# Suchalgorithmen

## Maturavorbereitung

# Aufgabe 1

Beschreibe, was ein *Suchverfahren* ist und gib einige Beispiele dafür an.

# Aufgabe 1

Ein *Suchverfahren* ist ein Algorithmus, der in einem Suchraum Objekten mit bestimmten Eigenschaften sucht.

Arten von Suchalgorithmen

- Suche in ungeordneten Listen
- Suche in geordneten Listen
- Suche in Bäumen (wurde bisher nicht explizit behandelt)
- Suche in Graphen (Tiefen- und Breitensuche)
- Suche in Zeichenketten (naiv, Boyer-Moore-Horspool, DFA)

# Aufgabe 2

Welche Laufzeitkomplexitäten haben die folgenden Suchalgorithmen?

(a) Suche in einer unsortierten Liste

(b) Suche in einer sortierten Liste

(c) Tiefensuche in einem Graphen mit $|V|$ Knoten und $|E|$ Kanten in Adjazenzlistendarstellung

(d) Breitensuch in einem Graphen mit $|V|$ Knoten und $|E|$ Kanten in Adjazenzlistendarstellung

(e) Naive Suche nach einer Zeichenkette mit $m$ Zeichen in einer Zeichenkette mit $n$ Zeichen

# Aufgabe 2

(a) Suche in einer unsortierten Liste mit $n$ Elementen
$O(n)$

(b) Suche in einer sortierten Liste mit $n$ Elementen
$O(\log n)$

(c) Tiefensuche in einem Graphen mit $|V|$ Knoten und $|E|$ Kanten in Adjazenzlistendarstellung
$O(|E| + |V|$

(d) Breitensuch in einem Graphen mit $|V|$ Knoten und $|E|$ Kanten in Adjazenzlistendarstellung
$O(|E| + |V|$

(e) Naive Suche nach einer Zeichenkette mit $m$ Zeichen in einer Zeichenkette mit $n$ Zeichen
$m \cdot n$

# Aufgabe 3

(a) Schreibe eine Python-Funktion `is_element_of(L, item)`, deren Parameter eine Liste `L` und ein beliebigies Objekt `item` sind und die `True` zurückgibt, wenn sich das Objekt in der Liste befindet und `False` sonst.

```
1 def is_element_of(L, item)
2
3
4
5
6
7
8
```

(b) Welche Laufzeit(en) hat diese Funktion?

(c) Beschreibe, wie man die Funktion verändern müsste, wenn sie

# Aufgabe 3

(a) Python-Code:

```
1 def is_element_of(L, i
2     for i in range(len(
3         if L[i] == item:
4             return True
5     return False
6
7
8
```

(b) Genau genommen müssen zwei Fälle unterschieden werden.

| | $x \in L$ | $x \notin L$ |
|---|---|---|
| Best Case | $O(1)$ | $O(n)$ |
| Average Case | $O(n/2) = O(n)$ | $O(n)$ |
| Worst Case | $O(n)$ | $O(n)$ |

# Aufgabe 4

(a) Vervollständige die Python-Fuktion `binary_search(L, x)`, die in einer Liste `L` mit aufsteigend sortierten Elementen nach dem Element `x` sucht und den Index zurückgibt, an der das Element gefunden wurde und `False` sonst.

```
def binary_search(L, x):
    a = 0
    b = len(L) - 1
    while a <= b:
        m = (a + b) // 2
```

# Aufgabe 4

## (a) Python-Code

```
def binary_search(L,
    a = 0
    b = len(L) - 1
    while a <= b:
        m = (a + b) // 2
        if x < L[m]:
            b = L[m] - 1
        elif x > L[m]:
            a = L[m] + 1
        else:
            return m
```

# Aufgabe 5

(a) Bestimme die Anzahl der Vergleiche, die der „naive“ Algorithmus für das String-Matching zum Auffinden des Musters `GGCA` im Textstring `GGGAAAGGCAT` benötigt.

(b) Leite die Worst-Case-Laufzeit für die Suche nach einem Muster $p$ der Länge $m$ in einem Text $t$ der Länge $n$ anhand eines geeigneten Beispiels her.

# Aufgabe 5

(a)

| G | G | G | A | A | A | G | G | C | A | T | Vergleiche |
|---|---|---|---|---|---|---|---|---|---|---|---|
| G | G | C | A | | | | | | | | 3 |
| | G | G | C | A | | | | | | | 3 |
| | | G | G | C | A | | | | | | 2 |
| | | | G | G | C | A | | | | | 1 |
| | | | | G | G | C | A | | | | 1 |
| | | | | | G | G | C | A | | | 1 |
| | | | | | | G | G | C | A | | 4 |
| Total | | | | | | | | | | | 15 |

(b) $t = \texttt{aaaaaaa}$, $p = \texttt{aab}$

```
a a a a a a a
a a b
  a a b
    a a b
      a a b
        a a b
          a a b
```

Das Worst-Case-Muster der Länge $m = 3$ wird
$n - m + 1 = 7 - 3 + 1 = 5$ mal erfolglos mit dem jeweiligen

# Aufgabe 6

Bestimme die Anzahl der Vergleiche, die der Boyer-Moore-Horspool-Algorithmus für das String-Matching zum Auffinden des Musters `GGCA` im Textstring `GGGAAAGGCAT` benötigt.

# Aufgabe 6

`pattern=GGCA` ($m = 4$)

Bad Character Table:

| Character | G | C | A | * |
|---|---|---|---|---|
| Shift | 2 | 1 | 4 | 4 |

Das Symbol * steht für alle Buchstaben, die nicht im Suchmuster vorkommen.

`Shift = Wert[pattern[j]] = m-j-1 (j=0, ..., m-2)`

| G | G | G | A | A | A | G | G | C | A | T | Vergleiche |
|---|---|---|---|---|---|---|---|---|---|---|---|
| G | G | C | A | | | | | | | | 2 |
| | | | | G | G | C | A | | | | 1 |
| | | | | | | G | G | C | A | | 4 |
| Total | | | | | | | | | | | 7 |

# Aufgabe 7

Erstelle schrittweise die Bad-Character-Table des Boyer-Moore-Horspool-Algorithmus für das Suchmuster `SALATTELLER`. Zeichen des Alphabets, die nicht im Suchmuster vorkommen, sind durch einen Stern (*) darzustellen.

# Aufgabe 7

Suchmuster: SALATTELLER (Länge: 11)

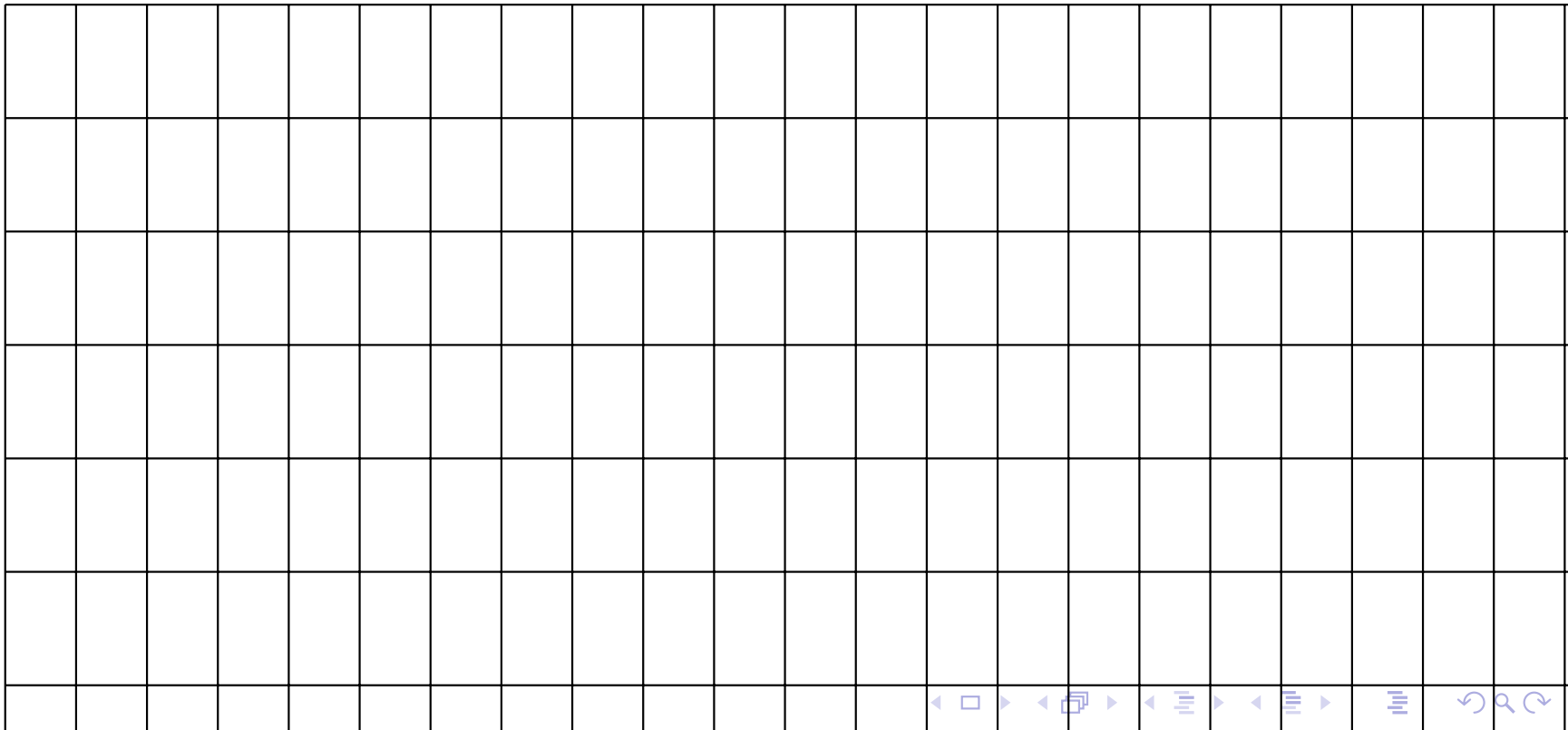
| S | A | L | T | E | R | * | |
|---|---|---|---|---|---|---|---|
| 11 | 11 | 11 | 11 | 11 | 11 | 11 | Initialisierung |
| 10 | 11 | 11 | 11 | 11 | 11 | 11 | $d(\mathrm{S},\mathrm{R}) = 11 - 1 = 10$ |
| 10 | 9 | 11 | 11 | 11 | 11 | 11 | $d(\mathrm{A},\mathrm{R}) = 11 - 2 = 9$ |
| 10 | 9 | 8 | 11 | 11 | 11 | 11 | $d(\mathrm{L},\mathrm{R}) = 11 - 3 = 8$ |
| 10 | 7 | 8 | 11 | 11 | 11 | 11 | $d(\mathrm{A},\mathrm{R}) = 11 - 4 = 7$ |
| 10 | 7 | 8 | 6 | 11 | 11 | 11 | $d(\mathrm{T},\mathrm{R}) = 11 - 5 = 6$ |
| 10 | 7 | 8 | 5 | 11 | 11 | 11 | $d(\mathrm{T},\mathrm{R}) = 11 - 6 = 5$ |
| 10 | 7 | 8 | 5 | 4 | 11 | 11 | $d(\mathrm{E},\mathrm{R}) = 11 - 7 = 4$ |
| 10 | 7 | 3 | 5 | 4 | 11 | 11 | $d(\mathrm{L},\mathrm{R}) = 11 - 8 = 3$ |
| 10 | 7 | 2 | 5 | 4 | 11 | 11 | $d(\mathrm{L},\mathrm{R}) = 11 - 9 = 2$ |
| 10 | 7 | 2 | 5 | 1 | 11 | 11 | $d(\mathrm{E},\mathrm{R}) = 11 - 10 = 1$ |

# Aufgabe 8

(a) Gib eine graphische Darstellung des determinstischen endlichen Automaten (DFA) an, mit dem in einem beliebigen Text nach dem Muster `ABAB` gesucht werden kann.

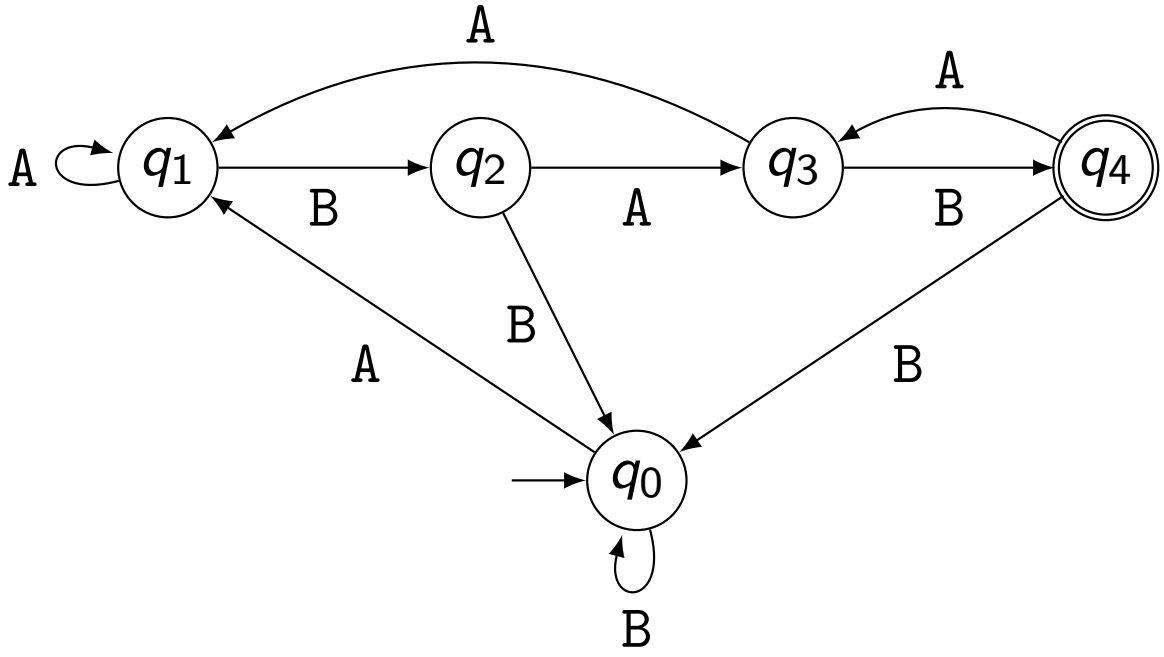
(b) Zeige mit einer Tabelle, wie die Zustandsänderungen des DFA algorithmus ermittelt werden können.

(c) Implementiere die Übergangstabelle des DFA als Python-Dictionary.



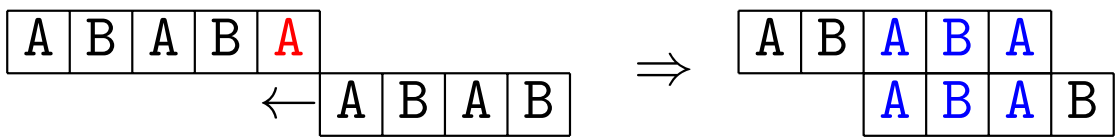
# Aufgabe 8

(a) Der DFA in graphischer Darstellung ($p = \texttt{ABAB}$):



(b) Erweitere alle Präfixe des Musters der Länge $i = 0, 1, \ldots, m$ (die Zustände $q_i$) um jeweils alle möglichen Zeichen des Alphabets $\Sigma$.
Schiebe das Muster von rechts her unter diese Präfixe bis es im überlappenden Bereich zur maximialen Üereinstimmung kommt.
Die Länge $\ell$ dieser Übereinstimmung, die auch 0 sein kann, ist der neue Zustand $q_\ell$.

Hat der DFA gerade das Muster $p$ erkannt (ist im Zustand $q_4$) und liest er das Zeichen `A`, so hat das maximale Suffix von $p + \texttt{A}$, das gleichzeitig auch Präfix von $p$ ist, die Länge 3, weshalb der neue Zustand $q_3$ ist.




| $q_i$ | p[0:i]+x | s | $q_{i+1}$ |
|---|---|---|---|
| 0 | A | A | 1 |
| 0 | B | $\varepsilon$ | 0 |
| 1 | AA | A | 1 |
| 1 | AB | AB | 2 |
| 2 | ABA | ABA | 3 |

(c) Python-Code der Übergangstabelle:

```
tab = {
                                                                    }
    0 : { 'A' : 1 , 'B' : 0 } ,
    1 : { 'A' : 1 , 'B' : 2 } ,
    2 : { 'A' : 3 , 'B' : 0 } ,
    3 : { 'A' : 1 , 'B' : 4 } ,
    4 : { 'A' : 3 , 'B' : 0 }
}
```