58. ročník Matematické olympiády – 2008/2009

Řešení úloh krajského kola kategorie P

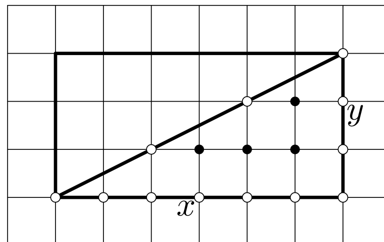
P-II-1 Lesník Jehlička

Intuitivně se zdá, že počet mřížových bodů uvnitř mnohoúhelníka by měl být úměrný jeho obsahu. A skutečně, platí následující vzorec pro každý mnohoúhelník se všemi vrcholy v mřížových bodech: $S = N + H/2 - 1$, kde S je obsah mnohoúhelníka, N je počet mřížových bodů uvnitř a H je počet mřížových bodů na hranici (včetně vrcholů). Naznačme si důkaz tohoto tvrzení: Každý mnohoúhelník lze vytvořit „slepením“ několika trojúhelníků. Nechť mnohoúhelník M vzniknul slepením mnohoúhelníků A a B přes hranu, v jejímž vnitřku leží x mřížových bodů. Označme počet mřížových bodů uvnitř A jako N_A a počet mřížových bodů na hranici A jako H_A. Podobně N_B a H_B je počet mřížových bodů uvnitř a na hranici B. Potom $N = N_A + N_B + x$ a $H = H_A + H_B - 2x - 2$. Z toho plyne, že $(N_A + H_A/2 - 1) + (N_B + H_B/2 - 1) = (N_A + N_B + x) + (H_A + H_B - 2x - 2)/2 - 1 = N + H/2 - 1$. Jestliže tedy obsah A je roven $N_A + H_A/2 - 1$ a obsah B je roven $N_B + H_B/2 - 1$, pak obsah M je roven $N + H/2 - 1$. Vzorec tedy stačí dokázat pro trojúhelníky.

Každý trojúhelník lze „vyjádřit“ z pravoúhlých trojúhelníků takových, že dvě z jejich hran jsou rovnoběžné s osami souřadnic, pokud dovolíme trojúhelníky i „odečítat“. Podobně jako v předchozím případě nahlédneme, že stačí vzorec dokázat pro takové pravoúhlé trojúhelníky. Obsah pravoúhlého trojúhelníka s odvěsnami o délkách x a y je $\frac{xy}{2}$. Je-li uvnitř jeho přepony k mřížových bodů, pak:

$$N = \frac{(x-1)(y-1)-k}{2},$$

jelikož uvnitř trojúhelníka leží polovina vnitřních mřížových bodů obdélníka o hranách x a y, kromě těch na diagonále:

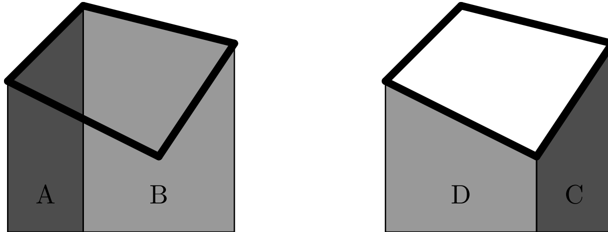


Dále, $H = x + y + k + 1$, tedy náš vzorec skutečně dává správnou hodnotu pro obsah:

$$\frac{(x-1)(y-1)-k}{2} + \frac{x+y+k+1}{2} - 1 = \frac{xy}{2}.$$

K vyřešení úlohy, tedy stačí určit obsah mnohoúhelníka a počet mřížových bodů na jeho hranici. Body na hranici spočítáme tak, že sečteme počet mřížových bodů uvnitř všech úseček tvořících hranici mnohoúhelníka s počtem jeho vrcholů. Počet mřížových bodů uvnitř úsečky, jejíž vrcholy mají souřadnice (x_1, y_1) a (x_2, y_2), je největší společný dělitel $x_1 - x_2$ a $y_1 - y_2$, zmenšený o 1.

Obsah mnohoúhelníka určíme tak, že si ho rozdělíme na lichoběžníky takové, že dvě z jejich hran jsou svislé a třetí leží na vodorovné ose souřadnic (některé z těchto lichoběžníků musíme odečítat). Obsah mnohoúhelníka je dán součtem obsahů těchto lichoběžníků (obsah odečítaných lichoběžníků bereme jako záporný). Například obsah čtyřúhelníka v následujícím obrázku je $A + B - C - D$.



Pro určení obsahu lichoběžníků použijeme známý vzorec – průměr délek základen × výška. Vzhledem k tomu, že jak v tomto vzorci, tak ve vzorci pro počet mřížových bodů se vyskytuje dělení dvěma, budeme raději počítat dvojnásobek obsahu, abychom mohli používat celá čísla.

Časová složitost určení obsahu je lineární. Časová složitost určení počtu bodů na hranici závisí na složitosti určení největšího společného dělitele. Použijeme-li Euklidův algritmus, pak je tato složitost $\mathcal{O}(\log \min(X, Y))$, kde X a Y jsou omezení na velikost souřadnic vrcholů mnohoúhelníka. Výsledná časová složitost tedy je $\mathcal{O}(n \log \min(X, Y))$. Kromě souřadnic vrcholů mnohoúhelníka si potřebujeme pamatovat pouze konstantní množství mezivýsledků, paměťová složitost je tedy $\mathcal{O}(n)$.

```
program les;

const MAXN = 100000;

type bod = record
             x, y : integer;
           end;

var n : integer;
    body : array [1 .. MAXN] of bod;
    hranice, dobsah : integer;

procedure nacti_vstup;
var i : integer;
begin
  readln (n);
  for i := 1 to n do
```

```
    readln (body[i].x, body[i].y);
end;

function nsd (n1, n2 : integer) : integer;
var t : integer;
begin
  if n1 > n2 then
    begin
      t := n1;
      n1 := n2;
      n2 := t;
    end;

  while n1 > 0 do
    begin
      t := n1;
      n1 := n2 mod n1;
      n2 := t;
    end;

  nsd := n2;
end;

function pocet_vnitrnich_bodu_usecky (a, b : bod) : integer;
var dx, dy : integer;
begin
  dx := abs (a.x - b.x);
  dy := abs (a.y - b.y);
  if (dx = 0) and (dy = 0) then
    pocet_vnitrnich_bodu_usecky := 0
  else
    pocet_vnitrnich_bodu_usecky := nsd (dx, dy) - 1;
end;

function bodu_na_hranici : integer;
var i, b : integer;
begin
  b := n + pocet_vnitrnich_bodu_usecky (body[1], body[n]);
  for i := 1 to n - 1 do
    b := b + pocet_vnitrnich_bodu_usecky (body[i], body[i + 1]);
  bodu_na_hranici := b;
end;

function dvakrat_obsah_lichobeznika (a, b : bod) : integer;
begin
  dvakrat_obsah_lichobeznika := (a.y + b.y) * (a.x - b.x);
end;

function dvakrat_obsah_mnohouhelnika : integer;
var i, s : integer;
begin
  s := dvakrat_obsah_lichobeznika (body[n], body[1]);
  for i := 1 to n - 1 do
    s := s + dvakrat_obsah_lichobeznika (body[i], body[i + 1]);
```

```
  dvakrat_obsah_mnohouhelnika := abs (s);
end;

begin
  nacti_vstup;
  hranice := bodu_na_hranici;
  dobsah := dvakrat_obsah_mnohouhelnika;
  writeln (1 + (dobsah + hranice) div 2);
end.
```

P-II-2 Čínsky nebo česky?

Každý si zcela jistě všiml podobnosti s úlohou z domácího kola. Tato podobnost není zcela náhodná, neboť pro řešení této úlohy použijeme stejnou datovou strukturu jako v řešení úlohy P-I-1. Tato struktura se nazývá trie a její popis naleznete v řešení právě zmíněné úlohy.

Ještě než se ale vrhneme na řešení úlohy zaveďme si značení, které nám usnadní další popis. S bude označovat počet slovníků, L_S počet písmen ve všech slovnících a nakonec délka překládaného textu ve znacích bude L_T.

Asi nejjednodušší způsob, který každého okamžitě napadne, je postupně si pro každý jazyk postavit samostatnou trii. Poté projdeme vstupní text po jednotlivých slovech a pro každé slovo zjistíme, ve kterých triích (tj. slovnících) se vyskytuje. Každý takový výskyt samozřejmě nezapomeneme započítat odpovídajícímu jazyku. Nakonec nalezneme jazyk s maximálním počtem takových výskytů a jednoduše vypíšeme odpověď.

Takto přímočarý postup má složitost $\mathcal{O}(L_T \cdot S + L_S)$, neboť každé slovo textu vyhledáváme v trii tolikrát, kolik máme trií (těch je stejně jako slovíků). Vytvoření všech trií nedokážeme lépe, než v čase L_S, protože každé slovo z každého slovníku musí být přidáno do odpovídající trie. Paměťová složitost bude nejhůře $\mathcal{O}(L_S)$.

To sice není špatné, ale jistě vás napadla spousta zlepšení. Jedním takovým může být to, že postavíme společnou trii pro všechny slovníky a u každého slova si zapamatujeme, v jakých slovnících se nachází. To lze implementovat například pomocí spojových seznamů v koncových uzlech trie.

Překládaný text pak opět čteme po jednotlivých slovech. Každé slovo ale v trii vyhledáváme pouze jednou a počítáme si výskyt slov nezávisle na jazyku. Po skončení celou trii projdeme (nejlépe průchodem do hloubky) a jazykům ve spojových seznamech vždy přičítáme výskyty za jednotlivá slova.

Je důležité si uvědomit, že vytvoření takové trie a zároveň i poslední průchod nezabere více než $\mathcal{O}(L_S)$. Přidané spojové seznamy totiž celkovou časovou ani paměťovou složitost neovlivní. Je to proto, že každý uzel v seznamu zastupuje jedno slovo. V trii tak máme nejvýše $\mathcal{O}(L_S)$ normálních uzlů a nejvýše $\mathcal{O}(L_S)$ uzlů spojového seznamu. Při vytváření i při závěrečném průchodu pak každý uzel zpracováváme právě jednou, takže celková časová složitost je $\mathcal{O}(L_S + L_T)$. Paměťová je ze stejných důvodů $\mathcal{O}(L_S)$.

Lepší asymptotické časové složitosti nelze dosáhnout, neboť vstupní soubory musíme alespoň jednou přečíst.

Přestože víme, že dalšího zrychlení dosáhnout nelze, můžeme se pokusit o implementačně jednodušší řešení. V právě popsaném algoritmu jsme si pro každé slovo pamatovali, ve kterých slovnících se nachází. Zkusme tuto informaci z trie zcela vynechat.

Po jednom přečtení vstupního textu budeme o každém slovu vědět, kolikrát se v textu vyskytovalo. Bohužel nevíme, ve kterých slovnících se tato slova nacházela.

Nic nám ale nebrání v tom, přečíst si všechny slovníky ještě jednou. Ve společné trii pak vyhledáme všechna slova z určitého slovníku a příslušnému jazyku přičítáme počet výskytů aktuálního slova.

Takové procházení bude trvat $\mathcal{O}(L_S)$ kroků, takže celkovou časovou složitost nepokazí. Naopak je tento algoritmus mnohem jednodušší na implementaci.

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

#define WORD_LEN 255
#define ALPHABET_SIZE 26

struct trie_node {
      struct trie_node *next[ALPHABET_SIZE];
      int count;
   };
typedef struct trie_node TRIE_NODE;

struct dict_desc {
      char name[WORD_LEN];
      int size;
   };
typedef struct dict_desc DICT_DESC;

void trie_add(TRIE_NODE *root, const char *word)
{
   TRIE_NODE *cur_root = root;

   while (*word)
   {
      if (!cur_root->next[*word-'a'])
      {
         cur_root->next[*word-'a'] = malloc(sizeof(TRIE_NODE));
         for (int i=0; i<ALPHABET_SIZE; i++)
            cur_root->next[*word-'a']->next[i] = NULL;
         cur_root->next[*word-'a']->count = 0;
      }

      cur_root = cur_root->next[*word-'a'];
      ++word;
   }
   //Není potřeba si pamatovat konce slov, v závěrečném počítání se hodnoty
   //v nekoncových uzlech zanedbají
}

void trie_increase_word_count(TRIE_NODE *root, const char *word)
```

```
{
   TRIE_NODE *cur_root = root;    //poslední fáze, máme jistotu že slovo najdeme

   while (*word && cur_root)
   {
      cur_root = cur_root->next[*word-'a'];
      ++word;
   }
   if (cur_root)
      cur_root->count++;
}

int trie_find(TRIE_NODE *root, const char *word)
//nalezne slovo v trii a vrátí, kolikrát se vyskytlo
{
   TRIE_NODE *cur_root = root;    //poslední fáze, máme jistotu že slovo najdeme

   while (*word)
   {
      cur_root = cur_root->next[*word-'a'];
      ++word;
   }
   return cur_root->count;
}

void trie_free(TRIE_NODE *root)  //uvolnění struktur
{
   if (!root)
      return;
   for(int i=0; i<ALPHABET_SIZE; i++)
      trie_free(root->next[i]);
   free(root);
}

TRIE_NODE *trie_root;
int N;

int main(void)
{
   trie_root = malloc(sizeof(TRIE_NODE));
   for (int i=0; i<ALPHABET_SIZE; ++i)
      trie_root->next[i] = NULL;
   trie_root->count = 0;

   //inicializace trie
   FILE *fdict = fopen("slovniky.in", "r");
   int dict_count;
   fscanf(fdict, "%d", &dict_count);
   for (int dict=0; dict<dict_count; dict++)
   {
      int dict_size;
      char word[WORD_LEN+1];
      fscanf(fdict, "%s %d", word, &dict_size);
      for(int cur_word=0; cur_word<dict_size; cur_word++)
      {
```

```
            fscanf(fdict, "%s", word);
            trie_add(trie_root, word);
        }
    }
    fclose(fdict);

    //zjištění počtů slov v textu
    FILE *ftext = fopen("text.in", "r");
    while( !feof(ftext))
    {
        char word[WORD_LEN+1];
        fscanf(ftext, " %s ", word);
        trie_increase_word_count(trie_root, word);
    }
    fclose(ftext);

    //zjištění počtů slov ve slovnících
    fdict = fopen("slovniky.in", "r");
    fscanf(fdict, "%d", &dict_count);
    DICT_DESC *dict_desc = malloc(sizeof(DICT_DESC) * dict_count);

    for (int dict=0; dict<dict_count; dict++)
    {
        int dict_size;
        fscanf(fdict, "%s %d", dict_desc[dict].name, &dict_size);
        dict_desc[dict].size = 0;

        for(int cur_word=0; cur_word<dict_size; cur_word++)
        {
            char word[WORD_LEN+1];
            fscanf(fdict, "%s", word);
            dict_desc[dict].size+= trie_find(trie_root, word);
        }
    }
    fclose(fdict);

    //nalezení slovníků s největším počtem slov
    int max = 0;
    for (int dict=0; dict<dict_count; dict++)
        if (dict_desc[dict].size>max)
            max = dict_desc[dict].size;

    //vypsání slovníků
    FILE *fout = fopen("jazyk.out", "w");
    for (int dict=0; dict<dict_count; dict++)
        if (dict_desc[dict].size==max)
            fprintf(fout, "%s\n", dict_desc[dict].name);
    fclose(fout);

    //uvolnění trie
    trie_free(trie_root);
    return 0;
}
```

P-II-3 Cyklistický závod

Hned po přečtení zadání je jasné, že tato úloha bude mít něco společného s grafovými algoritmy. Města budou vrcholy, možné etapy závodu budou hrany, a počty diváků si pak můžeme představit jako ohodnocení těchto hran nezápornými celými čísly. Jak ale najít kýžené řešení?

Začneme první částí úlohy, tedy nalezením nějaké trasy závodu z Vyšných Háků do Velkého Sumce s co nejméně etapami (ovšem bez požadavku na maximalizaci počtu diváků). Takto zadaná úloha je v podstatě učebnicovým příkladem na procházení grafu do šířky, neboli algoritmu vlny. Pro každé město spočteme minimální počet etap nutných pro jeho dosažení a to tak, že Vyšné Háky dostanou nulu, jejich sousedé jedničku, sousedé jejich sousedů dvojku, atd. V grafu nám tak vzniknou jakési vrstvy, přičemž v k-té vrstvě jsou vrcholy, do nichž se lze dostat nejkratší cestou za k etap. Nalezení nějaké trasy je pak jednoduché: Začneme z posledního vrcholu, tedy z Velkého Sumce, pak vezmeme nějakého jeho souseda ve vrstvě o jedničku nižší, a tak pokračujeme, dokud se nedostaneme do Vyšných Háků. Toto řešení vyžaduje lineární čas a lineární množství paměti, tj. $\mathcal{O}(M+N)$.

A jak nalezneme tu divácky nejatraktivnější trasu? Pozornému čtenáři jistě neuniklo, že každou trasu s nejmenším počtem etap (tedy i tu nejlepší) lze získat postupem z předchozího odstavce. Vše by se zjednodušilo, kdybychom pro každé město znali celkový počet diváků, který shlédne nejlepší trasu závodu v daném městě končící (tj. začínající ve Vyšných Hácích a s nejmenším počtem etap a mezi takovými s největší návštěvností). Pro dané město bychom to předchozí na trase našli tak, že bychom se podívali, kolik diváků shlédne nejlepší trasu do něj a pak zkusili, ze kterého města že jsme se tam dostali – díky předpočítaným hodnotám tímto postupem strávíme $\mathcal{O}(M+N)$ času, protože každou dvojici sousedních měst testujeme maximálně dvakrát a test trvá konstantní čas.

A jak provedeme požadovaný předvýpočet? Jednoduše rozšířením hledání do šířky ze začátku: Pro Vyšné Háky je počet diváků nulový (jsme teprve na startu), pro města vzdálená jednu etapu odpovídá tato hodnota počtu lidí, kteří se na tuto etapu podívají, atd. Přesněji, pro město X ve vrstvě $k+1$, které sousedí s městy $Y_1, \ldots, Y_\ell$ ve vrstvě k, je počet diváků roven maximu přes i ze součtů počtu diváků pro Y_i (která je již spočítaná, protože město je v nižší vrstvě) a počtu diváků etapy z Y_i do X. Tento výpočet zvládneme udělat v čase $\mathcal{O}(M+N)$, takže výsledný čas i paměť jsou $\mathcal{O}(M+N)$.

Samotný program je v podstatě přepisem výše uvedeného algoritmu. Abychom opravdu dosáhli časové a paměťové složitosti $\mathcal{O}(M+N)$, je graf měst a etap uložen jako seznam následníků. Za zmínku asi stojí malý trik, jak program řeší výpis trasy – místo hledání trasy od konce a následného otočení pořadí měst, jak by naznačoval algoritmus, najdeme cestu z Velkého Sumce do Vyšných Háků a při hledání trasy od konce ji rovnou vypisuje – takže vypíšeme trasu z Vyšných Háků do Velkého Sumce.

```
program zavod;
```

```
const maxN = 1000000;
      maxM = 1000000;
      zacatek = 2;  { prohodili jsme startovní a cílove město, abychom }
      konec = 1;    { nemuseli otáčet cestu }

var M,N:integer;

    mesta: array [1..maxN] of record { informace o mestech }
            vzdalenost : integer;   { počet etap od začátku }
            maxDivaku : integer;    { počet diváků, který shlédne trasu až sem}
            predchozi:integer;      { předchozí město na nejlepší trase }
            stupen: integer;        { počet sousedních měst}
            zacatek: integer;       { pozice prvního souseda v poli sousede }
            pozice:integer; {dočasná hodnota používaná při načítání}
        end;
    sousede: array[1..2*maxM] of record {pole sousedů. nejprve jsou uloženi }
                { sousedé města 1, pak sousedé města 2, atd. }
            mesto, divaku:integer;
        end;
    hrany: array[1..maxM] of record   { pole hran, neboli možných etap. Použito}
                                      { jen při načítání. }
        odkud, kam, divaku: integer;  { odkud a kam etapa vede a počet diváků }
        end;

{ Načtení dat ze vstupu - naplní pole mesta a sousede. }
procedure nacti;
var i, pozice:integer;
    a,b, divaku:integer; {města}
    p:integer;
begin
    readln(N, M);
    for i := 1 to N do begin
        mesta[i].stupen := 0;
    end;
    {načti hrany a spočti stupně}
    for i:= 1 to M do begin
        readln(a, b, hrany[i].divaku);
        hrany[i].odkud := a; hrany[i].kam := b;
        mesta[a].stupen := mesta[a].stupen + 1;
        mesta[b].stupen := mesta[b].stupen + 1;
    end;
    { urči začátky pro jednotlivá města v poli sousede }
    pozice := 1; {pruběžná pozice v poli sousede}
    for i := 1 to N do begin
        mesta[i].zacatek := pozice;
        mesta[i].pozice := pozice;
        pozice := pozice + mesta[i].stupen;
    end;
    { naplň pole sousede }
    for i:= 1 to M do begin
        a := hrany[i].odkud; b := hrany[i].kam; divaku := hrany[i].divaku;
        {přidej souseda městu a}
        p:= mesta[a].pozice;
        sousede[p].mesto := b;
        sousede[p].divaku := divaku;
```

```
        mesta[a].pozice := p+ 1;
        {přidej souseda městu b}
        p:= mesta[b].pozice;
        sousede[p].mesto := a;
        sousede[p].divaku := divaku;
        mesta[b].pozice := p+ 1;
    end;

end;

{ Fronta měst ke zpracování. Vzhledem ke způsobu přidávání platí, že }
{ město s nižší vzdáleností od začátku je v poli na nižší pozici než }
{ město s větší vzdáleností. }
var fronta: array[1..maxN] of integer;
    zacF,konF:integer; {začátek a konec fronty}

{ Projde sousedy vybraného města a pokusí se prodloužit do nich trasu. }
procedure projdiSousedy(mesto:integer);
var i: integer; {index do pole sousedů}
    posledni: integer; {index posledního souseda v poli sousede. }
    divaku, soused : integer;
begin
    posledni := mesta[mesto].zacatek + mesta[mesto].stupen - 1;
    for i:= mesta[mesto].zacatek to posledni do begin
        divaku := mesta[mesto].maxDivaku + sousede[i].divaku;
        soused := sousede[i].mesto;
        if mesta[soused].vzdalenost = -1 then begin
            {město jsme ještě nenavštívili.}
            konF := konF + 1;
            fronta[konF] := soused; {dej do fronty}
            mesta[soused].vzdalenost := mesta[mesto].vzdalenost + 1;
            mesta[soused].maxDivaku := divaku; { a toto je zatim nejlepší trasa}
            mesta[soused].predchozi := mesto;
        end else if (mesta[soused].vzdalenost = mesta[mesto].vzdalenost + 1)
            and (mesta[soused].maxDivaku < divaku) then begin
            { trasa pšes toto město je lepší, přenastavíme hodnoty u souseda.}
            mesta[soused].maxDivaku := divaku;
            mesta[soused].predchozi := mesto;
        end;
    end;
end;

{ Spočte hodnoty maxDivaku, vzdalenost a predchozi v poli mesta, čímž }
{ prakticky vyřeší celou úlohu. }
procedure spocti;
var mesto: integer;
    i:integer;
begin
    {inicializace}
    for i := 1 to N do begin
        mesta[i].maxDivaku := 0;
        mesta[i].vzdalenost := -1; {zatim jsme nikde nebyli}
    end;
    {počáteční město má vzdálenost 0 a žádné diváky}
    mesta[zacatek].vzdalenost := 0;
```

```
    mesta[zacatek].maxDivaku := 0;
    fronta[1] := zacatek;
    zacF := 1; konF := 1;
    {dokud není fronta prazdná}
    while zacF <= konF do begin
        mesto := fronta[zacF];
        zacF := zacF + 1;  {přečti hodnotu z fronty}
        projdiSousedy(mesto);
    end;
end;

{ Vypíše cestu od konce do začátku. Protože cesta, kterou najde spocti(), }
{ končí ve Vyšných Hacích, tak je vysledkem přesně to, co požaduje zadání. }
procedure vypis;
var mesto:integer;
begin
    writeln(mesta[konec].vzdalenost, ' ', mesta[konec].maxDivaku);
    mesto := konec;
    write(konec);
    while mesto <> zacatek do begin   {dokud nejsme na začátku}
        mesto := mesta[mesto].predchozi; {najdi město, ze kterého jsme přišli}
        write(' ', mesto);       {vypiš ho}
    end;
    writeln;
end;


{ Hlavní program }
begin
    nacti;
    spocti;
    vypis;
end.
```

P-II-4 Stackal

Popíšeme řešení používající jeden zásobník. Na tomto zásobníku si budeme v průběhu výpočtu udržovat co nejjednodušší výraz, který je ekvivalentní zatím přečtené části vstupu.

Zapomeňme na chvíli na závorky. Výraz budeme načítat po znacích. Pokud načteme hodnotu `0` nebo `1`, uložíme ji na zásobník. Pokud načteme operátor `|`, je možné vyhodnotit všechny již načtené a dosud nevyhodnocené operátory. Pokud nějaký nevyhodnocený operátor existuje, vypadá vrchol zásobníku jako *hodnota, operátor, hodnota*. Vyjmeme tedy tuto trojici ze zásobníku, vypočteme její hodnotu a vrátíme ji zpět na zásobník. Když už na zásobníku žádný operátor není, vložíme na vrchol načtený operátor `|`.

Pokud načteme `&`, můžeme vyhodnotit pouze načtené operátory `&`, protože načtené operátory `|` se vyhodnotí až po právě načteném operátoru. Takže dokud vypadá vrchol zásobníku jako *hodnota,* `&`*, hodnota*, vyhodnocujeme, a nakonec vložíme načtený operátor `&` na zásobník.

Po přečtení všech znaků výrazu stačí vyhodnotit operátory, které nám zbyly

na zásobníku. Pokud byl daný výraz korektní, zůstane na zásobníku právě jedna hodnota, a to hodnota celého výrazu.

Ještě vyřešíme závorky. Pokud načteme otevírací závorku, musíme rekurzivně vyhodnotit výraz mezi touto a zavírací závorkou. Provedeme to tak, že uložíme otevírací závorku na zásobník. Při načtení operátorů `|` a `&` budeme již popsaným způsobem vyhodnocovat uložené operátory na zásobníku. Pokud ale narazíme na otevírací závorku (vrchol zásobníku vypadá jako *levá závorka, hodnota*), vyhodnocování zastavíme a uložíme načtený operátor na zásobník. Poslední případ je, že načteme závorku zavírací. Tehdy budeme vyhodnocovat všechny operátory na zásobníku, dokud nenarazíme na otevírací závorku, kterou pak ze zásobníku odstraníme.

Není těžké si rozmyslet, že popsaný algoritmus funguje správně, pokud je zadaný výraz korektní, a běží v lineárním čase s jedním zásobníkem.

Pro ty, kteří znají algoritmus vyhodnocení výrazu pomocí dvou zásobníků, ještě popíšeme jeho úpravu na řešení s jedním zásobníkem. Řekněme, že ukládáme hodnoty na jeden zásobník a operátory na druhý zásobník. Tyto zásobníky jsou až na plus minus jedničku stejně velké, protože hodnoty a operátory se ve výrazu střídají. Můžeme tedy tyto dva zásobníky uložit prokládaně do zásobníku jednoho a získat tak řešení velmi podobné našemu předchozímu řešení. Jenom si musíme dát pozor na závorky, které se s hodnotami pravidelně střídat nemusí. To lze ošetřit například vložením nuly před otevírací závorku a jejím následným odstraněním při odebírání této otevírací závorky.

```
program vyraz;
var s : stack of char;
    c : char;

{ Vrátí operátor na vrcholu zásobníku. }
{ Pokud tam žádný není, vrátí '#'. }
function vrchni_op : char;
var num, op : char;
begin
   num := pop(s);
   if empty(s) then op := '#'
               else begin op := pop(s); push(s, op); end;
   push(s, num);
   vrchni_op := op;
end;

{ Vyhodnotí operátor na vrcholu zásobníku. }
procedure vyhodnot;
var a, b, op, vysledek : char;
begin
   b := pop(s);
   op := pop(s);
   a := pop(s);
   if op = '&' then vysledek := a='1' and b='1';
   if op = '|' then vysledek := a='1'  or b='1';
   if op = '(' then vysledek :=           b='1';
   if vysledek then push(s, '1') else push(s, '0');
```

```
end;

begin
   while read(c) do case c of
     '0' ,
     '1' : push(s, c);         { Číslo uložíme na zásobník. }
     '&' : begin               { Vyhodnotit všechny '&'. }
              while vrchni_op = '&' do vyhodnot;
              push(s, c);
           end;
     '|' : begin               { Vyhodnotit všechny '&' nebo '|'. }
              while vrchni_op in ['&', '|'] do vyhodnot;
              push(s, c);
           end;
     '(' : begin               { Před závorku uložíme pomocnou nulu. }
             push(s, '0');
             push(s, c);
           end;
     ')' : begin               { Vyhodnotit až do '('. }
              while vrchni_op <> '(' do vyhodnot;
              vyhodnot;        { Odstraní '(' z vrcholu zásobníku. }
           end;
   end;
   while vrchni_op <> '#' do vyhodnot;
   write(pop(s));
end.
```