8 je nula práve vtedy, ak na štvrtej pozícii čísla a je 0 a ináč je to 8. Číslo a **xor** 8 je číslo, v ktorom je oproti a zmenený štvrtý bit na opačný.

Celý program bude vyzerať nasledovne:

Listing programu:

```
program abcd;
var a, b: stack of 0..15; { zasobnik s pocitadlami, pomocny zasobnik }
    c: char;              { prave zpracovavany znak }
    m, x: 0..15;          { pomocne promenne }
begin
   while read(c) do begin
      if c='a' then m:=1        { ktory bit zodpoveda nacitanemu znaku? }
      else if c='b' then m:=2
      else if c='c' then m:=4
      else if c='d' then m:=8;
      repeat                    { zvysenie pocitadla o jedna }
         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 push(a, pop(b)); { kopirujeme z "b" spat do "a" }
   end;
   m := 1;
   while not empty(a) do begin
      x := pop(a);
      if (x<>0) and (x<>15) then m := 0;
   end;
   write(m);
end.
```

Program používa len dva zásobníky a spotrebúva pamäť $O(\log N)$, lebo každé číslo menšie alebo rovné N má v dvojkovej sústave najviac $\lfloor \log_2 N \rfloor + 1$ číslic. Dĺžka čísla tiež obmedzuje počet opakovaní cyklu repeat, takže časová zložitosť nebude horšia ako $O(N \log N)$. Teraz si ukážeme, že časová zložitosť je v skutočnosti lineárna.

Rozbor stačí spraviť pre jedno počítadlo (teda pre našu ukážkovú procedúru `zvys`), keďže jednotlivé počítadlá sa navzájom neovplyvňujú a záverečné porovnanie trvá len $O(\log N)$. Naviac sa stačí zamerať len na cyklus repeat, lebo v cykle while trávime najviac toľko času ako v cykle repeat.

Uvažujme, čo sa stane, ak procedúru **zvys** zavoláme N krát po sebe, pričom na začiatku bolo počítadlo nastavené na 0. Sledujme, ako sa pri tom mení počet jednotiek v dvojkovom zápise počítadla. Operácie vo vnútri počítadla buď prepisujú 0 na 1 (pribudne jednotka), alebo 1 na 0 (jednotka ubudne). Naviac prepísaním 0 na 1 sa cyklus zastaví, takže po celú dobu výpočtu sa počet jednotiek zvýši najviac o N. Počet jednotiek ale nikdy neklesne na 0, takže operácii, ktoré ho znižujú nebude nikdy viac ako N. Preto je všetkých operácii $O(N)$.

Iná cesta ako dokázať lineárnosť nášho algoritmu: Keď si vypíšeme, koľko krokov spraví algoritmus počítajúci $N+1$ pre $N = 0, 1, 2, \ldots$, dostaneme nasledujúcu postupnosť: $1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 5, \ldots$ Časová zložitosť nášho programu je priamo úmerná súčtu tejto postupnosti. Ten vieme odhadnúť nasledovne: každý člen je aspoň 1, každý druhý je aspoň 2, každý štvrtý je aspoň 3, atď., takže celkový súčet je rádovo rovný $N + N/2 + N/4 + \cdots = 2N$.

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY ŠTVRTÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX