

X

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

1. *m*-Ketten-Problem

(4 + 6 Punkte)

- a) *Definition*: Eine Zeichenkette ist eine *m*-Kette, wenn sie höchstens *m* unterschiedliche Zeichen enthält.

Beispiele: Die drei Zeichenketten *a*, *ab* und *abcbaac* sind 3-Ketten, die Zeichenkette *abcd* ist aber keine 3-Kette mehr, sondern eine 4-Kette. Eine Zeichenkette *s* der Länge *n* ist natürlich eine *n*-Kette, von Interesse ist aber das kleinste *m* für welches die Bedingung aus der Definition oben für *s* noch gilt. Entwickeln Sie daher eine möglichst effiziente Funktion

```
FUNCTION MinM(s: STRING): INTEGER;
```

zur Ermittlung des kleinsten *m* für eine Zeichenkette *s*.

- b) Gegeben sei eine nichtleere Zeichenkette *s* und eine ganze Zahl *m* mit $1 \leq m \leq \text{Length}(s)$. Entwickeln Sie eine möglichst effiziente Funktion

```
FUNCTION MaxMStringLen(s: STRING, m: INTEGER): INTEGER;
```

welche die Länge der längsten *m*-Kette, die als Teilkette in *s* enthalten ist, liefert.

2. Wildcard Pattern Matching

(4 + (5 + 5) Punkte)

Viele Programme, z. B. Texteditoren und Kommandozeilen-Interpretierer diverser Betriebssysteme (engl. *shells*), verwenden eine spezielle Variante der Zeichenkettensuche, die in der Musterkette Platzhalter (auch Jokerzeichen, engl. *wildcards*, genannt) zulässt. 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.

Jokerzeichen dürfen nur in Musterketten vorkommen: Dabei steht das Jokerzeichen *?* für *ein* beliebiges Zeichen in der Zeichenkette und das Jokerzeichen *** für eine *beliebige Anzahl* (null oder mehr) beliebiger Zeichen in der Zeichenkette. Jokerzeichen können auch gemischt und mehrfach in einer Musterkette vorkommen.

Nehmen Sie an, dass sowohl die Muster- als auch die Zeichenkette durch das spezielle Endezeichen *\$* abgeschlossen ist, welches innerhalb der Ketten nicht 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
AB <i>C</i> \$	AB\$	nein
ABC\$	ABCD\$	nein
A? <i>C</i> \$	A <i>X</i> C\$	ja
*\$	<i>beliebige auch leere Kette</i>	ja
A* <i>C</i> \$	AC\$	ja
A* <i>C</i> \$	A <i>XYZ</i> C\$	ja

- a) Erweitern/ändern Sie den *BruteForce*-Algorithmus für die Zeichenkettensuche so, dass er obige Aufgabenstellung bewältigt, jedoch als Jokerzeichen nur ? (auch mehrfach) in der Musterkette vorkommen darf.
 - b) Die zusätzliche Behandlung des Jokerzeichens * ist mit den Standard-Algorithmen 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 der Kette.
 - 2. Implementieren Sie das Prädikat *Matching* in Form einer rekursiven Funktion und testen Sie diese ausführlich.
-

Für besonders Interessierte: Implementieren Sie eine iterative Variante des Prädikats *Matching* aus 2.b.1, testen Sie diese ausführlich und stellen Sie Laufzeitvergleiche der rekursiven und iterativen Variante für lange Eingabeketten an. Als „**Belohnung**“ gibt es bis zu vier Zusatzpunkte.

Übung 2

Aufgabe 1

Lösungsidee

Die

```

1 PROGRAM kette;
  (* Implementation with lists *)
3
4 TYPE
5   nodePtr = ^listElement;
6   listElement = RECORD
7     next: nodePtr;
8     c: Char;
9   END; (* RECORD *)
10
11 (* Creates a new node*)
12 FUNCTION NewNode(c : Char): nodePtr;
13 VAR node : nodePtr;
14 BEGIN
15   New(node);
16   node^.next := NIL;
17   node^.c := c;
18   NewNode := node;
19 END;
20
21 (* Appends a Node to a List *)
22 PROCEDURE Append(var list : nodePtr; element : nodePtr);
23 VAR tmp : nodePtr;
24 BEGIN
25   if list = NIL THEN list := element ELSE
26   BEGIN
27     tmp := list;
28     WHILE tmp^.next <> NIL DO tmp := tmp^.next;
29
30     tmp^.next := element;
31   END;
32 END;
33
34 (* recursive; disposes every node in a list *)
35 PROCEDURE ClearList(var list : nodePtr);
36 BEGIN
37   IF list <> NIL THEN
38   BEGIN
39     IF list^.next = NIL THEN dispose(list) ELSE ClearList(list^.next);
40   END;
41 END;
42
43 (* Counts nodes in a list *)
44 FUNCTION CountNodes(n : nodePtr) : INTEGER;
45 VAR
46   count : INTEGER;
47 BEGIN
48   count := 0;
49   WHILE n <> NIL DO BEGIN

```

```

    Inc(count);
51   n := n^.next;
END;
53   CountNodes := count;
END;
55
PROCEDURE RemoveFirst(var list : nodePtr);
57   VAR
    temp : nodePtr;
59   BEGIN
    IF list <> NIL THEN BEGIN
61       temp := list;
        list := list^.next;
63       Dispose(temp);
    END;
65   END;

67   (* Check if char exists in the list;
    Returns TRUE OR FALSE *)
69   FUNCTION CharExists(list : nodePtr; c : Char): Boolean;
    VAR
71       n : nodePtr;
    BEGIN
73       n := list;

75       WHILE n <> NIL DO BEGIN
        IF n^.c = c THEN BEGIN
77             CharExists := TRUE;
            break;
79         END;
        n := n^.next;
81     END;

83     IF n = NIL THEN CharExists := FALSE;
    END;
85
    (* Counts different chars in a list;
    RETURNS 0 if empty*)
87   FUNCTION CountDistinct(list: nodePtr): Integer;
    VAR
89       temp, temp2 : nodePtr;
    BEGIN
91       BEGIN
        IF list <> NIL THEN
93             BEGIN
                temp := list;
95                 temp2 := NIL;

97                 WHILE temp <> NIL DO BEGIN
                    IF NOT CharExists(temp2, temp^.c) THEN Append(temp2, NewNode(temp^.c));
99                     temp := temp^.next;
                END;
101
                CountDistinct := CountNodes(temp2);
103                ClearList(temp2);
            END
105        ELSE
            CountDistinct := 0;
107        END;

```

```

109  (* Prints out list *)
PROCEDURE PrintList(list : nodePtr);
111  VAR
    n : nodePtr;
113  BEGIN
    n := list;
115
    WHILE n <> NIL DO BEGIN
117        Write(n^.c);
        n := n^.next;
119    END;
    WriteLn;
121  END;

123  (* Insert to string from a list RETURNS STRING*)
FUNCTION InsertinString(list : nodePtr): STRING;
125  VAR
    n : nodePtr;
127    s : STRING;
    BEGIN
129        n := list;
        s := '';
131
        WHILE n <> NIL DO BEGIN
133            s := Concat(s,n^.c);
            n := n^.next;
135        END;
        InsertinString := s;
137    END;

139  (* Implementation with Single Linked List *)
FUNCTION MinM(s: STRING) : Integer;
141  VAR
    i : Integer;
143    list : nodePtr;
    BEGIN
145        list := NIL;

147        FOR i := 1 TO Length(s) DO BEGIN
            IF NOT CharExists(list,s[i]) THEN Append(list, NewNode(s[i]));
149        END;

151        MinM := CountNodes(list);
    END;

153  (* Implementation with Single Linked List *)
FUNCTION MaxMStringLen(var longestS: STRING; s: STRING; m: Integer): Integer;
155  VAR
    i, count, tempCount, maxLength : Integer;
157    list : nodePtr;
    BEGIN
159        list := NIL;
        count := 0;
        maxLength := 0;
161
        FOR i := 1 TO Length(s) DO BEGIN
163
165

```

```

167 Append(list , NewNode(s[i]));
    tempCount := CountDistinct(list);

169 IF tempCount > m THEN
    BEGIN
171     RemoveFirst(list);
    END
173 ELSE
    count := count + 1;

175 IF count > maxLength THEN BEGIN
177     maxLength := count;
    longestS := InsertinString(list);

179 END;
181 END;
    MaxMStringLen := maxLength;
183 END;

185 VAR
    s1, s2, s3, longest : String;
187 BEGIN
    s1 := 'abcbaac';
189 s2 := 'abcd';
    s3 := 'abcdefgggggggggggggggggraabbdertzuioaaaaaaaaaaaaaaaaaaaaa';
191 s3 := 'abcdefgggggggggggggggggraabbdertzuioaaaaaaaaaaaaaaaaaaaaa';
    longest := '';

193 WriteLn('String 1: ', s1, #13#10#9, 'Min m: ', MinM(s1));
195 WriteLn('String 2: ', s2, #13#10#9, 'Min m: ', MinM(s2));
    WriteLn('String 3: ', s3, #13#10#9, 'Min m: ', MinM(s3));

197 WriteLn('_____');
199 WriteLn('String 1 mit m 2: ', s1, #13#10#9, 'MaxM: ', MaxMStringLen(longest, s1
    , 2),
    #13#10#9, 'Longest substring: ', longest, #13#10);
201 WriteLn('String 2 mit m 4: ', s2, #13#10#9, 'MaxM: ', MaxMStringLen(longest, s2
    , 4),
    #13#10#9, 'Longest substring: ', longest, #13#10);
203 WriteLn('String 3 mit m 5: ', s3, #13#10#9, 'MaxM: ', MaxMStringLen(longest, s3
    , 5),
    #13#10#9, 'Longest substring: ', longest, #13#10);
205 WriteLn('String 3 mit m 7: ', s3, #13#10#9, 'MaxM: ', MaxMStringLen(longest, s3
    , 7),
    #13#10#9, 'Longest substring: ', longest, #13#10);
207 END.

```

Kette.pas

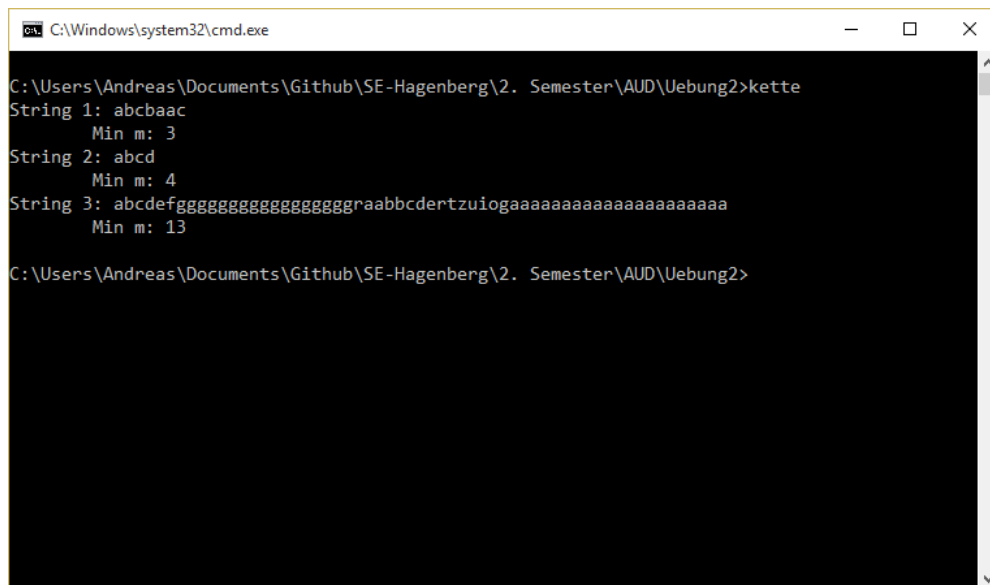


Abbildung 1: Ausgabe

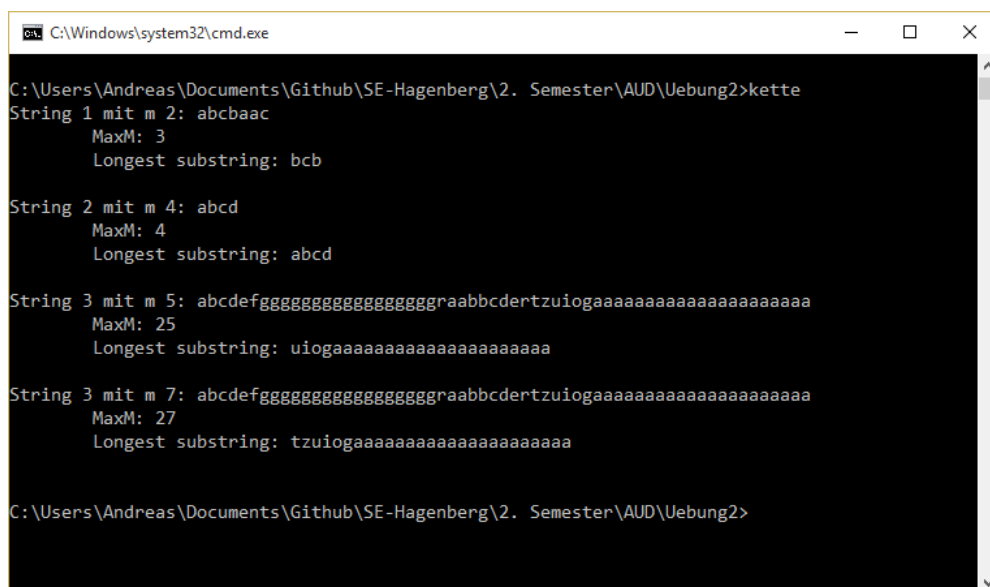


Abbildung 2: Ausgabe

Aufgabe 2

Lösungsidee

 \ddot{A}

```

1 PROGRAM wildcard;
2
3 (* Matching Going from the left to right
4    recursive *)
5 FUNCTION Matching(p, s : STRING): Boolean;
6 VAR
7     i, j: Integer;

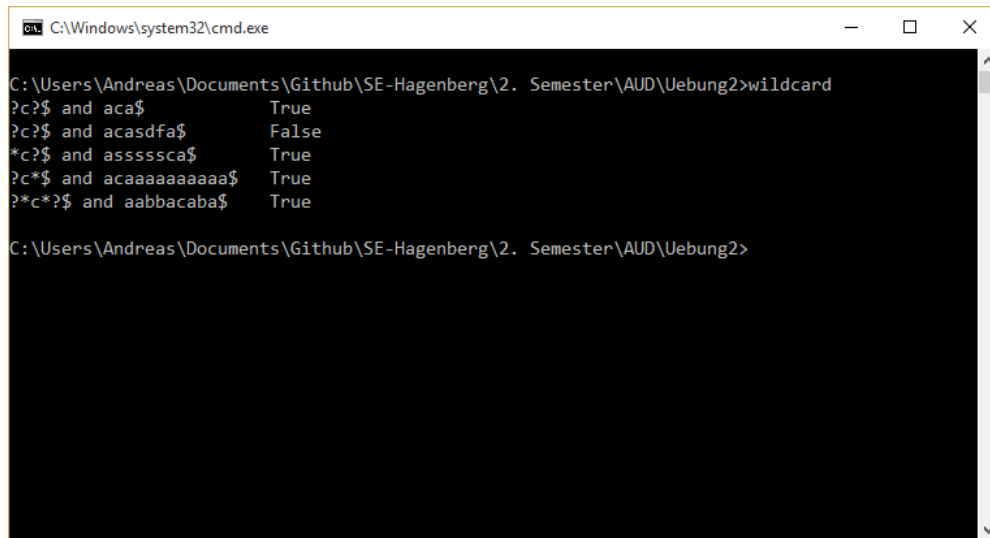
```

```

8 BEGIN
9   i := 1;
10  j := 1;
11
12  (* *)
13  WHILE (p[j] <> '*') AND (j <= length(p)) AND ((s[i] = p[j]) OR (p[j] = '?'))
14  ) DO BEGIN
15    i := i + 1;
16    j := j + 1;
17  END;
18
19  IF (p[j] <> '*') AND (i <= length(s)) THEN BEGIN
20    Matching := Matching(p, Copy(s, 2, length(s)))
21  END
22  ELSE IF (j <= length(p)) AND (i <= length(s)) THEN BEGIN
23    Matching := Matching(Copy(p, j + 1, Length(p)), Copy(s, 2, length(s)));
24  END
25  ELSE IF ((j >= length(p)) AND (i >= length(s))) OR ((j = length(p)) AND (p[
26  j] = '*')) THEN BEGIN
27    Matching := True;
28  END
29  ELSE Begin
30    Matching := False;
31  END;
32 END;
33 VAR
34 s, p : STRING;
35 BEGIN
36   s := '?c?$';
37   p := 'aca$';
38
39   IF Matching(s, p) THEN WriteLn(s, ' and ', p, #9#9, ' True')
40   ELSE WriteLn(s, ' and ', p, #9, ' False');
41
42   s := '?c?$';
43   p := 'acasdfa$';
44   IF Matching(s, p) THEN WriteLn(s, ' and ', p, #9, ' True')
45   ELSE WriteLn(s, ' and ', p, #9, ' False');
46
47   s := '*c?$';
48   p := 'assssca$';
49   IF Matching(s, p) THEN WriteLn(s, ' and ', p, #9, ' True')
50   ELSE WriteLn(s, ' and ', p, #9, ' False');
51
52   s := '?c*$';
53   p := 'aaaaaaaaaaa$';
54   IF Matching(s, p) THEN WriteLn(s, ' and ', p, #9, ' True')
55   ELSE WriteLn(s, ' and ', p, #9, ' False');
56
57   s := '?*c*?$';
58   p := 'aabbacaba$';
59   IF Matching(s, p) THEN WriteLn(s, ' and ', p, #9, ' True')
60   ELSE WriteLn(s, ' and ', p, #9, ' False');
61 END.

```

wildcard.pas



```
C:\Windows\system32\cmd.exe

C:\Users\Andreas\Documents\Github\SE-Hagenberg\2. Semester\AUD\Uebung2>wildcard
?c? and aca$      True
?c? and acasdfa$  False
*c? and asssssca$ True
?c* and aaaaaaaaa$ True
?*c* and aabbacaba$ True

C:\Users\Andreas\Documents\Github\SE-Hagenberg\2. Semester\AUD\Uebung2>
```

Abbildung 3: Ausgabe

Hier sieht man deutlich das die Verteilung der Elemente in der Tabelle sehr schlecht ist, die Verteilung der Elemente ist sehr nahe beieinander.