INFORMATIKA

Maximální palindrom (Úlohy z MO – kategorie P, 24. část)

PAVEL TÖPFER

Matematicko-fyzikální fakulta UK, Praha

Tentokrát se v našem seriálu zajímavých úloh z Matematické olympiády – kategorie P vrátíme hodně daleko do historie. Kategorie P vznikla ve 35. ročníku MO ve školním roce 1985/86 a hned v následujícím 36. ročníku (tzn. ve školním roce 1986/1987) byla v domácím kole zadána následující jednoduchá úloha s velmi krátkým zadáním.

⋆⋆⋆

Konečnou posloupnost čísel nazveme *symetrickou*, jestliže se nezmění, zapíšeme-li její prvky v obráceném pořadí. Navrhněte algoritmus, který pro libovolnou konečnou posloupnost kladných celých čísel určí délku jejího nejdelšího souvislého úseku, který je symetrickou posloupností. Například pro posloupnost 3, 1, 2, 3, 2, 1, 4, 2, 1 je tato délka 5 (neboť nejdelším souvislým symetrickým úsekem je 1, 2, 3, 2, 1).

⋆⋆⋆

K zadání ještě doplňme, že souvislému symetrickému úseku nějaké posloupnosti (například posloupnosti znaků, tzn. textu) se také říká *palindrom*. Úkolem tedy je určit délku maximálního palindromu v zadané

posloupnosti. Úloha má jistě řešení pro každou neprázdnou posloupnost, v krajním případě je výsledkem číslo 1, pokud daná posloupnost žádný delší palindrom neobsahuje (každý prvek posloupnosti je palindromem délky 1). Nepožaduje se určit maximální palindrom samotný, neboť ten na rozdíl od délky nemusí být určen jednoznačně, v dané posloupnosti může být obsaženo více různých palindromů téže maximální délky.

Při řešení úlohy budeme předpokládat, že známe délku zadané posloupnosti (tzn. počet jejích prvků N), abychom si celou posloupnost mohli uložit do pole a s tímto polem pak v programu dále pracovat. V době vzniku kategorie P Matematické olympiády ještě nebyla zadání soutěžních úloh formulována takto precizně, řešitelé pouze teoreticky navrhovali algoritmy, ale neprogramovali je. V tehdejší době studenti neměli prakticky žádný přístup k počítačům, takže ani v olympiádě se pochopitelně neřešily praktické úlohy u počítačů.

Nalézt nějaké správné řešení této úlohy je velmi snadné, jedná se o jednu z vůbec nejlehčích úloh, které dosud byly v soutěži MO – kategorie P zadány. Úloha je přesto velmi zajímavá, jakmile začneme navržená řešení posuzovat z hlediska časové složitosti algoritmů. Řešení s kubickou časovou složitostí $\mathrm{O}(N^3)$ je triviální a navrhne ho jistě každý bez delšího přemýšlení. Řešení s kvadratickou časovou složitostí $\mathrm{O}(N^2)$ již vyžaduje určitý nápad, ale nalézt ho není obtížné. Zato časově optimální řešení pracující v lineárním čase $\mathrm{O}(N)$ vymyslí jen málokdo a pro většinu čtenářů asi bude hodně velkým překvapením už jenom samotné zjištění, že je tuto úlohu možné v lineárním čase vyřešit.

Pro pořádek začneme od toho nejprimitivnějšího řešení. Zadanou posloupnost čísel si uložíme do pole, abychom se k prvkům posloupnosti mohli opakovaně vracet. Z posloupnosti pak budeme postupně vybírat všechny různé souvislé úseky a každý z nich vždy otestujeme, zda je symetrický. Průběžně si přitom pamatujeme délku dosud nejdelšího nalezeného palindromu. Po otestování symetrie všech úseků, které je možné ze zadané posloupnosti vybrat, budeme jistě znát hledanou délku maximálního palindromu. Popsané řešení dává sice správný výsledek, ale je značně neefektivní. Označíme-li počet čísel ve zpracovávané posloupnosti symbolem N, vyžaduje tento algoritmus provedení $\mathrm{O}(N^3)$ operací. Z posloupnosti délky N lze totiž vybrat řádově N^2 různých souvislých úseků, neboť N způsoby postupně volíme levou mez vybraného úseku a ke každé zvolené levé mezi v průměru $N/2$ způsoby volíme pravou mez. K otestování symetričnosti každého z nich je pak třeba provést $\mathrm{O}(N)$ porovnání.

Popsaný algoritmus jsme naprogramovali v Pascalu ve tvaru funkce *MaxPalindrom1*. Pomocná funkce *Palindrom1* slouží k testování, zda je zvolený úsek čísel v poli A symetrický.

```
function Palindrom1(var A: Pole; Odkud, Kam: integer):
boolean;
{test, zda úsek pole A mezi indexy Odkud -- Kam je palindrom}
var i, j: integer;
begin
i:=Odkud; j:=Kam;
while (i<j) and (A[i]=A[j]) do
  begin i:=i+1; j:=j-1 end;
Palindrom1:= i>=j
end; {function Palindrom1}

function MaxPalindrom1(var A: Pole; N: integer): integer;
var Leva, Prava, MaxDelka: integer;
begin
MaxDelka:=0;
for Leva:=1 to N do
  for Prava:=Leva to N do
    if Palindrom1(A, Leva, Prava) and
       (Prava-Leva+1 > MaxDelka) then
       MaxDelka:=Prava-Leva+1;
MaxPalindrom1:=MaxDelka
end; {function MaxPalindrom1}
```

Než přikročíme k asymptoticky lepšímu řešení, ukážeme si, že určité rozdíly v rychlosti výpočtu mohou být i mezi různými programy s časovou složitostí $O(N^3)$ založenými na právě popsané myšlence systematicky probírat a testovat všechny souvislé úseky. Lepší organizací práce je možné alespoň částečně snížit počet prováděných operací a zkrátit tak průměrnou délku výpočtu. Jestliže jsme už například našli palindrom délky 5, je zbytečné ztrácet čas testováním symetrie všech dalších souvislých úseků kratších než 6. I kdybychom nějaký takový kratší symetrický úsek nalezli, na celkový výsledek to stejně nebude mít vliv. Dobrým nápadem na zrychlení výpočtu proto je prohlížet úseky v pořadí podle jejich délky, počínaje nejdelším z nich. Díky tomuto uspořádání můžeme vybírání a tes-

tování úseků ukončit, jakmile najdeme první palindrom. Všechny další by již byly stejně dlouhé nebo kratší a nemohly by tedy ovlivnit celkový výsledek. Ušetříme tak práci s testováním všech kratších úseků. Algoritmus má ovšem i nadále časovou složitost v nejhorším případě $O(N^3)$, zlepšili jsme pouze jeho průměrnou časovou složitost. Zrychlení bude významné tehdy, jestliže v posloupnosti existuje nějaký dlouhý palindrom. Má-li nejdelší palindrom v posloupnosti délku 1, neušetříme oproti předchozímu řešení téměř nic.

Následující funkce *MaxPalindrom2* realizuje popsanou úpravu řešení. K testování symetrie úseků využívá stejnou pomocnou funkci *Palindrom1*, kterou jsme použili v předchozím případě, takže ji zde znovu neuvádíme.

```
function MaxPalindrom2(var A: Pole; N: integer): integer;
var Leva, Delka: integer;
begin
for Delka:=N downto 1 do
  for Leva:=1 to N-Delka+1 do
    if Palindrom1(A, Leva, Leva+Delka-1) then
      begin MaxPalindrom2:=Delka; exit end;
end; {function MaxPalindrom2}
```

Zamysleme se nyní nad tím, jakou změnou algoritmu bychom mohli dosáhnout výraznějšího zrychlení výpočtu. V dosud popsaných algoritmech může existovat celá řada takových dvojic čísel, která jsou navzájem porovnávána vícekrát. Děje se tak při zkoumání úseků vnořených do sebe. Jestliže budou například během výpočtu prováděny testy úseků v rozmezí indexů $\langle 1, 10\rangle$ a $\langle 2, 9\rangle$ a pokud se první a desátý prvek sobě rovnají, potom budou druhý a devátý prvek posloupnosti mezi sebou zbytečně porovnávány opakovaně, při obou uvedených testech. K vylepšení algoritmu bude proto výhodné provádět najednou testování symetrie všech úseků se společným středem.

Musíme si uvědomit, že palindrom maximální délky může obsahovat buď lichý, nebo sudý počet čísel. Podle toho buď je jeho středem některý z prvků zadané posloupnosti, nebo jeho střed leží mezi dvěma sousedními prvky. Celkově tedy existuje $2N-1$ možných umístění středu palindromu (N prvků, $N-1$ pozic mezi nimi), které musíme všechny vyšetřit. Pro každé z těchto $2N-1$ míst určíme délku maximálního palindromu se středem ve zvoleném místě. Nalezení takového maximálního palindromu je

snadné. Postupně od zvoleného středu směrem k oběma okrajům posloupnosti porovnáváme dvojice čísel stejně vzdálených od středu. To provádíme tak dlouho, dokud jsou porovnávaná čísla shodná a zároveň také dokud nenarazíme na některý z okrajů posloupnosti. Pro každý z uvažovaných středů se vykoná maximálně $N/2$ porovnání, takže celkově se provede nejvýše N^2 porovnání čísel. Algoritmus má tedy časovou složitost $O(N^2)$. Výsledkem výpočtu je maximum z délek maximálních palindromů, které jsme získali pro jednotlivé uvažované středy.

Také tento algoritmus jsme pro vás naprogramovali v Pascalu. Pomocná funkce *Palindrom3* tentokrát slouží k určení délky maximálního palindromu se zadaným středem. Můžeme ji použít jak pro hledání maximálního palindromu liché délky se středem v některém prvku posloupnosti, tak i pro hledání maximálního palindromu sudé délky se středem mezi dvěma sousedními prvky. To je realizováno pomocí dvojice parametrů *Stred1*, *Stred2*. V případě liché délky bude funkce volána se stejnou hodnotou obou těchto parametrů (oba určují index prostředního prvku zkoumaného úseku), při hledání palindromu sudé délky se budou při volání funkce hodnoty těchto parametrů lišit o 1 (parametry určují indexy sousedních prvků posloupnosti, mezi nimiž leží střed zkoumaného úseku).

```
function Palindrom3(var A: Pole; N, Stred1, Stred2: integer):
                    integer;
{délka maximálního palindromu se středem mezi prvky Stred1,
Stred2}
var i: integer;
begin
i:=0;
while (Stred1-i>=1) and (Stred2+i<=N) and (A[Stred1-i]=
       =A[Stred2+i]) do
  i:=i+1;
if Stred1=Stred2 then
  Palindrom3:=2*i-1
else
  Palindrom3:=2*i
end; {function Palindrom3}

function MaxPalindrom3(var A: Pole; N: integer): integer;
var Stred, Delka, MaxDelka: integer;
```

```
begin
MaxDelka:=1;
for Stred:=1 to N do
  begin
  Delka:=Palindrom3(A, N, Stred, Stred);  {lichá délka}
  if Delka>MaxDelka then MaxDelka:=Delka;
  Delka:=Palindrom3(A, N, Stred, Stred+1); {sudá délka}
  if Delka>MaxDelka then MaxDelka:=Delka
  end;
MaxPalindrom3:=MaxDelka
end; {function MaxPalindrom3}
```

Uvedený algoritmus můžeme ještě dále vylepšovat, ale zrychlení výpočtu nebude příliš významné a zápis algoritmu se stane o dost složitějším. Je například možné procházet středy testovaných úseků nikoliv postupně zleva doprava, jako v předchozím řešení, ale od středu celé posloupnosti směrem k oběma okrajům posloupnosti. Výpočet pak můžeme ukončit ve chvíli, kdy vzdálenost uvažovaného středu od bližšího okraje posloupnosti bude menší než polovina délky maximálního dosud nalezeného palindromu. V takové situaci je totiž jisté, že delší palindrom již nenajdeme. Tím se může snížit počet provedených volání pomocné funkce *Palindrom3*, pro „středy“ blízké k okrajům posloupnosti se volání již nebude provádět. Takto upravený algoritmus má ovšem nadále časovou složitost v nejhorším případě $O(N^2)$, zlepšili jsme pouze jeho průměrnou časovou složitost. Opakuje se podobná situace, jakou jsme již jednou viděli – zrychleni bude významné tehdy, jestliže v posloupnosti existuje nějaký dlouhý palindrom. Má-li naopak nejdelší palindrom v posloupnosti délku 1, výpočet vůbec nezrychlíme.

Následující funkce *MaxPalindrom4* využívá stejnou pomocnou funkci *Palindrom3* jako v předchozím případě, takže ji zde neuvádíme podruhé.

```
function MaxPalindrom4(var A: Pole; N: integer): integer;
var Stred1, Stred2, Delka, MaxDelka: integer;
begin
Stred1:=N div 2 + N mod 2; {střed posloupnosti}
Stred2:=N div 2 + 1;       {střed posloupnosti}
if N mod 2=0 then      {posloupnost sudé délky}
  MaxDelka:=Palindrom3(A, N, Stred1, Stred2)
```

```
else
  MaxDelka:=1;
while Stred1*2-1 > MaxDelka do
  begin
  Delka:=Palindrom3(A, N, Stred1, Stred1);
  if Delka>MaxDelka then MaxDelka:=Delka;
  Delka:=Palindrom3(A, N, Stred2, Stred2);
  if Delka>MaxDelka then MaxDelka:=Delka;
  Delka:=Palindrom3(A, N, Stred1-1, Stred1);
  if Delka>MaxDelka then MaxDelka:=Delka;
  Delka:=Palindrom3(A, N, Stred2, Stred2+1);
  if Delka>MaxDelka then MaxDelka:=Delka;
  Stred1:=Stred1-1;
  Stred2:=Stred2+1;
  end;
MaxPalindrom4:=MaxDelka
end; {function MaxPalindrom4}
```

Úlohu máme úspěšně vyřešenou, ale tím naše snaha o nalezení co nejefektivnějšího řešení nekončí. Jak jsme se již zmínili v úvodu, problém určení délky maximálního palindromu lze kupodivu řešit ještě mnohem lépe, a to v lineárním čase vzhledem k délce zkoumané posloupnosti. K tomuto časově optimálnímu řešení se vrátíme v příštím čísle.

(*Pokračování*)

(Autorkou úvodní ilustrace je *Mgr. Jaroslava Čermáková* ze Suchých Lazců.)

Palindrom

REDAKCE

Vyhledáme-li slovo palindrom na internetu, dozvíme se mnoho zajímavého. Předně zjistíme, že *palindrom je slovo, věta, číslo, obecně jakákoli sekvence symbolů, která má tu vlastnost, že ji lze číst v libovolném směru (zprava doleva nebo zleva doprava) a má vždy stejný význam.* Přitom se

neuvažují případné mezery a interpunkční znaménka, ani se nerozlišují malá a velká písmena. Palindrom je například slovo *nezařazen* nebo věta *Kuna nese nanuk*, i když při zpětném přesném čtení dostaneme *kunan esen anuK* V češtině také zanedbáváme čárky nad samohláskami (i háček nad e), takže věta *Kobyla má malý bok.* se řadí ke známým palindromům, i když obrácený postup dává *kob ýlam ám alyboK.* Jinak je to s písmeny s háčkem, ty se požaduji správně, i když třeba ve větě *Nekouká žák u oken* to je jedno, protože *ž* je střední písmeno.

Při jedné soutěži žáků základních škol dostali soutěžící za úkol připravit program, který u věty vložené z klávesnice zjistí, je-li to palindrom (pracuje se jen s anglickou abecedou a číslicemi).

Uvádíme zde postup tří soutěžících, jejichž programy mají určité odlišnosti. Formální části úvodu programu jsme sjednotili.

```
program PalindrA;

var
  I: Integer;
  Tex, L, P: string;
  C: Char;

begin
  WriteLn('Zjistovani palindromu');
  Write('Zadej text: ');
  ReadLn(Tex);
  L := '';
  P := '';
  for I := 1 to Length(Tex) do
  begin
    C := Tex[I];
    if C in ['a'..'z'] then
      C := UpCase(C);
    if C in ['A'..'Z','0'..'9'] then
    begin
      P := P + C;
      L := C + L
    end
  end;
```

```
  if L = P then
    WriteLn('je')
  else
    WriteLn('neni');
  ReadLn
end.
```

Soutěžící A si definuje dva na počátku prázdné řetězce L a P, pak čte postupně jednotlivé znaky ze zadaného textu a malá písmena převádí na velká. Jestliže načtený a případně upravený znak je velké písmeno nebo číslice, přidává jej k řetězci P zprava a k řetězci L zleva. Tím jsou ignorovány ostatní načtené znaky a dotaz, zda $L = P$ odpovídá na otázku, zda byl zadaný text palindrom.

Program funguje zcela správně, ale z hlediska naprogramování v něm najdeme určité nešikovnosti. Místo příkazů

```
C := Tex[I];
if C in ['a'..'z'] then
  C := UpCase(C);
```

by bylo lepší napsat do programu jednoduchý příkaz

```
C := UpCase(Tex[I]);
```

kterým bychom dosáhli úplně stejného efektu. Standardní funkce `UpCase` totiž sama testuje, jaký znak dostala ve vstupním parametru, malá písmena převádí na velká a ostatní znaky vrací beze změny.

Druhá nešikovnost je skryta ve vytváření znakového řetězce L. Zatímco příkaz `P := P + C` bude vykonán v konstantním čase (hodnota znaku C se jednoduše přidá na konec řetězce P), příkaz `L := C + L` má sám o sobě lineární časovou složitost. Celý znakový řetězec, který je obsahem proměnné L, se v něm totiž musí posunout o jeden znak doprava, aby vzniklo volné místo pro znak C přidávaný na jeho začátek. Rychlejší by bylo převrácený řetězec L vůbec nevytvářet, sestavit pouze řetězec P tvořený z přípustných znaků (všechna písmena jsou v něm již převedena velká) a pak jednoduše otestovat, zda je řetězec P symetrický:

```
I := 1; J := Length(P);
while I < J) and (P[I] = P[J]) do
begin
  Inc(I); Dec(J)
```

```
  end;
  if I >= J then WriteLn('je')
  else WriteLn('neni');

program PalindrB;

var
  I: Integer;
  Tex, L, P: string;
  C: Char;

begin
  WriteLn('Zjistovani palindromu');
  Write('Zadej text: ');
  ReadLn(Tex);
  L := '';
  P := '';
  for I := 1 to Length(Tex) do
  if Tex[I] in ['A'..'Z','a'..'z'] then
  begin
    C := UpCase(Tex[I]);
    L := L + C;
    P := C + P
  end;
  if L = P then WriteLn('je')
  else WriteLn('neni');
  ReadLn
end.
```

Soutěžící B postupuje v podstatě stejně s tím rozdílem, že každý znak testuje jen jednou a příkaz *UpCase* pak použije na všechna písmena (soutěžící A jen na malá). Avšak soutěžící B zapomněl na číslice.

```
program PalindrC;

var
  I, J: Integer;
  Tex, Xet: string;
```

```
begin
  WriteLn('Zjistovani palindromu');
  Write('Zadej text: ');
  ReadLn(Tex);
  Xet := ''; J := 0;
  for I := 1 to Length(Tex) do
    if Tex[I] in ['A'..'Z','a'..'z'] then
    begin
      Tex[I-J] := UpCase(Tex[I]);
      Xet := Tex[I-J] + Xet
    end else
    Inc(J);
  if Copy(Tex,1,Length(Xet)) = Xet then WriteLn('je')
  else WriteLn('neni');
  ReadLn
end.
```

Soutěžící C si volí dvě řetězcové proměnné *Tex* a *Xet*, do *Tex* je načten vstupní text. Pak vstupní text upravuje tak, že povolené znaky (písmena) klade od indexu 1 postupně za sebou zpět do proměnné *Tex* (do proměnné *Xet* je přidává zleva), čímž se načtený vstup přepisuje (tedy ničí). Odpověď pak dává otázka, zda řetězec *Xet* je roven první části (stejného rozsahu) řetězce *Tex*. Stejně jako B zapomněl i C na číslice.

Řešení soutěžícího C je na první pohled zdánlivě úsporné v tom, že ušetřil jednu proměnnou L, do níž soutěžící A a B ukládali nově vytvářený řetězec z přípustných znaků. Tato úspora však není vůbec podstatná a bylo jí dosaženo za cenu nižší přehlednosti a horší srozumitelnosti programu. Takový způsob programování vám proto rozhodně nemůžeme doporučit.

Soutěžící B a C mohli bez další změny svého programu rozšířit přípustnou množinu o znaky '0' ... '9', příkaz `UpCase` číslicím nijak neublíží. Tak by vyhověli podmínkám soutěže. Neefektivita spojená s vytvářením převráceného řetězce je v jejich programech stejná, jako v případě soutěžícího A, a šla by také napravit stejným způsobem.

Na závěr si uveďme ještě dva palindromy, první považujeme za velmi povedený a druhý za docela vtipný:

A man, a plan, a canal – Panama
V elipse spí lev.