

<input checked="" type="checkbox"/> Gr. 1, Dr. G. Kronberger	Name <u>PAPESH Konstantin</u>	Aufwand in h <u>8</u>
<input type="checkbox"/> Gr. 2, Dr. H. Gruber		
<input type="checkbox"/> Gr. 3, Dr. D. Auer	Punkte _____	Kurzzeichen Tutor / Übungsleiter _____ / _____

## 1. Längste gemeinsame Teilkette

(10 Punkte)

Entwickeln Sie eine Pascal-Prozedur

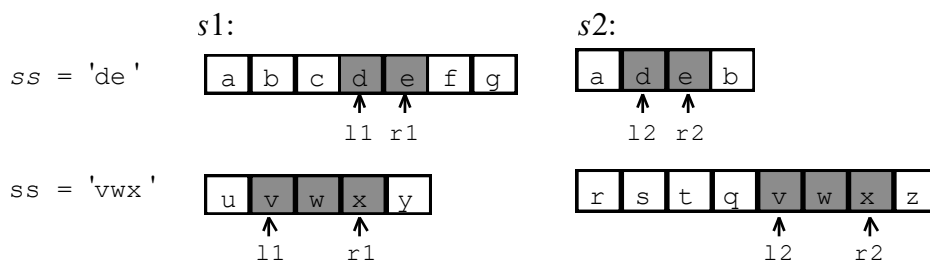
```
PROCEDURE FindLongestMatch (    s1,    s2:    STRING;
                               VAR ss: STRING;
                               VAR l1, r1, l2, r2: INTEGER);
```

die für zwei Zeichenketten  $s1$  und  $s2$  jene längste Teilkette (*substring*,  $ss$ ) findet, die sowohl in  $s1$  als auch in  $s2$  vorkommt. Die Anfangs- und die Endposition der gefundenen längsten Teilkette muss

- für  $s1$  in  $l1$  (*left* für die Anfangsposition) und  $r1$  (*right* für die Endposition) und
- für  $s2$  in  $l2$  und  $r2$

zurückgegeben werden.

Zwei Beispiele (untereinander angeordnet):



Gibt es keine gemeinsame Teilkette, also nicht einmal ein gemeinsames Zeichen, muss  $ss$  auf die leere Kette und es müssen  $lx$  und  $rx$  (für  $x = 1$  oder  $2$ ) auf 0 gesetzt werden.

Gibt es mehrere längste Teilketten, dann ist eine davon frei zu wählen und es sind für diese die entsprechenden Positionen in  $lx$  und  $rx$  zu liefern.

Für die Lösung von Teilaufgaben in *FindLongestMatch* können natürlich bekannte Verfahren zur Zeichenkettensuche eingesetzt werden, versuchen Sie aber, eine möglichst effiziente Gesamtlösung zu finden, und geben Sie eine Abschätzung der asymptotischen Laufzeitkomplexität dieser Gesamtlösung an.

## 2. Wildcard Pattern Matching

(4 + (5 + 5) Punkte)

Viele Programme, z. B. Texteditoren oder Kommandozeilen-Interpretierer diverser Betriebssysteme (engl. *shells*, z. B. *cmd* in Windows), verwenden eine spezielle Pattern-Matching-Variante, bei der es darum geht, festzustellen, ob eine Musterkette zu einer Zeichenkette passt. Denken Sie z. B. an den MS-DOS/Windows-Befehl *del \*.\** bzw. an das äquivalente UNIX-Kommando *rm \**. Hier muss festgestellt werden, ob die Musterkette (\*.\*) bzw. \*) zu einem Dateinamen im aktuellen Verzeichnis passt.

In Musterketten können so genannte Jokerzeichen (engl. *wildcards*) wie z. B. '?' oder '\*' vorkommen. Dabei steht das Jokerzeichen '?' in der Musterkette für *ein* beliebiges Zeichen in der Zeichenkette und das Jokerzeichen '\*' für eine *beliebige Anzahl* (null oder mehr) beliebiger Zeichen in der Zeichenkette. Diese Jokerzeichen können auch mehrfach in einer Musterkette vorkommen. Nehmen Sie an, dass sowohl die Muster- als auch die Zeichenkette durch das spezielle Endzeichen '\$' abgeschlossen ist, wobei das Endzeichen nicht innerhalb der Ketten vorkommt.

Folgende Tabelle zeigt einige einfache *Beispiele*:

Musterkette <i>p</i>	Zeichenkette <i>s</i>	<i>p</i> und <i>s</i> passen zusammen?
ABC\$	ABC\$	ja
ABC\$	AB\$	nein
ABC\$	ABCD\$	nein
A?C\$	AXC\$	ja
*\$	<i>beliebige auch leere Kette</i>	ja
A*C\$	AC\$	ja
A*C\$	XYZC\$	ja

- Erweitern/ändern Sie den *BruteForce*-Algorithmus so, dass er obige Aufgabenstellung bewältigt, jedoch als Jokerzeichen nur '?' (auch mehrfach) in der Musterkette vorkommen darf.
- Die zusätzliche Behandlung des Jokerzeichens '\*' ist mit den Standardalgorithmen leider nicht mehr so einfach möglich. Allerdings lässt sich das Problem relativ einfach mittels Rekursion lösen: (1) Definieren Sie zuerst ein rekursives Prädikat *Matching(p, s)*, das *true* liefert, wenn *p* und *s* zusammenpassen, sonst *false*. Zerlegen Sie dabei sowohl *p* als auch *s* geschickt in zwei Teile: in das erste Zeichen und den Rest. (2) Implementieren Sie das Prädikat *Matching* in Form einer rekursiven Funktion und testen Sie diese ausführlich.

# ADF2x & PRO2X Algorithmen & Datenstrukturen und OOP

## – SS 2018

### Übungsabgabe 3

Konstantin Papesh

18. April 2018

#### Zusammenfassung

In dieser Übung wird zuerst aus zwei Strings die größte Teilkette ermittelt. Im zweiten Teil wird dann mithilfe von Rekursion das Jokerzeichen `*` eingeführt, welches mehrere Buchstaben im Suchstring ersetzen kann.

### 3.1 Längste gemeinsame Teilkette

Es sind zwei Teilketten gegeben, aus diesen soll die größte gemeinsame Teilkette ermittelt werden.

#### 3.1.1 Lösungsidee

Ein einfacher Brute-Force Algorithmus vergleicht `s1` mit `s2`, sollte kein Match sein verschiebt er `s2` einmal nach links und wiederholt die Aktion. Sind `s1` und `s2` an der Stelle gleich, wird eine Schleife aufgerufen, welche wiederum immer eine Stelle weiter geht und so lange `s1` und `s2` vergleicht, bis eine Unstimmigkeit gefunden wird. Dabei wird bei jeder Übereinstimmung ein Counter erhöht, welcher der Länge des Teilstrings entspricht. Sobald eine Unstimmigkeit auftritt, wird die Länge des Teilstrings mit der Länge des bis jetzt längsten Teilstrings verglichen. Ist der momentane Teilstring länger, werden die Werte `l1, r1, l2, r2` gespeichert.

Ist der Vergleich am Ende von `s1` angelangt, wird wieder von vorne begonnen, jedoch wird der erste Buchstabe von `s2` entfernt. So werden die Strings nach und nach abgearbeitet.

#### 3.1.2 Implementierung

**Listing 3.1:** main.pas

```
1 Program TestPM;
2
3 uses ModPatternMatching;
4
```

```

5 procedure findLongestMatch(s1, s2: string;
6                             var ss: string;
7                             var l1, r1, l2, r2: integer);
8 var
9     pos : integer;
10    l1Temp,r1Temp,l2Temp,r2Temp : integer;
11    s2Length : integer;
12    s2Temp : string;
13    lettersRemovedFromS2 : integer;
14 //s2temp : string;
15 longestLength : integer;
16 begin
17     s2Length := length(s2);
18     s2Temp := s2;
19     l1Temp := 0;
20     r1Temp := 0;
21     l2Temp := 0;
22     r2Temp := 0;
23     lettersRemovedFromS2 := 0;
24     //s2temp := s2; // we will shorten s2temp one letter at a time
25     longestLength := 0;
26     for pos := 1 to s2Length do begin
27         BruteForcePos(s1, s2Temp, l1Temp, r1Temp, l2Temp, r2Temp); // search for s2
28         in s1
29         if(r1Temp-l1Temp > longestLength) then begin
30             longestLength := r1Temp-l1Temp+1;
31             l1 := l1Temp;
32             r1 := r1Temp;
33             l2 := l2Temp + lettersRemovedFromS2;
34             r2 := r2Temp + lettersRemovedFromS2;
35         end;
36         delete(s2Temp,1,1); // remove first letter from s2
37         inc(lettersRemovedFromS2);
38     end;
39     ss := copy(s1,l1,longestLength);
40 end;
41
42 procedure title(name : string);
43 begin
44     writeln;
45     writeln(name);
46     writeln('-----');
47 end;
48
49 var
50     s1, s2 : string;
51     ss : string;
52     l1,r1,l2,r2 : integer;
53 begin
54     s1 := 'bba';
55     s2 := 'abbaabba';
56     ss := '';
57     l1 := 0;
58     r1 := 0;
59     l2 := 0;
60     r2 := 0;
61     title('findLongestMatch');

```

```

61   findLongestMatch(s1, s2, ss, l1, r1, l2, r2);
62   writeln(ss);
63   writeln(l1);
64   writeln(r1);
65   writeln(l2);
66   writeln(r2);
67 end.

```

**Listing 3.2:** ModPatternMatching.pas

```

1  unit ModPatternMatching;
2
3  interface
4  procedure BruteForcePos(s, p : string;
5                          var lsTemp, rsTemp, lpTemp, rpTemp : integer
6                          );
7
8  procedure WriteAndResetStats;
9
10 implementation
11 var numComp : longint;
12
13 function Eq(a,b : char) : boolean;
14 begin
15   inc(numComp);
16   Eq := (a = b);
17 end;
18
19 procedure WriteAndResetStats;
20 begin
21   writeln('Number of comparisons: ', numComp);
22   numComp := 0;
23 end;
24
25 procedure BruteForcePos(s, p : string;
26                         var lsTemp, rsTemp, lpTemp, rpTemp : integer
27                         );
28 var
29   i : integer;
30   pLen, sLen : integer;
31   curLength, longestLength : integer;
32 begin
33   pLen := Length(p);
34   sLen := Length(s);
35   i := 0;
36   curLength := 0;
37   longestLength := 0;
38   for i := 1 to sLen do begin
39     if sLen - i < longestLength then// remaining text to match is already
40       shorter than longest length
41       break;
42     if Eq(s[i],p[1]) then begin
43       curLength := 1;
44       while Eq(s[i+curLength],p[curLength+1]) and (curLength <= pLen) do begin
45         inc(curLength);
46       end;
47       if curLength > longestLength then begin

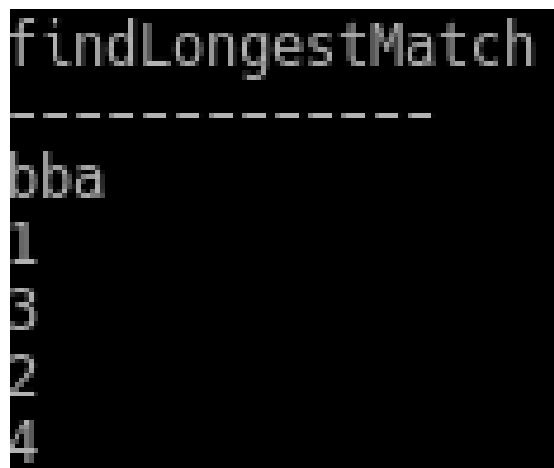
```

```

47         lsTemp := i;
48         rsTemp := i + curLength - 1;
49         lpTemp := 1;
50         rpTemp := curLength;
51         longestLength := rsTemp - lsTemp + 1;
52     end;
53     curLength := 0;
54 end;
55 end;
56 end;
57
58 begin
59 end.

```

### 3.1.3 Ausgabe



```

findLongestMatch
-----
bba
1
3
2
4

```

Abbildung 3.1: Ausgabe für  $s1=bba$  und  $s2=abbaabba$

### 3.1.4 Auswertung

Bei der Ausgabe wird zuerst der längste gemeinsame String ausgegeben und danach die jeweiligen Pointer auf die Start- und Endpunkte in den jeweiligen Strings. In diesem Beispiel war  $s1$  *bba* und  $s2$  *abbaabba*.

## 3.2 Wildcard Pattern Matching

Bei Suchen wurden sogenannte *Jokerzeichen* eingeführt, welche Verwendung finden, wenn ein Nutzer den genauen Wortlaut bzw. den vollen Suchstring nicht kennt.

### 3.2.1 Lösungsidee

a

Um diese Aufgabe zu Lösen muss die im Unterricht erstellte Funktion *Eq* nur um den Vergleich mit *?* erweitert werden.

b

Der Suchvorgang wird durch den Einsatz von Rekursion vereinfacht. Dabei ruft die Funktion sich selbst auf. Zuerst zerlegt die Funktion die beiden erhaltenen Strings  $s$  und  $p$  in den ersten Buchstaben und die restlichen. Dann wird überprüft, ob die ersten Buchstaben ident sind, oder  $p$  einem Jokerzeichen entspricht. Falls ja, werden die restlichen Strings wieder der gleichen Funktion übergeben, damit dort wieder eine Zerlegung statt finden kann. Dies geschieht so lange, bis das Ende der Zeichenkette  $s$  erreicht ist.

Matching([Erster Buchstabe|Rest]) :- Matching(Rest).  
 Abbruchbedingung: Rest = '\$';

### 3.2.2 Implementierung

a

**Listing 3.3:** ModPatternMatchingUnsharp.pas

```
1 function Eq(a,b : char) : boolean;
2 begin
3   inc(numComp);
4   Eq := (a = b) or (a = '?') or (b = '?');
5 end;
```

b

**Listing 3.4:** Recursion.pas

```
1 unit Recursion;
2
3 interface
4 function Matching(s, p : string) : boolean;
5
6 procedure WriteAndResetStats;
7
8 implementation
9 var numComp : longint;
10
11 function Eq(s,p : char) : boolean;
12 begin
13   inc(numComp);
14   Eq := (s = p) or (p = '?') or (p = '*');
15 end;
16
17 procedure WriteAndResetStats;
18 begin
19   writeln('Number of comparisons: ', numComp);
20   numComp := 0;
21 end;
22
23 function Matching(s,p : string) : boolean;
24 var
25   firstCharS, firstCharP : char;
26   remainingCharsS, remainingCharsP : string;
```

```

27     matchingSingle, matchingRemaining : boolean;
28 begin
29     firstCharS := s[1];
30     firstCharP := p[1];
31     matchingRemaining := TRUE;
32     matchingSingle := FALSE;
33     Matching := TRUE;
34     remainingCharsS := copy(s,2,length(s)-1);
35     remainingCharsP := copy(p,2,length(p)-1);
36     if(Eq(firstCharS,firstCharP)) then
37         matchingSingle := TRUE
38     else
39         Matching := FALSE;
40     if(remainingCharsS <> '$') and (matchingSingle <> FALSE) then
41         if(firstCharP = '*') then begin
42             if(firstCharS = remainingCharsP[1]) then
43                 matchingRemaining := Matching(s,remainingCharsP)
44             else
45                 matchingRemaining := Matching(remainingCharsS,p);
46         end
47     else
48         matchingRemaining := Matching(remainingCharsS,remainingCharsP);
49     Matching := matchingSingle and matchingRemaining;
50 end;
51 begin
52     numComp := 0;
53 end.

```

**Listing 3.5:** main.pas

```

1 Program main;
2
3 uses Recursion;
4
5 procedure title(name : string);
6 begin
7     writeln;
8     writeln(name);
9     writeln('-----');
10 end;
11
12 procedure printMatch(s, p : string);
13 begin
14     writeln(s, ' ', p, '=', Matching(s,p));
15 end;
16 var
17     firstCharS, firstCharP : char;
18     remainingCharsS, remainingCharsP : string;
19     s, p : string;
20 begin
21     printMatch('ABC$', 'ABC$');
22     printMatch('IMG_32323.png$', '*.png$');
23     printMatch('MANN$', 'M?NN$');
24     printMatch('MXXA$', 'M?AA$');
25 end.

```



### 3.2.3 Ausgabe

```
ABC$ ABC$=TRUE  
IMG_32323.png$ *.png$=TRUE  
MANN$ M?NN$=TRUE  
MXXA$ M?AA$=FALSE
```

**Abbildung 3.2:** Ausgabebeispiel für rekursives Vergleichen