52. ročník Matematické olympiády – 2002/2003

Komentáře k úlohám domácího kola kategorie P

Tento pracovní materiál není určen přímo studentům – řešitelům olympiády. Má pomoci učitelům na školách při přípravě konzultací a pracovních seminářů pro řešitele soutěže, pracovníkům krajských výborů MO slouží jako podklad pro opravování úloh domácího kola MO kategorie P. Studentům lze tyto komentáře poskytnout až po termínu stanoveném pro odevzdání řešení úloh domácího kola MO-P jako informaci, jak bylo možné úlohy správně řešit, a pro jejich odbornou přípravu na účast v krajském kole soutěže.

P-I-1 Čajovník

Pro řešení úlohy si vypůjčíme terminologii z teorie grafů. Významná místa na čajovníku budeme nazývat *vrcholy*, části kmene čajovníku mezi dvěma významnými místy pak *hrany*. Vrcholy spolu s hranami si pojmenujeme *strom* (pokud bychom chtěli být opravdu přesní, měli bychom jej nazvat *grafem*. My ale víme, že náš čajovník nemá žádný cyklus a graf s touto vlastností se nazývá strom). Počet hran, které vedou z nějakého vrcholu v (tedy vlastně počet částí kmene, které vedou z významného místa v), nazveme *stupněm* vrcholu v.

Nejdříve si všimneme, že každý strom s alespoň dvěma vrcholy má alespoň jeden vrchol stupně jedna (takovýto vrchol se nazývá *list*). Tento vrchol můžeme snadno nalézt následovně. Začneme strom prohledávat v libovolném vrcholu. Pokud ještě nejsme v listu, přejdeme do libovolného sousedního (tzn. připojeného hranou) vrcholu, ve kterém jsme dosud nebyli. Jelikož ve stromu nejsou cykly, musí takovýto vrchol vždy existovat. Protože vrcholů je konečný počet, musíme jednou skončit – a to můžeme pouze v listu.

Nyní ukážeme, že součet stupňů všech vrcholů v libovolném stromu je $2N-2$ (kde N je počet vrcholů). Naopak platí, že máme-li N kladných celých čísel se součtem $2N-2$, pak existuje strom s N vrcholy majícími tyto stupně. Z toho už je jasné, že stačí zjistit, zda je součet čísel na vstupu roven $2N-2$, a podle výsledku vypsat patřičnou zprávu.

První tvrzení dokážeme indukcí podle počtu vrcholů. Strom o dvou vrcholech obsahuje jedinou hranu. Součet stupňů vrcholů je tedy $1+1=2$ a naše tvrzení platí. Pokud má strom více vrcholů, víme z předchozího pozorování, že má list. Když tento list odebereme (tedy zrušíme vrchol a hranu, která ho připojuje ke zbytku stromu), získáme zřejmě opět strom. Pro něj z indukčního předpokladu platí, že součet stupňů je $2\cdot(N-1)-2=2N-4$. Protože v původním stromu měl jeden vrchol stupeň o jedna vyšší (ten, ke kterému byl připojen list) a byl v něm navíc list, je součet stupňů v původním stromu $2N-4+2=2N-2$. Tím je první tvrzení dokázáno.

Druhé tvrzení dokážeme indukcí dle počtu členů posloupnosti: Nechť máme posloupnost dvou kladných celých čísel, jejichž součet je $2\cdot 2-2=2$. Tato čísla tedy mohou být pouze dvě jedničky. Pro ně jsou zřejmě odpovídajícím stromem dva vrcholy spojené hranou. Pokud má posloupnost více než dvě čísla, musí zřejmě obsahovat alespoň jednu jedničku (jinak by součet N čísel byl alespoň $2N$ a ne $2N-2$). Analogicky musí také obsahovat alespoň jedno číslo větší než jedna. Když z posloupnosti vypustíme jednu jedničku a jedno z čísel větších než jedna snížíme o jedna, získáme posloupnost čísel o jedna kratší se součtem $2N-2-2=2\cdot(N-1)-2$. Z indukčního předpokladu tedy existuje strom na $N-1$ vrcholech s příslušnými stupni vrcholů. Když do stromu přidáme jeden list a připojíme ho hranou k vrcholu, který odpovídá číslu, jež jsme zmenšovali o jedničku, získáme přesně strom pro naši původní posloupnost. Tím je dokázáno i druhé tvrzení.

Časová složitost algoritmu je $O(N)$, paměťová $O(1)$.

```
program Caj;
var
  N, i, Suma, Casti : Integer;
  vstup, vystup : Text;
begin
  Assign(vstup, 'caj.in');
  Assign(vystup, 'caj.out');
  Reset(vstup);
  Rewrite(vystup);
  Suma := 0;
  ReadLn(vstup, N);
  for i := 1 to N do begin
```

```
    Read(vstup, Casti);
    Suma := Suma + Casti;
  end;
  if Suma = 2*(N-1) then
    WriteLn(vystup, 'EXISTUJE')
  else
    WriteLn(vystup, 'NEEXISTUJE');
  Close(vstup);
  Close(vystup);
end.
```

P-I-2 Knihovna

Předvedeme si dvě možná řešení této úlohy. Obě jsou založena na metodě zvané dynamické programování: Úloha se nejprve vyřeší pro podúlohu velikosti 1. Tohoto řešení se použije pro nalezení řešení podúlohy velikosti 2. Takto nalezených řešení se použije pro vyřešení podúlohy velikosti 3 atd. V našem případě bude velikost podúlohy určená počtem knih, které chceme do skříně umístit.

První řešení je založeno na vytvoření dvojrozměrného pole A o rozměrech $N \times V$, kde N je celkový počet knih, které máme do skříně umístit, a V je maximální výška skříně; V je v našem případě rovno 250 podle zadání úlohy. Hodnota $A[i,j], 0 \leq i \leq N, 1 \leq j \leq V$ udává minimální možnou šířku skříně výšky j, do které lze umístit prvních i knih. Pokud do skříně výšky j prvních i knih nelze umístit, tj. některá z těchto knih je vyšší než $j-2$ cm, pak je hodnota $A[i,j]$ rovna nějaké speciální hodnotě, např. -1. Popíšeme si, jak lze v čase $O(N)$ spočítat hodnotu $A[i_0,j_0]$, máme-li již spočítány hodnoty $A[i,j]$ pro $i < i_0$. Pokud je $i_0 = 0$, pak zřejmě $A[i_0,j_0] = 0$ cm. Pokud existuje $i, 1 \leq i \leq i_0$, takové, že výška v_i i-té knihy je větší než $j_0 - 2$ cm, pak prvních i knih nelze do skříně výšky j_0 umístit a $A[i_0,j_0]$ bude rovno -1. Ve zbylých případech určíme hodnotu $A[i_0,j_0]$ následovně: Pro $0 \leq i < i_0$ zkusíme umístit na poslední poličku skříně $(i+1)$-ní až i_0-tou knihu a prvních i knih dáme na předcházející poličky; výška poslední poličky by tedy musela být alespoň $v = \max_{i+1 \leq k \leq i_0} v_k$ a můžeme předpokládat, že je právě v. Šířka této poličky musí být alespoň $i_0 - i$. Pokud $A[i, j_0 - v - 1]$ je rovno -1, pak nelze vytvořit skříň výšky j_0, která by obsahovala prvních i_0 knih a na poslední poličce by z nich měla posledních $i_0 - i$. V opačném případě je nejmenší šířka skříně výšky j_0, která obsahuje prvních i_0 knih a na poslední poličce má z nich umístěno posledních $i_0 - i$, rovna $\max\{A[i, j_0 - v - 1], i_0 - i\}$. Nejmenší z těchto výrazů pro $0 \leq i < i_0$ bude roven hledané hodnotě $A[i_0,j_0]$. Výše popsaný výpočet lze provést v čase $O(N)$, budeme-li postupovat od $i = i_0 - 1$ k $i = 0$; v takovém případě lze $v = \max_{i+1 \leq k \leq i_0} v_k$ spočítat z v pro hodnotu i o 1 větší v konstantním čase. Hodnota pole $A[N, 250]$ je hledanou minimální možnou šířkou skříně. Pokud chceme zároveň nalézt i rozmístění knih do skříně a výšky jednotlivých poliček, zavedeme si ještě pomocné pole $B[i,j], 0 \leq i \leq N, 1 \leq j \leq V$, do jehož položky $B[i_0,j_0]$ si při výpočtu hodnoty $A[i_0,j_0]$ uložíme to i, pro které je šířka skříně minimální při výšce j_0. Z hodnoty $B[N, 250]$ určíme počet knih, které jsou v optimálním řešení na poslední poličce; tato hodnota nám umožní spočítat výšku skříně bez poslední poličky a počet knih v těchto poličkách. Z příslušné hodnoty v poli B určíme počet knih na předposlední poličce a takto postupujme, dokud nedosáhneme první poličky. Vzhledem k velikosti pole A jsme si právě popsali algoritmus, jehož časová složitost je $O(VN^2)$ a paměťová složitost $O(VN)$.

Nyní si popíšeme druhé možné řešení. Nejprve si ukážeme, jak lze v čase $O(N^2)$ rozhodnout, zda lze knihy umístit do skříně šířky s a výšky V. K tomu si vytvoříme pomocné pole $A[i], 0 \leq i \leq N$, které udává minimální výšku skříně šířky s, do které lze umístit prvních i knih. Pokud $A[N] > V$, pak knihy nelze umístit do skříně šířky s a výšky V; v opačném případě je lze do skříně s těmito rozměry umístit. K určení hodnot v poli A opět použijeme dynamické programování. Hodnota $A[0]$ je 1 cm, což je speciální případ obecného vztahu „součet výšek poliček + (počet poliček + 1) $\times$ 1 cm“ pro výšku skříně. Popíšeme, jak lze určit hodnotu $A[i_0]$, pokud známe hodnoty $A[i]$ pro $0 \leq i < i_0$. Zvolme $i_0 - s \leq i < i_0$; na poslední poličku chceme umístit v takovémto případě posledních $i_0 - i$ knih (proto podmínka $i_0 - s \leq i$). Výška skříně je pak rovna $A[i] + 1 + \max_{i<k\leq i_0} v_k$; nejmenší z těchto výrazů pro i, $i_0 - s \leq i < i_0$ je hledaná hodnota $A[i_0]$. Hodnotu $A[i_0]$ lze spočítat v čase $O(N)$, pokud budeme postupovat od $i = i_0 - 1$ k $i = i_0 - s$ (potom lze $\max_{i<k\leq i_0} v_k$ spočítat z hodnoty pro $i+1$ v konstantním čase). Popsaná procedura v čase $O(N^2)$ s pamětí $O(N)$ rozhodne, zda lze zadaných N knih umístit do skříně šířky s a výšky V. Zbývá popsat, jak lze tuto proceduru použít pro vyřešení původní úlohy. Nejprve zkontrolujeme, že výška všech knih je nejvýše $V - 2$ cm = 248 cm a tedy že knihy lze vůbec umístit do nějaké skříně výšky V. K určení minimální šířky s_0 skříně použijeme metodu zvanou půlení intervalu. Budeme si udržovat dvě proměnné $s_1 \leq s_2$, které nám budou ohraničovat možný interval, ve kterém je hledaná šířka s_0, tj. $s_1 \leq s_0 \leq s_2$. Nejprve položíme $s_1 = 1$ a

$s_2 = N$. V každém kroku zvolíme $s = \lfloor (s_1 + s_2)/2 \rfloor$ a pomocí výše popsané procedury zkontrolujeme, zda lze našich N knih umístit do skříně výšky V a šířky s. Pokud knihy lze do takové skříně umístit, položíme $s_2 = s$; v opačném případě položíme $s_1 = s + 1$. Celý postup opakujeme, dokud se hodnoty s_1 a s_2 liší, tj. dokud nanelezneme hledanou hodnotu s_0. Všimněte si, že v každém kroku se rozdíl $s_2 - s_1$ zmenší alespoň o 1 (kdybychom při volbě s použili horní celou část místo dolní celé části, nebylo by toto tvrzení pravdivé) a tento rozdíl se zmenší zhruba na polovinu. Tedy po $O(\log N)$ krocích nalezneme hledanou optimální šířku skříně s_0. Výšky poliček a rozmístění knih lze nalézt podobně jako v předcházejícím algoritmu zavedením pomocného pole B, do kterého si budeme ukládat počet knih na poslední poličce v optimálním řešení. Celková časová složitost právě popsaného algoritmu je tedy $O(N^2 \log N)$ a paměťová složitost je $O(N)$.

Zbývá vyřešit otázku, který ze dvou popsaných algoritmů je lepší. Odpověď je, že ani jeden není lepší. Vzhledem k zadání úlohy, kde V je omezeno, je časová složitost prvního algoritmu sice $O(N^2)$ a paměťová pouze $O(N)$, ale multiplikativní konstanta skrytá ve „velkém O“ je lineární s V; na druhou stranu paměťová složitost druhého algoritmu je pouze $O(N)$, kde multiplikativní konstanta je nezávislá na výšce. Dle výše popsaného postupu dokonce druhý algoritmus pracuje s poli, která jsou 250-krát menší. Stejně v časové složitosti člen $\log N$ bude menší než člen V vyskytující se v časové složitosti prvního algoritmu. První algoritmus je tedy pro omezenou výšku V asymptoticky lepší, ale ve skutečnosti bude lepší než druhý popsaný algoritmus až pro velmi velké hodnoty N. Lze tedy říci, že druhý algoritmus je použitelnější.

```
program p_1_2;
{ Řešení úlohy P-I-2 verze 1 }
const MAXN=100;
      VYSKA_MISTNOSTI=250;
var vyska: array[1..MAXN] of word;      { výšky knih }
    n: word;                            { počet knih }
    A: array[0..MAXN,1..VYSKA_MISTNOSTI] of integer;
                                        { pole minimálních šířek skříně }
    B: array[0..MAXN,1..VYSKA_MISTNOSTI] of word;
                                        { počty knih na poslední poličce
                                          v optimálním řešení }
function max(a,b:longint):longint;
  begin
    if a<b then max:=b else max:=a
  end;
procedure vypis(n: word; v: word);
  var i,k:word;
  begin
    if n=0 then
      begin
        writeln('Výška skříně: ',VYSKA_MISTNOSTI-v+1,' cm');
        exit;
      end;
    k:=0;
    for i:=n-B[n,v]+1 to n do k:=max(k,vyska[i]);
    vypis(n-B[n,v],v-k-1);
    writeln;
    writeln('Výška poličky: ',k,' cm');
    write('Knihy na poličce:');
    for i:=n-B[n,v]+1 to n do write(' ',i,'(',vyska[i],' cm)');
    writeln;
  end;
var i,j,k: word;
    maxvyska: word;
begin
  readln(n);
  for i:=1 to n do read(vyska[i]);
```

```
  for i:=1 to n do
    if vyska[i]>VYSKA_MISTNOSTI-2 then
      begin
        writeln('Pro zadané rozměry knih neexistuje knihovna!');
        halt;
      end;
  for j:=1 to VYSKA_MISTNOSTI do A[0,j]:=0;
  for i:=1 to n do
    for j:=1 to VYSKA_MISTNOSTI do
      begin
        maxvyska:=vyska[i];
        A[i,j]:=-1;
        for k:=1 to i do
          begin
            maxvyska:=max(maxvyska,vyska[i-k+1]);
            if maxvyska+2>j then break;
            if A[i-k,j-maxvyska-1]=-1 then continue;
            if (A[i,j]=-1) or (A[i,j]>max(A[i-k,j-maxvyska-1],k)) then
              begin
                A[i,j]:=max(A[i-k,j-maxvyska-1],k);
                B[i,j]:=k;
              end;
          end;
      end;
  if A[n,VYSKA_MISTNOSTI]=-1 then
    begin
      writeln('Pro zadané knihy nelze knihovnu navrhnout.');
      halt;
    end;
  writeln('Optimální šířka skříně je ',A[n,VYSKA_MISTNOSTI],' cm.');
  vypis(n,VYSKA_MISTNOSTI);
end.

program p_1_2;
{ Řešení úlohy P-I-2 verze 2 }
const MAXN=1000;
      VYSKA_MISTNOSTI=250;
var vyska: array[1..MAXN] of word;      { výšky knih }
    n: word;                            { počet knih }
function lze_skrin(s: word; v: word; vypisovat: boolean):boolean;
  var A: array[0..MAXN] of word;        { pole s minimálními výškami knihoven }
      B: array[1..MAXN] of word;        { pole s počty knih na poličkách }
      maxvyska: word;
      i, j: word;
  begin
    A[0]:=1;
    for i:=1 to n do
      begin
        maxvyska:=vyska[i];
        A[i]:=A[i-1]+maxvyska+1;
        B[i]:=1;
        j:=i-1;
        while (j>0) and (i-j<s) do
          begin
            if maxvyska<vyska[j] then maxvyska:=vyska[j];
```

```
            if A[i]>A[j-1]+maxvyska+1 then
              begin
                A[i]:=A[j-1]+maxvyska+1;
                B[i]:=i-j+1;
              end;
            dec(j);
          end
      end;
    lze_skrin:= A[n] <= v;
    if not(vypisovat) then exit;
    i:=n;
    writeln('Výška skříně: ',A[n],' cm.');
    writeln('Výšky poliček ve skříni a jejich naplnění knihami od spodu knihovny:');
    while i>0 do
      begin
        writeln;
        writeln('Výška poličky: ',A[i]-A[i-B[i]]-1,' cm');
        write('Knihy v poličce:');
        for j:=i-B[i]+1 to i do
          write(' ',j,'(',vyska[j],' cm)');
        writeln;
        i:=i-B[i];
      end
  end;
var i:word;
    s1,s2:word;
begin
  readln(n);
  for i:=1 to n do read(vyska[i]);
  for i:=1 to n do
    if vyska[i]>VYSKA_MISTNOSTI-2 then
      begin
        writeln('Pro zadané rozměry knih neexistuje knihovna!');
        halt;
      end;
  s1:=1; s2:=n;
  while s1<s2 do
    if lze_skrin((s1+s2) div 2, VYSKA_MISTNOSTI, false) then
      s2:=(s1+s2) div 2
    else
      s1:=(s1+s2) div 2+1;
  writeln('Optimální šířka skříně je ',s1,' cm.');
  lze_skrin(s1,VYSKA_MISTNOSTI,true);
end.
```

P-I-3 Transformace

Máme zadán poslední sloupec setříděné tabulky. Základní myšlenka celého řešení spočívá v tom, že tímto je dán i první sloupec – stačí setřídit písmena posledního sloupce podle abecedy. Nyní využijeme toho, že jednotlivé řádky tabulky vznikly rotací nějakého řetězce; tedy je-li na některém řádku v prvním sloupci písmeno x a v posledním sloupci písmeno y, znamená to, že v původním řetězci bylo písmeno x za písmenem y (bráno cyklicky – tedy za posledním písmenem následuje první). Vzhledem k tomu, že každé písmeno se v řetězci vyskytuje nejvýše jednou, n řádků tabulky nám již určuje pořadí písmen v původním řetězci až na rotaci. Navíc máme zadáno, na kterém řádku se vyskytuje námi hledané slovo; tím máme určeno jeho první písmeno.

Sestrojit na základě této myšlenky algoritmus je již jednoduché. Písmena zadaného řetězce si setřídíme podle abecedy (vzhledem k tomu, že hodnoty jsou z omezeného rozsahu, nabízí se přihrádkové třídění) a vyrobíme si

tabulku, v níž bude každému písmenu přiřazeno jemu odpovídající následující písmeno. Pak začneme od písmene, o němž víme, že je první, a postupujeme od něj po následnících, přičemž rovnou vypisujeme výsledek, dokud se k tomuto písmeni nevrátíme.

Časová i paměťová složitost algoritmu jsou zjevně lineární v délce zadaného řetězce.

```
program bw;
const MAX = 100;
type slovo = array[1..MAX] of char;
var prvni_sloupec, posledni_sloupec : slovo;
    radek : integer;
    delka : integer;
    buckets : array[char] of boolean;
    naslednik : array[char] of char;
    s : string;
    i, l : integer;
    ch : char;

begin
  readln (s);                                   { načtení zadání }
  delka := length (s);
  for i := 1 to delka do
    posledni_sloupec[i] := s[i];
  readln (radek);

  for ch := #0 to #255 do                       { bucket sort }
    buckets[ch]:=false;
  for i := 1 to delka do
    buckets[posledni_sloupec[i]]:=true;
  l := 0;
  for ch := #0 to #255 do
    if buckets[ch] then
      begin
        inc (l);
        prvni_sloupec[l] := ch;
      end;

  for i:=1 to delka do                          { určení následníků }
    naslednik[posledni_sloupec[i]] := prvni_sloupec[i];

  ch := prvni_sloupec[radek];                   { výpis }
  for i := 1 to delka do
    begin
      write (ch);
      ch := naslednik[ch];
    end;
  writeln;
end.
```

P-1-4 Reverzibilní výpočty: Políčko pole

Políčit na pozici příslušného prvku prohledáváním pole políčko po políčku přinese požadovanou proceduru, přesto poněkud pomalou. Pokračujme proto, přátelé, v přemýšlení:

Sestrojíme reverzibilní verzi binárního vyhledávání, místo tradičního zápisu pomocí cyklu `while` ovšem použijeme rekurzi. Zavedeme si podproceduru `Hledej(var l,p:word)`, která bude vyhledávat hodnotu *co* v úseku $X_l, X_{l+1}, \ldots, X_p$ a výsledek přičte k proměnné *kde*. Zařídí to tak, že si nejdříve spočte pozici prostředního prvku

X_m zadaného úseku (pokud má úsek sudou délku, zaokrouhlíme libovolným směrem) a podle jeho hodnoty zjistí, ve které polovině úseku má hledání pokračovat: pokud $X_m < co$, pak od $m+1$ do r, je-li $X_m > co$, tak od l do $m-1$. Na tento úsek pak zavoláme tutéž proceduru rekurzivně, ale nesmíme zapomenout, až se vrátí, m ještě odpočítat. Nastane-li kdykoliv při porovnávání rovnost, právě jsme hodnotu *co* našli a po zvýšení *kde* o m se z procedury vracíme. Dospějeme-li v rekurzi k úseku nulové délky ($r < l$), vracíme se s prázdnou, tedy aniž bychom *kde* jakkoliv měnili.

Podle tohoto algoritmu již snadno vytvoříme program, pro odpočítávání v něm použijeme příkaz `wrap`:

```
procedure Najdi(var n:word; var X:array [1..n] of word; var co,kde:word);
   var one:word;
   procedure Hledej(var l,r:word);
   var m:word;
   begin
      if l<=r then                        { úsek není prázdný }
         wrap m += (l+r) div 2            { spočteme střed }
         on
            if X[m]=co then               { našli jsme }
               kde += m
            else if X[m]<co then begin    { postoupíme do pravého úseku }
               wrap m += 1
               on Hledej(m,r)
               end
            else begin                    { do levého }
               wrap m -= 1
               on Hledej(l,m)
               end
   end;
begin
   wrap one += 1
   on Hledej(one,n)
end;
```

Postoupíme-li v rekurzi o úroveň hlouběji, zmenší se prohledávaný úsek minimálně o polovinu, takže po nejvýše $\lceil \log_2 n \rceil$ rekurzivních voláních buďto hledanou hodnotu odhalíme nebo dospějeme k úseku nulové délky, kde rekurze rovněž končí. Časová složitost tedy činí $O(\log n)$ a paměťová taktéž (pro každou úroveň rekurze spotřebujeme konstantní množství paměti).