58. ročník Matematické olympiády – 2008/2009

Řešení úloh domácího kola kategorie P

P-I-1 Učebnice

Zadanou úlohu je možné řešit mnoha různými způsoby, které se skládají ze dvou fází: načítání slov textu a hledání jejich náhrad ve slovníku. Protože načítání slov je pouze technická záležitost, nebudeme se jí v popisu řešení zabývat a pouze odkážeme na vzorové programy.

Mnohem zajímavější je problém převodu slova na jeho náhradu. Aby byl popis srozumitelný, budeme v dalším textu zadaný seznam dvojic označovat jako *slovník* a vhodnější náhradu za slovo označíme jako jeho *překlad.*

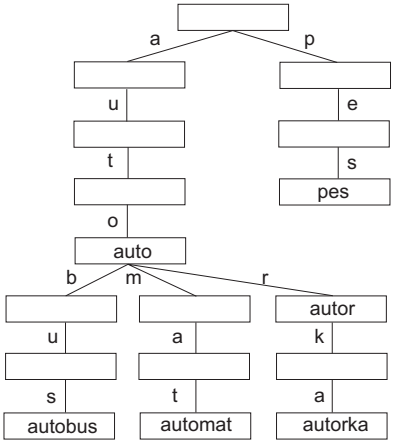
Zřejmě nejjednodušší přístup je načtení celého slovníku do pole dvojic a pro každé slovo ze vstupu toto pole prohledat. Bohužel časová složitost tohoto postupu je $\mathcal{O}(N \cdot S \cdot P)$, kde S je počet slov ve slovníku, N počet slov na vstupu a P je čas nutný na porovnání dvou slov. Pokud bychom slovník lexikograficky setřídili, mohli bychom slova vyhledávat metodou půlení intervalů, což by časovou složitost snížilo na $\mathcal{O}(N \cdot \log S \cdot P)$. V obou případech závisí paměťová složitost lineárně na velikosti seznamu dvojic slov. Další možností by bylo vytvoření vyhledávacího stromu ze slov slovníku. Časová i paměťová složitost takového postupu by byla stejná jako v předcházejícím případě.

My si ale ukážeme strukturu, která umožňuje vyhledávat v množině slov M v čase, který závisí pouze na délce hledaného slova bez ohledu na velikost množiny M. Tato datová struktura se nazývá prefixový strom (nebo též trie) a myšlenkově vychází z vyhledávacích stromů.

Trie je totiž strom, jehož každý uzel má nejvýše L následníků, kde L je počet písmenek, která se ve slovech mohou vyskytovat. V našem případě je L rovno 26, neboť právě tolik je písmen anglické abecedy. Následníci každého uzlu jsou pak pojmenováni přímo těmito písmeny. Pro tento strom platí, že písmenka na cestě z kořenu trie do libovolného uzlu zachycují prefix (předponu) nějakého slova z množiny M. Na obrázku je ukázka této struktury se slovy ‘auto’, ‘autobus’, ‘automat’, ‘autor’, ‘autorka’ a ‘pes’.

Jak vypadá vyhledávání slova v této struktuře? Aktuálním uzlem bude kořen trie. Nyní vezmeme první písmenko hledaného slova a podíváme se na odpovídajícího následníka aktuálního uzlu. Pokud takový následník neexistuje, tak hledané slovo v množině není. V opačném případě se prohlásíme za aktuální uzel příslušného následníka, vezmeme další písmenko slova a celý postup zopakujeme. Takto pokračujeme, až zpracujeme postupně všechna písmena hledaného slova. Pokud jsme došli až do listu, tak hledané slovo ve struktuře bylo.

Tento přímočarý postup ale selže pro taková dvě slova, že jedno je prefixem druhého. Například pokud jsou ve slovníku slova ‘auto’ a ‘automat’ a my bychom hledali slovo ‘auto’, tak bychom došli do nějakého uzlu. Ten ale nebude listem, takže



bychom řekli, že slovo ‘auto’ ve slovníku nebylo. Tento problém vyřešíme jednoduše tak, že každý uzel rozšíříme o údaj, jestli v něm končí nějaké slovo z množiny nebo ne.

Vkládání slov do trie je podobné jeho vyhledávání. Jediný rozdíl spočívá v tom, že když zpracováváme určité písmenko vkládaného slova a neexistuje příslušný následník aktuálního uzlu, tak jej vytvoříme. Právě popsaná struktura se dá jednoduše rozšířit na hledání překladů slov. Nic nám totiž nebrání si v každém uzlu značícím konec slova pamatovat i jeho překlad.

Implementace řešení je již jednoduchá – každý uzel bude obsahovat 26 ukazatelů na své následníky přímo indexované písmenky anglické abecedy a proměnnou `preklad`. Pokud následník k určitému písmenku neexistuje, tak bude ukazovat na `nil`. Skutečnost, že v daném uzlu končí nějaké slovo ze slovníku, bude indikována tím, že proměnná `preklad` bude obsahovat jeho překlad. Časová složitost vložení nebo nalezení jednoho slova je $\mathcal{O}(D)$, kde D je délka slova. Časová složitost celého programu bude $\mathcal{O}(N + M)$, kde N je délka vstupního textu včetně seznamu dvojic a M je délka výstupního textu. Tato časová složitost je zjevně optimální, protože musíme načíst vstup a vypsat výsledek. Paměťová složitost je lineární k velikosti slovníku (součtu délek slov v jednotlivých dvojicích), což je opět zjevně, neboť si pro překlad textu všechny tyto dvojice musíme zapamatovat.

```
program uprava_ucebnice;

type
   ptUZEL = ^tUZEL;
   pPREKLAD = ^string;
   tUZEL = record
      preklad: pPREKLAD;
      potomci: array['A'..'Z'] of ptUZEL;
   end;

var
   i, N:  longint;
```

```
koren: tUZEL;
znak:  char;
ve_slove: boolean;
slovo: string;
preklad: string;
vstup: text;
vystup: text;

{ Pomocná funkce pro snažší manipulaci s konci řádků }
procedure cti_znak(var vstup: text; var znak: char);
begin
   if eoln(vstup) then begin
      znak := #10;      { Nový řádek vrátí jako znak #10 a přejde na nový řádek }
      Readln(vstup);
   end else
      Read(vstup, znak);
end;

procedure inicializuj_trii(var koren: tUZEL);
var
   znak: char;
begin
   for znak := 'A' to 'Z' do
      koren.potomci[znak] := nil;
   koren.preklad := nil;
end;

procedure vloz_do_trie(var koren: tUZEL; var klic, hodnota: string);
var
   i: integer;
   znak: char;
   uzel: ptUZEL;
begin
   uzel := @koren;
   for i := 1 to length(klic) do begin
      znak := klic[i];
      if uzel^.potomci[znak] <> nil then  { Pokud potomek existuje, přejdeme tam }
         uzel := uzel^.potomci[znak]
      else begin                          { Neexistující potomky vytváříme }
         new(uzel^.potomci[znak]);
         uzel := uzel^.potomci[znak];
         for znak := 'A' to 'Z' do        { Inicializace nového uzlu }
            uzel^.potomci[znak] := nil;
         uzel^.preklad := nil;
      end;
   end;
   new(uzel^.preklad);
   uzel^.preklad^ := hodnota;
end;

function najdi_v_trii(var koren: tUZEL; var klic, hodnota: string): boolean;
var
   i: integer;
   znak: char;
   uzel: ptUZEL;
```

```
   begin
      najdi_v_trii := False;

      uzel := @koren;
      for i := 1 to length(klic) do begin
         znak := klic[i];

         if uzel^.potomci[znak] <> nil then  { Buď existuje příslušný potomek }
            uzel := uzel^.potomci[znak]
         else                                { nebo slovo ve slovníku nebylo }
            exit;
      end;

      if uzel^.preklad = nil then
         exit;

      hodnota := uzel^.preklad^;
      najdi_v_trii := True;
   end;

   procedure smaz_uzel(uzel : ptUZEL);
   var
      znak   : char;
   begin
      if uzel<>nil then begin
         for znak:='A' to 'Z' do
            smaz_uzel(uzel^.potomci[znak]);  { Rekurzivně smažeme celý strom }
         dispose(uzel);
      end;
   end;

   procedure smaz_trii(var koren : tUZEL);   { Odstraní trii z paměti }
   var
      znak   : char;
   begin
      for znak:='A' to 'Z' do
         smaz_uzel(koren.potomci[znak]);
   end;

begin
   Assign(vstup, 'ucebnice.in');
   Reset(vstup);
   Assign(vystup, 'ucebnice.out');
   Rewrite(vystup);

   Readln(vstup, N);

   inicializuj_trii(koren);

   for i := 1 to N do begin
      slovo := '';
      Read(vstup, znak);
      while znak <> ' ' do begin             { Přidáme slovo do trie }
         slovo := slovo + znak;
         Read(vstup, znak);
```

```
      end;
      Readln(vstup, preklad);
      vloz_do_trie(koren, slovo, preklad);
   end;

   slovo := '';
   ve_slove := False;
   while not eof(vstup) do begin             { Čteme až do konce souboru }
      cti_znak(vstup, znak);

      if znak in ['A'..'Z'] then begin
         if ve_slove then begin              { Pokud jsme ve slově, ... }
            slovo := slovo + znak;           { ... přidáme k němu další znak }
         end else begin                      { Jinak začneme nové slovo }
            slovo := znak;
         end;
         ve_slove := True;
      end else begin
         if ve_slove then begin              { Na konci slova }
            if najdi_v_trii(koren, slovo, preklad) then
               Write(vystup, preklad)        { vypíšeme buď náhradu slova }
            else
               Write(vystup, slovo);         { nebo původní slovo }
         end;
         if znak <> #10 then
            Write(vystup, znak)
         else
            Writeln(vystup);

         ve_slove := False;
      end;
   end;

   if ve_slove then begin                    { Soubor skončil slovem }
      if najdi_v_trii(koren, slovo, preklad) then
         Write(vystup, preklad)
      else
         Write(vystup, slovo);
   end;

   smaz_trii(koren);
   close(vstup);
   close(vystup);
end.

/* Učebnice -- řešení v C++ */

#include <cstdio>
#include <string>
#include <sstream>
#include <fstream>

using namespace std;
#define WORD_LEN 1024
```

```
struct TRIE_UZEL{
      TRIE_UZEL *potomci[26];
      string *hodnota;
      TRIE_UZEL()
      {
         for(int i=0; i<26; i++)
            potomci[i] = NULL;
         hodnota = NULL;
      }
      ~TRIE_UZEL() { delete hodnota; }
   };

class TRIE
{
private:
   TRIE_UZEL *koren_;
   void smaz(TRIE_UZEL *koren);
public:
   TRIE() { koren_ = new TRIE_UZEL; }
   ~TRIE() { smaz(koren_); }
   void pridej(string &slovo, string &hodnota);
   void najdi(string &slovo, string *&vysledek);
};

// Přidá nové slovo do trie
void TRIE::pridej(string &slovo, string &hodnota)
{
   TRIE_UZEL *akt_koren = koren_;
   for (string::iterator it=slovo.begin();it!=slovo.end();++it)
   {
      if (!akt_koren->potomci[*it-'A'])  // Pokud neexistuje potomek, vytvoříme ho
      {
         akt_koren->potomci[*it-'A'] = new TRIE_UZEL;
      }
      akt_koren = akt_koren->potomci[*it-'A']; //Jdeme do odpovídajícího potomka
   }
   // V koncovém uzlu cesty si uložíme překlad vloženého slova
   akt_koren->hodnota = new string(hodnota);
}

// Nalezne překlad slova, pokud existuje
void TRIE::najdi(string &slovo, string *&vysledek)
{
   TRIE_UZEL *akt_koren = koren_;
   for (string::iterator it=slovo.begin();it!=slovo.end();++it)
   {
      if (!akt_koren->potomci[*it-'A'])
      {
         // Pokud neexistuje potomek, slovo v trii není
         vysledek = NULL;
         return;
      }
      akt_koren = akt_koren->potomci[*it-'A'];
   }
   vysledek = akt_koren->hodnota;
```

```
}

// Rekurentně projdeme trii a smažeme z paměti všechny uzly
void TRIE::smaz(TRIE_UZEL *koren)
{
   if (!koren)
      return;
   for(int i=0; i<26; i++)
      smaz(koren->potomci[i]);
   delete koren;
}

int main()
{
   ifstream fin;              // Vstupní soubor
   fin.open("ucebnice.in");
   int N;
   fin >> N;

   TRIE trie;
   string klic, hodnota;
   getline(fin, hodnota);
   for(; N>0; N--)            // Načteme slovník a vytvoříme trii
   {
      getline(fin, klic, ' ');
      getline(fin, hodnota);
      trie.pridej(klic, hodnota);
   }

   string vstup;
   string slovo;
   ofstream fout;             // Výstupní soubor
   fout.open("ucebnice.out");

   // Samotný překlad, načteme celou řádku a rozdělíme ji na slova
   while(getline(fin, vstup ))
   {
      stringstream line_fin(vstup);
      while(getline(line_fin, slovo, ' '))
      {
         string *result;
         trie.najdi(slovo, result);

         if (result)
            fout << *result << ' ';
         else
            fout << slovo << ' ';
      }
      fout << endl;
   }
   fout.close();
   fin.close();

   return 0;
}
```

```
/* Učebnice -- řešení v C */

#include <stdio.h>
#include <string.h>

#define WORD_MAX 256          /* Maximální délka slova + 1 */

struct trie_node {
  struct trie_node *next[26];
  char val[WORD_MAX];
};

void add_to_trie(struct trie_node *root, char *word, char *val)
{
  int c = word[0] - 'A';
  if (!root->next[c])
    {
      root->next[c] = malloc(sizeof(struct trie_node));
      memset(root->next[c], 0, sizeof(struct trie_node));
    }
  if (word[1])
    add_to_trie(root->next[c], word+1, val);
  else
    strcpy(root->next[c]->val, val);
}

char *find_in_trie(struct trie_node *root, char *word)
{
  if (!root)
    return NULL;
  else if (!word[0])
    return root->val;
  else
    return find_in_trie(root->next[word[0]-'A'], word+1);
}

int main(void)
{
  FILE *in = fopen("ucebnice.in", "r");
  FILE *out = fopen("ucebnice.out", "w");
  struct trie_node trie_root = { };
  int N;

  fscanf(in, "%d", &N);
  for (int i=0; i<N; ++i)
    {
      char input[WORD_MAX], val[WORD_MAX];
      fscanf(in, "%s%s ", input, val);
      add_to_trie(&trie_root, input, val);
    }

  char word[WORD_MAX];
  int wlen = 0;
  int input;
  do
```

```
    {
      input = fgetc(in);
      if (input >= 'A' && input <= 'Z')
          word[wlen++] = input;
      else
          {
            if (wlen)
              {
                word[wlen] = 0;
                char *val = find_in_trie(&trie_root, word);
                if (val && val[0])
                    fputs(val, out);
                else
                    fputs(word, out);
                wlen = 0;
              }
            if (input != EOF)
              fputc(input, out);
          }
    }
  while (input != EOF);

  fclose(out);
  fclose(in);
  return 0;
}
```

P-I-2 Egyptské pyramidy

Základem řešení úlohy je metoda průchodu do šířky stavovým prostorem celé úlohy. Nejprve si popíšeme řešení varianty úlohy bez klíčů. Úloha se pak stává známým problémem o hledání nejkratší cesty v bludišti, který se dá vyřešit jednoduše průchodem do šířky, jak si dále popíšeme.

U navštívených políček si budeme pamatovat, po kolika krocích jsme na dané políčko přišli a z jakého políčka jsme tam dorazili. Políčka k prohledávání si ukládáme do fronty, ze které je postupně vybíráme. Na úplném počátku celého prohledávání se ve frontě nalézá pouze políčko, na kterém se Amon nachází na začátku. Pro každé políčko vybrané z fronty se podíváme na jeho sousedy a pokud nalezneme nějakého dosud nenavštíveného souseda, přidáme si ho na konec fronty políček určených ke zpracování. Zároveň si u takovéhoto políčka poznamenáme počet kroků, které jsme museli učinit k jeho dosažení, a políčko, ze kterého jsme na něj dorazili.

V momentě, kdy navštívíme políčko s pokladem, můžeme algoritmus ukončit, neboť kratší cesta na toto políčko nebo jiné políčko s pokladem nemůže existovat. Cestu z výchozího políčka na políčko s pokladem snadno zrekonstruujeme podle záznamů, odkud jsme na políčka na této cestě dorazili. Naopak, pokud se fronta vyprázdní a políčko s pokladem nebylo dosud označeno, můžeme s jistotou prohlásit, že se na žádné z políček s pokladem nedá z Amonovi výchozí pozice dostat.

Teď řešení rozšíříme na problém s klíči. Místo na políčka se budeme dívat na *stavy*, ve kterých se můžeme nacházet. Stav je v každém momentě popsán kombinací

klíčů, které vlastníme, a políčkem, na němž se právě nacházíme. Místo sousedství dvou políček budeme hovořit o možnosti přechodu z jednoho stavu do druhého. Úlohu lze pak přeformulovat tak, že chceme nalézt nejkratší cestu z výchozího stavu, kdy se Amon nachází na počátečním políčku a nemá klíč žádné barvy, do stavu, kdy se Amon nachází na některém z políček s pokladem (a má libovolnou kombinaci klíčů).

K vyřešení úlohy s klíči lze použít opět prohledávání do šířky – stačí v popsaném algoritmu vyměnit slova políčka za stavy. Všimněme si, že nám vůbec nezáleží na tom, kdy jsme jaký klíč sebrali a ani odkud ho máme. Záleží nám pouze na tom, zda klíč dané barvy máme k dispozici. Vzhledem k tomu, že klíče jsou právě čtyři, je počet různých kombinací klíčů, které můžeme mít u sebe, roven $2^4 = 16$. Políček je celkem $R \cdot S$. Celkový počet stavů je pak tedy $16 \cdot R \cdot S$, což je asymptoticky $\mathcal{O}(R \cdot S)$.

Přechod mezi jednotlivými stavy odpovídá buď přesunu z jednoho políčka na druhé, nebo přesunu a sebrání klíče, pokud se na daném políčku nachází klíč. Možnost, že klíč nesebereme, nebudeme uvažovat, protože tím, že klíč sebereme si nemůžeme uškodit. Z každého stavu lze přejít tedy do nanejvýše čtyř jiných stavů. Může to být méně, neboť některá políčka mohou být nedostupná, např. proto, že nemáme potřebný klíč k průchodu dveřmi.

V tomto momentě jsme si ujasnili, co jsou jednotlivé stavy a přechody mezi nimi. K vyřešení úlohy pak stačí jen použít průchod do šířky na stavech a nalézt nejkratší cestu z výchozího stavu (počáteční políčko, žádný klíč), do nějakého cílového (libovolné políčko s pokladem a libovolná kombinace klíčů). Časová složitost zřejmě záleží pouze na počtu stavů, které projdeme, neboť žádný nenavštívíme dvakrát. Ke zpracování každého z nich nám stačí konstantní čas, protože zkoumáme konstantní počet „sousedních“ stavů. Rovněž v paměti si držíme konstantně mnoho údajů ke každému stavu. Zpětná rekonstrukce nejkratší cesty pak rovněž vyžaduje nejvýše tolik kroků, kolik je možných stavů. Protože počet stavů je $\mathcal{O}(R \cdot S)$, je časová i paměťová složitost našeho řešení $\mathcal{O}(R \cdot S)$.

Na závěr poznamenejme, že ve vzorové implementaci ukládáme všechny informace o políčkách bitově, abychom se vešli do paměťového limitu. To samozřejmě nemá žádný vliv na asymptotický odhad, ale pro daná paměťová omezení je toto velmi podstatné.

```
program BludisteKlice;

const MaxSq = 1000000;
      Keys  = 4;
      KeyCn = 1 shl Keys;         { Počet kombinací klíčů }
      KeyMask = (1 shl Keys) - 1; { = 1111 dvojkově }

      { Následující konstanty definujeme tak, aby platilo: }
      { Libovolný klíč < Exit < Prázdné políčko < Překážka < Dveře }

      sqKeyRed      = 1;          { = 0001 (klíče) }
      sqKeyGreen    = 2;          { = 0010 }
      sqKeyBlue     = 4;          { = 0100 }
      sqKeyMagenta  = 8;          { = 1000 }
```

```
      sqExit        = 10;         { východ }
      sqEmpty       = 11;         { prázdné políčko }
      sqBlock       = 12;         { zeď }

      sqDoorRed     = 33;         { = 100001 (dveře) }
      sqDoorGreen   = 34;         { = 100010 }
      sqDoorBlue    = 36;         { = 100100 }
      sqDoorMagenta = 40;         { = 101000 }

      availMovesCnt = 4;
      availMoves: array[1..availMovesCnt, 1..2] of Shortint =
        ((-1, 0), (1, 0), (0, -1), (0, 1));
      availMovesName: array[1..availMovesCnt] of Char =
        ('Z','V','S','J');

var
   plan: array[0..MaxSq-1] of Byte;
   r, s: LongInt;
   startX, startY: LongInt;

   fromDir: array[0..MaxSq-1] of Cardinal; { dvojice bitů pro každou kombinaci }
   { Směr, ze kterého jsme na dané políčko přišli }

   fromDifCn: array[0..MaxSq-1] of Word;   { jeden bit pro kombinaci }
   { Přešli jsme z jedné kombinace klíčů do jiné? }

   used: array[0..MaxSq-1] of Word;        { jeden bit pro kombinaci }
   { Byli jsme již na tomto políčku? }

   queueI: array[0..MaxSq * KeyCn - 1] of Longint;
   queueCn: array[0..MaxSq * KeyCn - 1] of Byte;
   qCount: Longint;
   qPos: Longint;

   endIndex: Longint;
   endCn: Byte;

function GetIndex(x, y: Longint): Longint;
begin
  GetIndex := (y - 1) * s + (x - 1);
end;

function GetX(index: Longint): Longint;
begin
  GetX := (index mod s) + 1;
end;

function GetY(index: LongInt): Longint;
begin
  GetY := (index div s) + 1;
end;

function GetUsed(index: Longint; cn: Byte): Boolean;
begin
  GetUsed := (used[index] and (1 shl cn)) > 0;
```

```
end;

procedure SetUsed(index: Longint; cn: Byte);
begin
  used[index] := used[index] or (1 shl cn);
end;

function GetFromDir(index: Longint; cn: Byte): Byte;
begin
  GetFromDir := 1 + (fromDir[index] shr (2 * cn)) mod 4;
end;

procedure SetFromDir(index: Longint; cn: Byte; AFrom: Byte);
begin
  fromDir[index] := fromDir[index] or ((AFrom-1) shl (2 * cn));
end;

function GetFromDifCn(index: Longint; cn: Byte): Boolean;
begin
  GetFromDifCn := (fromDifCn[index] and (1 shl cn)) > 0;
end;

procedure SetFromDifCn(index: Longint; cn: Byte);
begin
  fromDifCn[index] := fromDifCn[index] or (1 shl cn);
end;

function MoveIndex(dx, dy: Shortint; index:Longint): Longint;
var x, y:Longint;
begin
  x := GetX(index) + dx;
  y := GetY(index) + dy;
  if x > s then x := x - s;
  if x < 1 then x := x + s;
  if y > r then y := y - r;
  if y < 1 then y := y + r;
  MoveIndex := GetIndex(x, y);
end;

procedure ReadInput(filename: String);
var c:Char;
    i, j: Longint;
    index: Longint;
    f:Text;
begin
  Assign(f, filename);
  Reset(f);
  Readln(f, r, s);
  for j := 1 to r do
  begin
    for i := 1 to s do
    begin
      index := GetIndex(i, j);
      read(f, c);
      case c of
```

```
      '*':begin
            startX := i;
            startY := j;
            plan[index] := sqEmpty;
          end;
      '$': plan[index] := sqExit;
      '.': plan[index] := sqEmpty;
      '#': plan[index] := sqBlock;

      'c': plan[index] := sqKeyRed;
      'z': plan[index] := sqKeyGreen;
      'm': plan[index] := sqKeyBlue;
      'f': plan[index] := sqKeyMagenta;

      'C': plan[index] := sqDoorRed;
      'Z': plan[index] := sqDoorGreen;
      'M': plan[index] := sqDoorBlue;
      'F': plan[index] := sqDoorMagenta;
      end;
    end;
    readln(f);
  end;
  Close(f);
end;

procedure WriteOutput(filename: String);
var f:Text;
    start: LongInt;
    index: Longint;
    cn: Byte;
    dir: Byte;
    i: Longint;

begin
  Assign(f, filename);
  Rewrite(f);
  if endIndex = -1 then
  begin
    writeln(f, 'NELZE');
  end else
  begin
    index := endIndex;
    cn := endCn;

    start := GetIndex(startX, startY);

    qPos := 0;
    while (index <> start) or (cn > 0) do
    begin
      dir :=  GetFromDir(index, cn);
      inc(qPos);
      queueI[qPos] := dir;

      if GetFromDifCn(index, cn) then
       begin
```

```
        cn := (cn xor plan[index]);
      end;
      index := MoveIndex(-availMoves[dir, 1], -availMoves[dir, 2], index);
    end;

    for i:=qPos downto 1 do
    begin
      write(f, availMovesName[queueI[i]]);
    end;
    writeln(f);
  end;
  Close(f);
end;

var index: Longint;
    cn: Byte;
    i: Longint;
    newIndex: Longint;
    newCn: Byte;
    sqType: Byte;
    keysNeeded: Byte;

begin
  ReadInput('pyramida.in');
  endIndex := -1;
  qCount := 0;
  qPos := 0;
  index := GetIndex(startX, startY);
  queueI[0] := index;
  queueCn[0] := 0;
  SetUsed(index, 0);

  while qPos <= qCount do
  begin
    index := queueI[qPos];
    cn := queueCn[qPos];

    for i:=1 to availMovesCnt do
    begin
      newIndex := MoveIndex(availMoves[i, 1], availMoves[i, 2], index);
      newCn := cn;
      sqType := plan[newIndex];
      keysNeeded := sqType and KeyMask;
      if sqType < sqExit then
        newCn := newCn or sqType;

      if not GetUsed(newIndex, newCn) then
        if (sqType < sqBlock) or
            ((sqType > sqBlock) and (keysNeeded and newCn = keysNeeded)) then
        begin
          SetUsed(newIndex, newCn);
          SetFromDir(newIndex, newCn, i);
          if newCn<>cn then SetFromDifCn(newIndex, newCn);

          qCount := qCount + 1;
```

```
          queueI[qCount] := newIndex;
          queueCn[qCount] := newCn;

          if sqType = sqExit then
          begin
            qCount := 0; { přerušení cyklu while - vyprázdníme frontu políček }
            endIndex := newIndex;
            endCn := newCn;
            Break;
          end;
        end;
    end;

    qPos := qPos + 1;
  end;

  WriteOutput('pyramida.out');
end.
```

P-I-3 Horský maraton

Nejjednodušším řešením zadané úlohy je postupné zkoušení všech hodnot i a j a porovnání příslušného součtu s X. Takové řešení však potřebuje $\mathcal{O}(N^3)$ kroků a zkušeným řešitelům olympiády je jistě jasné, že pro zisk plného počtu bodů je potřeba udělat trochu víc.

Pozorováním vedoucím k nalezení efektivnějšího řešení této úlohy je fakt, že součet podposloupnosti $a_i + \cdots + a_j$ je roven $s_j - s_{i-1}$, kde s_k značí součet prvních k členů posloupnosti (speciálně tedy platí $a_0 = 0$). Úlohu lze pak vyřešit tak, že pro každé j, nalezneme mezi čísly $s_0, \ldots, s_{j-1}$ takové, které je nejbližší hodnotě $s_j - X$. Pokud je takové číslo s_i, pak rozdíl $s_j - s_i$ je součet prvků posloupnosti, která končí i-tým prvkem a jejíž součet prvků je co nejbližší hodnotě X. Protože podposloupnost, jejíž součet je nejbližší X, musí někde končit, po provedení výše uvedeného postupu pro $j = 1, \ldots, N$ a vybrání nejlepšího nalezeného řešení, budeme mít řešení celé úlohy. Pokud výše popsaný postup přímočaře naimplementujeme, dostáváme algoritmus s časovou složitostí $\mathcal{O}(N^2)$. V následujících odstavcích si popíšeme, jak časovou složitost vylepšit na $\mathcal{O}(N \log N)$.

Pokud se zamyslíme, kde v našem algoritmu ztrácíme čas, zjistíme, že dobrým místem ke zlepšení by mohlo být vyhledávání čísla s_i mezi $s_0, \ldots, s_{j-1}$, které provádíme sekvenčně. Kdybychom ale měli k dispozici datovou strukturu, která by uměla rychle najít k zadanému číslu nejbližší hodnotu v ní uloženou, dosáhli bychom složitosti $\mathcal{O}(N \cdot \textit{složitost\_vyhledání} + N \cdot \textit{složitost\_přidání})$.

Jednou z takových datových struktur jsou binární vyhledávací stromy, které jsou schopny požadované operace provádět v logaritmickém čase. Přesný popis jejich struktury a implementace jednotlivých operací v našem řešení neuvedeme, ale odkážeme čtenáře na literaturu o datových strukturách, např. [1]–[3]. Pro naše účely postačí vědět, že se jedná o stromovou datovou strukturu, kde je každý vrchol v označen číslem x_v a má nejvýše dva syny, přičemž čísla v podstromu levého syna jsou menší než x_v a čísla v podstromu pravého syna jsou větší než x_v.

Přidání hodnoty do stromu je standardní operací, kterou je možné najít ve zmíněné literatoře, takže popíšeme jen hledání hodnoty, která je nejblíže zadanému x. To provedeme tak, že najdeme nejmenší hodnotu, která je větší nebo rovna x, a největší hodnotu menší nebo rovnou než x. Z těchto dvou hodnot pak vybereme tu bližší z nich.

A jak tedy najdeme tu nejmenší hodnotu větší nebo rovnou x? Provedeme vyhledání hodnoty x ve stromě obvyklým způsobem. Pokud najdeme přímo hodnotu x, můžeme vyhledávání ukončit. Jinak skončíme v nějakém listu v stromu. Nyní projdeme vrcholy na cestě z kořene do listu v a vybereme mezi nimi ten, jehož hodnota je nejmenší větší než x. Poznamenejme, že se může stát, že strom neobsahuje žádné hodnoty větší než x a v takovém případě vrátíme „nekonečno“.

Jak jsme již napsali, složitost přidání a nalezení nejbližší hodnoty ve stromě má časovou složitost logaritmickou v počtu prvků uložených ve stromě. Ovšem není to zadarmo – aby hloubka stromu, která určuje složitost vyhledávání, byla logaritmická, je potřeba provádět takzvané *vyvažování.* My si ale ukážeme jiné řešení – místo postupného přidávání hodnot do stromu vytvoříme hned na začátku strom, který bude obsahovat všechny hodnoty, ale na začátku všechny hodnoty ve stromě budou *neaktivní.* Postupně budeme hodnoty *aktivovat,* přičemž při hledání nejbližší hodnoty budeme neaktivní hodnoty ignorovat. Místo vložení hodnoty do stromu příslušnou hodnotu jen aktivujeme.

V souvislosti s touto změnou je potřeba upravit operaci hledání nejbližší hodnoty vyšší nebo rovné x. Tentokrát nám nestačí hledat jen na aktivních vrcholech na cestě z v do kořene, ale je třeba navíc prověřit ještě jeden vrchol. Na cestě z v do kořene najdeme první (tj., nejblíže k v) vrchol w, který má v pravém podstromu aktivní hodnotu a vrchol v je přitom v levém podstromu vrcholu w. V takovém případě pak mezi vrcholy, které uvažujeme při hledání minima, zahrneme i aktivní vrchol v pravém podstromu vrcholu w s nejnižší hodnotou. Pokud si pro každý vrchol budeme kromě hodnoty a příznaku aktivity pamatovat, zda v jeho levém a pravém podstromu tohoto vrcholu existuje nějaký aktivní vrchol, zvládneme nalézt hledanou hodnotu taktéž v čase úměrném hloubce stromu. Poznamenejme, že tyto příznaky zvládneme nastavit při aktivaci vrcholu bez zhoršení časové složitosti (hodnoty upravujeme jen na cestě z vrcholu do kořene).

Nyní již zbývá jen popsat, jak jednoduše vyrobíme strom, ve kterém budou uloženy všechny hodnoty s_i a jehož hloubka bude logaritmická. Nejprve hodnoty s_i setřídíme a odstraníme duplicity (pokud $s_i = s_{i'}$ a $i < i'$, pak $s_{i'}$ můžeme zapomenout). Nyní pomocí rekurzivní procedury postavíme strom. Řekněme, že chceme postavit strom pro hodnoty $t_1, t_2, \ldots, t_k$. Pokud je $k = 0$, pak je výsledkem prázdný strom, jinak do kořene stromu uložíme hodnotu $t_{\lfloor k/2 \rfloor}$, levý podstrom vytvoříme rekurzivním voláním na hodnoty $t_1, \ldots, t_{\lfloor k/2 \rfloor - 1}$ a pravý voláním na hodnoty $t_{\lfloor k/2 \rfloor + 1}, \ldots, t_k$. Protože délka intervalu s rostoucí hloubkou exponenciálně klesá, dostáváme, že celková hloubka stromu je logaritmická.

Naše řešení je nyní již zcela jasné. Nejprve spočítáme hodnoty s_i, postavíme strom, a postupně procházíme j od 1 do N. Při průchodu vždy nejprve najdeme

nejbližší hodnotu k $s_j - X$ (a její index) a pak hodnotu s_j aktivujeme. Nakonec vypíšeme nejlepší řešení.

Kolik času nám zabere naše řešení? Utřídění hodnot s_i (například použitím třídění pomocí haldy) zvládneme v čase $\mathcal{O}(N \log N)$. Postavení stromu pak potřebuje již jen lineární čas. V každém kroku následného cyklu pak jednou vyhledáváme a jednou aktivujeme, takže jedna iterace cykly trvá čas $\mathcal{O}(\log N)$. Celková časová složitost je tedy $\mathcal{O}(N \log N)$. Paměťová složitost je zjevně lineární v délce vstupu.

[1] D. Kráľ, M. Mareš, M. Straka: Recepty z programátorské kuchařky Korespondenčního semináře z programování – IV. část, *Rozhledy matematicko-fyzikální* 82(1) (2007), 22–35.

[2] M. Mareš, T. Valla: Kuchařky Korespondečního semináře z programování, *http://ksp.mff.cuni.cz/tasks/16/cook4.html.*

[3] P. Töpfer: Algoritmy a programovací techniky, Prometheus Praha 1995, 2. vydání 2007.


```
program posloupnost;
const MAX:word=100000;
var a:array[1..MAX] of longint;
    soucty:array[0..MAX] of longint;
    index:array[0..MAX] of word;
    X:longint;
    N:word;
type uzel=record
            hodnota:longint;   { hodnota součtu uložená v uzlu }
            aktivni:boolean;   { příznak, zda je hodnota aktivní }
            akt_strom:boolean; { příznak existence aktivních listů v podstromě,
                                 tj. u listu určuje jeho aktivitu }
            index:word;        { určuje délku posloupnosti odp. součtu }
            levy,pravy:^uzel;
          end;
type puzel=^uzel;
var koren:^uzel;

procedure bublej_nahoru(odkud, mez: word);
  { pomocná procedure pro heap-sort - prvek na pozici, odkud
    je zabubláván směrem nahoru v haldě po velikost určenou parametrem mez }
  var j1,j2,k:word;
  begin
    j2:=odkud;
    repeat
      j1:=j2;
      if (2*j1+1<=mez) and (soucty[j2]<soucty[2*j1+1]) then j2:=2*j1+1;
      if (2*j1+1<=mez) and (soucty[j2]=soucty[2*j1+1])
                       and (index[j2]<index[2*j1+1]) then j2:=2*j1+1;
      if (2*j1+2<=mez) and (soucty[j2]<soucty[2*j1+2]) then j2:=2*j1+2;
      if (2*j1+2<=mez) and (soucty[j2]=soucty[2*j1+2])
                       and (index[j2]<index[2*j1+2]) then j2:=2*j1+2;
      k:=soucty[j1]; soucty[j1]:=soucty[j2]; soucty[j2]:=k;
      k:=index[j1]; index[j1]:=index[j2]; index[j2]:=k;
    until j1=j2;
  end;
```

```
procedure nacti_a_setrid;
  var i,k:word;
  begin
    readln(N);
    for i:=1 to N do read(a[i]);
    readln(X);
    soucty[0]:=0;
    for i:=1 to N do soucty[i]:=soucty[i-1]+a[i];
    for i:=0 to N do index[i]:=i;
    { heap-sort }
    for i:=N downto 0 do bublej_nahoru(i,N);
    for i:=N downto 1 do
      begin
        k:=soucty[0]; soucty[0]:=soucty[i]; soucty[i]:=k;
        k:=index[0]; index[0]:=index[i]; index[i]:=k;
        bublej_nahoru(0,i-1);
      end;
    for i:=1 to N do
      begin
        if soucty[i-1]=soucty[i] then index[i]:=index[i-1];
      end;
    { ve stromě chceme, aby součty odpovídaly nejkratší možné posloupnosti }
  end;

function postav_strom(prvni, posledni: word):puzel;
  var u:^uzel;
      k:word;
  begin
    new(u);
    k:=(prvni+posledni) div 2;
    u^.hodnota:=soucty[k];
    u^.index:=index[k];
    u^.akt_strom:=false;
    u^.aktivni:=false;
    u^.levy:=nil;
    u^.pravy:=nil;
    if prvni<k then u^.levy:=postav_strom(prvni,k-1);
    if k<posledni then u^.pravy:=postav_strom(k+1,posledni);
    postav_strom:=u
  end;

procedure aktivuj(hodnota: word);
  var u:^uzel;
  begin
    u:=koren;
    u^.akt_strom:=true;
    while true do
      begin
        if hodnota=u^.hodnota then
          begin
            u^.aktivni:=true;
            exit
          end;
        if hodnota<u^.hodnota then
```

```
          u:=u^.levy
        else
          u:=u^.pravy;
        u^.akt_strom:=true
      end;
  end;

function nejblizsi_mensi(u: puzel; hodnota: longint): integer;
  var vysledek: integer;
  begin
    if (u=nil) or (not u^.akt_strom) then
      begin
        nejblizsi_mensi:=-1;
        exit
      end;
    if u^.aktivni and (u^.hodnota=hodnota) then
      begin
        nejblizsi_mensi:=u^.index;
        exit
      end;
    if u^.hodnota>hodnota then
      begin
        nejblizsi_mensi:=nejblizsi_mensi(u^.levy,hodnota);
        exit
      end;
    vysledek:=nejblizsi_mensi(u^.pravy,hodnota);
    if vysledek<>-1 then
      begin
        nejblizsi_mensi:=vysledek;
        exit
      end;
    if u^.aktivni then
      begin
        nejblizsi_mensi:=u^.index;
        exit
      end;
    nejblizsi_mensi:=nejblizsi_mensi(u^.levy,hodnota);
  end;

function nejblizsi_vetsi(u: puzel; hodnota:longint): integer;
  var vysledek: integer;
  begin
    if (u=nil) or (not u^.akt_strom) then
      begin
        nejblizsi_vetsi:=-1;
        exit
      end;
    if u^.aktivni and (u^.hodnota=hodnota) then
      begin
        nejblizsi_vetsi:=u^.index;
        exit
      end;
    if u^.hodnota<hodnota then
      begin
        nejblizsi_vetsi:=nejblizsi_vetsi(u^.pravy,hodnota);
```

```
        exit
      end;
    vysledek:=nejblizsi_vetsi(u^.levy,hodnota);
    if vysledek<>-1 then
      begin
        nejblizsi_vetsi:=vysledek;
        exit
      end;
    if u^.aktivni then
      begin
        nejblizsi_vetsi:=u^.index;
        exit
      end;
    nejblizsi_vetsi:=nejblizsi_vetsi(u^.pravy,hodnota);
  end;

procedure vyres;
  var soucet: longint;
      opt_soucet: longint;
      opt_od, opt_do: word;
      i: word;
      k: integer;
  begin
    aktivuj(0);
    aktivuj(a[1]);
    soucet:=a[1];
    opt_soucet:=a[1];
    opt_od:=1; opt_do:=1;
    soucty[0]:=0;
    for i:=1 to N do soucty[i]:=soucty[i-1]+a[i];
    for i:=2 to N do
      begin
        soucet:=soucet+a[i];
        k:=nejblizsi_mensi(koren,soucet-X);
        if (k<>-1) and (abs(soucty[i]-soucty[k]-X)<abs(opt_soucet-X)) then
          begin
            opt_soucet:=soucty[i]-soucty[k];
            opt_od:=k+1; opt_do:=i;
          end;
        k:=nejblizsi_vetsi(koren,soucet-X);
        if (k<>-1) and (abs(soucty[i]-soucty[k]-X)<abs(opt_soucet-X)) then
          begin
            opt_soucet:=soucty[i]-soucty[k];
            opt_od:=k+1; opt_do:=i;
          end;
        aktivuj(soucet);
      end;
    writeln(opt_od,' ',opt_do,' ',opt_soucet);
  end;

begin
  nacti_a_setrid;
  koren:=postav_strom(0,N);
  vyres;
end.
```

P-I-4 Stackal

a) Nejprve ukážeme, jak pomocí tří zásobníků rozpoznat symetrický řetězec v lineárním čase. Kdybychom uměli poznat, kde má zadaný řetězec svůj střed, byla by úloha snadná: všechny znaky nalevo od středu bychom postupně ukládali na zásobník a až bychom střed překročili, zase bychom je vybírali a porovnávali s načítanými znaky. Střed ovšem ve vstupu nepoznáme, protože nevíme, kolik znaků ještě přijde. Musíme na to tedy jít trochu důmyslněji:

Vstup budeme kopírovat do zásobníku V a další zásobník P budeme používat jako počítadlo – počet v něm uložených znaků nám bude říkat délku vstupu. Jakmile vstup skončí, budeme odebírat znaky z P a za každé dva odebrané přesuneme jeden znak z V do třetího zásobníku W. Pokud měl vstupní řetězec sudou délku, tímto postupem se do W dostane druhá polovina vstupu a ve V zůstane polovina první, přičemž na vrcholu obou těchto zásobníků bude znak nejbližší ke středu řetězce. Pak nám stačí vybírat z obou zásobníků současně a ověřovat, že si poloviny odpovídají. Pokud bude délka vstupu lichá, v P zbude jeden nepárový znak, což snadno poznáme a odstraníme ho i z V, protože ho není s čím porovnat.

Program bude vypadat následovně:

```
program symetrie_rychle;

var V, W, P: stack of char;          { první polovina, druhá, počítadlo }
    c: char;                         { právě zpracovávaný znak }

begin
   { Přečteme celý vstup do V a počítáme si v P jeho délku }
   while read(c) do begin
      push(V, c);
      push(P, c);
      end;

   { Pomocí P hledáme střed }
   while not empty(P) do begin       { odebíráme z P po dvojicích }
      pop(P);
      if empty(P) then               { korekce na lichý vstup }
         pop(V)
      else begin                     { přesuneme jeden znak z V do W }
         pop(P);
         c := pop(V);
         push(W, c);
         end;
      end;

   { Porovnáme obě poloviny }
   while not empty(V) do begin
      c := pop(V);
      if pop(W) <> c then begin
         write(0);
         halt;
         end;
      end;
```

```
   write(1);
end.
```

b) Pokud jsme ochotni obětovat rychlost, vystačíme si se dvěma zásobníky. Využijeme toho, že libovolný řetězec můžeme uložit do dvou zásobníků tak, aby všechny znaky před pozicí, kterou právě zkoumáme, byly v prvním zásobníku a znaky za ní v tom druhém, přičemž znaky na vrcholech zásobníků budou nejbližší ke zkoumané pozici. Přesun zkoumaného místa o znak doleva nebo doprava se pak odehraje přesunem znaku z jednoho zásobníku do druhého; podobně také můžeme zkoumaný znak z řetězce smazat.

Načteme si tedy vstup do dvojice zásobníků a přesuneme se na jeho začátek. Pak odebereme první znak, přesuneme se na konec řetězce a porovnáme první znak s posledním. Následně odebereme i ten poslední, přesuneme se na začátek a vše opakujeme, dokud nějaké znaky zbývají.

Správnost tohoto postupu je evidentní, jak je to s časovou složitostí? Při porovnávání první dvojice znaků provedeme řádově n kroků, na druhou dvojici $n-2$, na třetí $n-4$ atd. až na poslední dvojici znaků dva kroky. To je celkem $\mathcal{O}(n^2)$. Zbývá program:

```
program symetrie_usporne;

var L, P: stack of char;            { před kurzorem, za kurzorem }
    c: char;                        { zkoumaný znak }
    d: char;                        { zapamatovaný počáteční znak }

begin
   { Načteme celý vstup }
   while read(c) do push(L, c);

   { Přesuneme se na začátek }
   while not empty(L) do begin c := pop(L); push(P, c); end;

   { Střídavě odebíráme znaky z obou konců a porovnáváme je }
   while not empty(P) do begin
      { Odebereme první znak; když byl jediný, končíme }
      d := pop(P);
      if empty(P) then break;

      { Přesuneme se na konec a porovnáme poslední znak }
      while not empty(P) do begin c := pop(P); push(L, c); end;
      if pop(L) <> d then begin write(0); halt; end;

      { A zpět na začátek }
      while not empty(L) do begin c := pop(L); push(P, c); end;
      end;

   write(1);
end.
```