Karlsruher Institut für Technologie

Institut für Visualisierung und Datenanalyse
Lehrstuhl für Computergrafik
Prof. Dr.-Ing. Carsten Dachsbacher

# Hauptklausur Algorithmen I
# unverbindlicher Lösungsvorschlag
# SS 2020

26. September 2020

| | |
|---|---|
| Name: | |
| Matrikelnummer: | |
| Klausurcode: | |

**Beachten Sie:**

- Schreiben Sie Ihren **Klausurcode** (2 Buchstaben), vollständigen **Namen** und **Matrikelnummer** in Druckschrift in das Feld auf dem Deckblatt! Sie sollten Ihren Code gut aufheben, um später Ihre Note zu erfahren.
- Schreiben Sie Ihre Matrikelnummer oben auf jedes bearbeitete Aufgabenblatt.
- Schreiben Sie Ihre Lösungen auf die Aufgabenblätter. Bei Bedarf können Sie weiteres Papier anfordern, welches Sie am Ende der Klausur mit in den Umschlag stecken.
- Halten Sie sich bei Pseudocode-Aufgaben an die Notation der Vorlesung / Übung, um Missverständnissen vorzubeugen.
- Wir akzeptieren auch englische Antworten.
- Sie dürfen **ein handbeschriebenes Din A4-Blatt** mitbringen, sonst sind **keine Hilfsmittel** zugelassen.
- Sie haben **120 Minuten** Bearbeitungszeit.
- Die Klausur umfasst 30 Seiten (12 Blätter) mit 9 Aufgaben.
- Tragen Sie bitte außerhalb von Ihrem Platz einen Mund-Nasen-Schutz.

| Aufgabe | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | **Gesamt** |
|---|---|---|---|---|---|---|---|---|---|---|
| Erreichte Punkte | | | | | | | | | | |
| Erreichbare Punkte | 15 | 26 | 24 | 20 | 22 | 25 | 30 | 34 | 44 | **240** |

| **Note** |
|---|
| |

## Aufgabe 1: Laufzeit (15 Punkte)

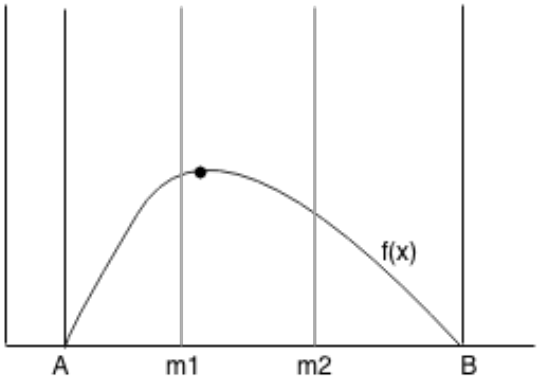
a) Zeigen Sie, dass $g(n) := n + n \log_2 n \in \mathcal{O}(n^2)$! **(5 Punkte)**

**Lösungsvorschlag**

Mögliche Varianten:

- Über die Definition von $\mathcal{O}$:
  $g(n) \leq n \cdot \log(n) + n \cdot \log(n) = 2n \cdot \log(n) \leq 2n^2$
- Limes-Betrachtung: $g(n) \in \mathcal{O}(n^2) \Leftrightarrow 0 \leq \limsup_{n\to\infty} \frac{g(n)}{n^2} < \infty$
  $\frac{n+n\log n}{n^2} = \frac{1+\log n}{n} \to 0$ für $n \to \infty \Rightarrow g(n) \in \mathcal{O}(n^2)$.

b) Die Funktion $f$ habe im Intervall $[A, B]$ nur ein lokales Maximum. Dieses Maximum lässt sich rekursiv bestimmen, indem man das Suchintervall um ein Drittel pro Schritt verkleinert.




- Ausgehend vom Intervall $[A, B]$ bestimmt man die Funktionswerte $f(m_1)$ und $f(m_2)$ an den inneren Grenzen $m_1 = (2A + B)/3$ und $m_2 = (A + 2B)/3$.
- Wenn $f(m_1) \leq f(m_2)$, betrachtet man im Anschluss das Intervall $[m_1, B]$, ansonsten das Intervall $[A, m_2]$, und fährt darin rekursiv mit der Suche fort.
- Wenn die Intervallgrenzen weniger als $\varepsilon$ auseinander liegen, wird dieses Intervall zurückgegeben.
- Der Algorithmus kann damit $n \approx (B - A)/\varepsilon$ verschiedene Intervalle ausgeben.

i) Stellen Sie eine Rekurrenz für die Laufzeit $T(n)$ auf! **(6 Punkte)**

**Lösungsvorschlag**

$$T(n) = \begin{cases} 1 & \text{falls } n \leq 1 \\ T(2/3n) + \mathcal{O}(1) & \text{sonst} \end{cases} \quad (1)$$

- Fall 1: falls $n \leq 3/2$ auch in Ordnung
- Statt $\mathcal{O}(1)$ "const", "konstant", oder tatsächliche Zahl der Operationen $>0$ auch in Ordnung.

ii) Welche Laufzeit erwarten Sie in $\mathcal{O}$-Notation und warum? **(4 Punkte)**

Name: LÖSUNGSVORSCHLAG Matrikelnummer: _____

**Lösungsvorschlag**

$T(n) \in \mathcal{O}(\log n)$ .
Für die Begründung gibt es mehrere Möglichkeiten, z.B. :

- Das Intervall wird pro Schritt um einen konstanten Faktor reduziert. Exponentielles schrumpfen $\Rightarrow$ logarithmische Laufzeit.
- Vollständiges Mastertheorem:
  - $a = 1, b = 3/2, f(n) = const$
  - $\log_b a = 0$
  - $f(n) \in \Theta(n^0 \log^0 n) = \Theta(1)$
  - Fall 2 des Mastertheorems
  - $\Rightarrow T(n) \in \Theta(n^{\log_b a} \log^{0+1} n) = \Theta(\log n)$
- Achtung: Das einfache Mastertheorem kann diese Rekurrenz nicht auflösen. Andere Begründung erforderlich.
- Nach $k$ Schritten ist das betrachtete Intervall nur noch $(2/3)^k \cdot (B - A)$ groß.
  $(2/3)^k * (B - A) < \varepsilon \Leftrightarrow k > \log_{2/3}(\varepsilon/(B - A))$

  Achtung: Der Logarithmus hat eine Basis $< 1$. Damit zu Rechnen kann anstrengend sein (z.B. hat sich deswegen in der Zeile darüber das Vorzeichen gedreht), deswegen machen wir einen Basiswechsel:
  $\log_{2/3} x = \ln x \cdot \log_{2/3} e$
  und substituieren $\epsilon/(B - A) = 1/n$. Damit erhalten wir:

  $\log_{2/3}(\epsilon/(B - A)) = \ln(1/n) \cdot \log_{2/3}(e)$
  $= (\ln 1 - \ln n) \cdot \log_{2/3}(e)$
  $= -\ln n \cdot \log_{2/3}(e)$
  Es gilt: $\log_{2/3}(e) =: c$ ist ein konstanter Wert $< 0$.
  Außerdem giltfür $n \to \infty$: $-\ln n < 0$.
  Daraus folgt:
  $(2/3)^k (B - A) < \varepsilon \Leftrightarrow k > |c| \ln n$
  d.h. nach $|c| \ln n \in \mathcal{O}(\log n)$ Schritten ist das Intervall $< \varepsilon$ groß und damit die Abbruchbedingung erfüllt.

## Aufgabe 2: Listen und Hashing (26 Punkte)

a) Im Folgenden soll ein Stapel (Stack) mittels eines unbeschränkten Feldes (`UArray`) implementiert werden.

i) Was ist die Worst Case-Laufzeit und die amortisierte Laufzeit für das Einfügen eines neuen Elements in einen solchen Stack in $\mathcal{O}$-Notation? **(2 Punkte)**

**Lösungsvorschlag**

Worst Case: $\mathcal{O}(n)$, amortisiert: $\mathcal{O}(1)$ .

ii) Nennen Sie einen Vorteil und einen Nachteil, einen Stack mit `UArrays` statt mit verketteten Listen zu implementieren! **(4 Punkte)**

**Lösungsvorschlag**

Vorteil:
- Cache Effizienz / keine Indirektion
- weniger Speicher bei kleinen Elementen (kleiner als ein Zeiger)

Nachteil:
- Worst Case $\mathcal{O}(n)$ / nicht-deterministische Laufzeit
- mehr Speicher bei großen Elementen

keine gültige Vor-/Nachteile:
- Implementierungsaufwand ist bei beiden ähnlich
- Speicherverbrauch allgemein (bei Elementen der Größe eines Zeigers brauchen beide gleich viel)
- „in der Praxis schneller“ ohne Begründung

b) Im Folgenden soll Hashing mit linearer Suche angewandt werden. Die Hashfunktion und Ausgangszustand der Hashtabelle sind gegeben als:

**Hashfunktion**

| Element | $a$ | $b$ | $d$ | $f$ | $m$ | $v$ | $s$ | $t$ |
|---|---|---|---|---|---|---|---|---|
| Hash-Wert | 0 | 1 | 3 | 5 | 3 | 3 | 0 | 1 |

**Hashtabelle**

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Wert | a | | | d | m | v | | | |

Führen Sie nun nacheinander die folgenden Operationen durch und geben Sie den Zustand der Hashtabelle nach Ausführen jeder Operationen an! **(4 Punkte)**

**Lösungsvorschlag**

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| `insert(s)` | a | **s** | | d | m | v | | | |
| `insert(f)` | a | s | | d | m | v | **f** | | |
| `remove(v)` | a | s | | d | m | **f** | | | |
| `insert(b)` | a | s | **b** | d | m | f | | | |

Name: LÖSUNGSVORSCHLAG Matrikelnummer: _____

c) Um die Ausbreitung des neuartigen Coronavirus in der Bevölkerung zu messen, wurde ein Datensatz $\mathcal{D}$ von positiv getesteten Personen erstellt. $\mathcal{D}$ enthält $n \in \mathbb{N}$ Tripel $(p, L, t) \in \mathcal{D}$ mit einer eindeutigen Personalausweisnummer $p \in \mathbb{N}$, einem Landkreis $L \in \{0, \ldots, 294\}$ und den vergangenen Tagen $t \in \mathbb{N}$ seit einem positiven Test.

Personen können mehrfach positiv getestet worden sein. Es kann daher mehrere Einträge $(p_i, L_i, t_i)$ und $(p_j, L_j, t_j)$ mit der gleichen Personalausweisnummer $(p_i = p_j)$ geben, welche aber von einem anderen Tag $(t_i \neq t_j)$ oder Landkreis $(L_i \neq L_j)$ stammen.

i) Geben Sie in der Funktion `findPersons` einen Algorithmus in Pseudocode an, der mit Speicherbedarf $\mathcal{O}(n)$ in erwarteter Laufzeit $\mathcal{O}(n)$ alle Personalausweisnummern der Personen ausgibt, die in den letzten 14 Tagen $(t \leq 14)$ positiv getestet wurden! Geben Sie weiterhin die Tage seit dem *letzten* positiven Test aus! Keine Personalausweisnummer soll mehr als einmal ausgegeben werden. Mit der Funktion `print` $(p, t)$ können Sie eine Personalausweisnummer $p$ und die vergangenen Tage $t$ ausgeben. **(12 Punkte)**

ii) Begründen Sie das erwartete Laufzeitverhalten Ihres Algorithmus aus i)! **(4 Punkte)**

**Function** findPersons(D : [(int, int, int); $n$])
{

Lösungsvorschlag

$n \leftarrow |\mathcal{D}|$ // $\mathcal{O}(n)$
Initialisiere Hashtabelle $H : N \to N \times N$ mit $k \cdot n$ Slots mit $k \geq 1$ // $\mathcal{O}(n)$
*oder* Initialisiere Hashtabelle $H : N \to N \times N$ mit $\Theta(n)$ Slots *und verketteten Listen zur Kollisionsauflösung* // $\mathcal{O}(n)$
**for** $(p, L, t) \in \mathcal{D}$ **do**
  **if** $t \leq 14$ **then**
    // $p \in H$, H.insert und Zugriff $H[p]$ ist *erwartet* $\mathcal{O}(1)$
    **if** $p \notin H$ **then**
      H.insert $(p, t)$
    **else**
      H[p] $\leftarrow \min(t, H[p])$
    **end**
  **end**
**end**
// $\mathcal{O}(n)$
**for** $(p, t) \in H$ **do**
  `print` $(p, t)$
**end**

## Aufgabe 3: Sortieren und Selektieren (24 Punkte)

a) Nennen Sie je einen Vorteil von *Quicksort* und *Mergesort*! **(2 Punkte)**

Lösungsvorschlag

| Vorteil *Quicksort* | Vorteil *Mergesort* |
|---|---|
| • In-Place / weniger Speicher<br>• Best Case ist besser<br>• Speicher in erwartet in $\mathcal{O}(\log(n))$<br>Nicht gültig:<br>• Hohe Cache-Effizienz<br>• in der Praxis schneller | • Stabil<br>• Worst Case Laufzeit von $\mathcal{O}(n \log n)$<br>• balancierte Rekursion (Parallelisierung) |

b) Gegeben sei die Sequenz $\langle 28, 24, 21, 70, 56, 82, 23, 61, 47, 18\rangle$. Bestimmen Sie das Element mit Rang $k = 7$ mit Hilfe des *Quickselect*-Algorithmus aus der Vorlesung! Geben Sie jeweils für jeden Rekursionsschritt die aktuell betrachtete Teilsequenz $S$, den aktuellen Wert für $k$ sowie die Sequenzen $a := \langle e \in S : e < p\rangle$, $b := \langle e \in S : e = p\rangle$ und $c := \langle e \in S : e > p\rangle$ an! Die Elemente von $a$, $b$ und $c$ müssen dabei die gleiche Reihenfolge wie in $S$ haben. Wählen Sie als Pivot immer das erste Element von $S$! **(10 Punkte)**

Name: LÖSUNGSVORSCHLAG Matrikelnummer: _____

**Lösungsvorschlag**

| $s$ | $k$ | Sequenzen |
|---|---|---|
| $\langle 28, 24, 21, 70, 56, 82, 23, 61, 47, 18\rangle$ | 7 | $a$:$\langle 24, 21, 23, 18\rangle$<br>$b$:$\langle 28\rangle$<br>$c$:$\langle 70, 56, 82, 61, 47\rangle$ |
| $\langle 70, 56, 82, 61, 47\rangle$ | 2 | $a$:$\langle 56, 61, 47\rangle$<br>$b$:$\langle 70\rangle$<br>$c$:$\langle 82\rangle$ |
| $\langle 56, 61, 47\rangle$ | 2 | $a$:$\langle 47\rangle$<br>$b$:$\langle 56\rangle$<br>$c$:$\langle 61\rangle$ |

Das Element mit Rang $k = 7$ ist 56.
Falls man davon ausgeht, dass der Rang bei 0 beginnt, ist das Ergebnis 61; an der Tabelle ändert sich nichts.

c) Sequenzen der Form $x_0 \leq \cdots \leq x_k \geq \cdots \geq x_{n-1}$ mit $0 \leq k < n$ werden *bitonisch* genannt (z. B. $\langle 1, 2, 5, 10, 9, 8, 4 \rangle$). Entwerfen Sie in Pseudocode einen Algorithmus `sortBitonicSeq`, der die Elemente einer bitonischen Sequenz in aufsteigend sortierter Reihenfolge mit der Funktion `print(i: int)` ausgibt! Die Laufzeit muss dabei in $\mathcal{O}(n)$ liegen. Die Eingabe liegt als Array $a$ vor und darf nicht verändert werden. Für Lösungen mit mehr als $\mathcal{O}(1)$ Speicherbedarf werden nur Teilpunkte vergeben. Begründen Sie außerdem die Laufzeit Ihres Algorithmus! **(12 Punkte)**

**Function** sortBitonicSeq(a : [int; $n$])
{

Lösungsvorschlag

```
start ← 0
end ← n-1
while start ≤ end do
    if a[start] ≤ a[end] then
        print(a[start])
        start ← start+1
    end
    else
        print(a[end])
        end ← end-1
    end
end
```

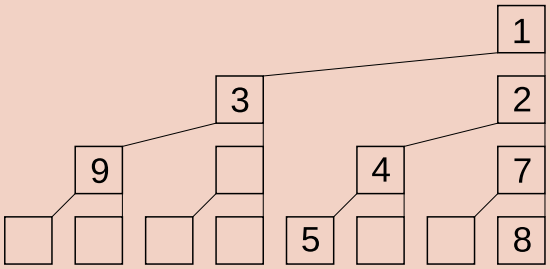
Name: LÖSUNGSVORSCHLAG Matrikelnummer: _____

**Aufgabe 4: Heaps (20 Punkte)**

*Kartesische Bäume* zu einer Sequenz $S = \langle s_1, s_2, ... \rangle$ sind besondere binäre Min-Heaps, bei denen neben der Heap-Eigenschaft eine Ordnung für die Geschwisterknoten gilt: Jeder innere Knoten $s_j$ enthält in seinem linken Kindteilbaum nur Elemente, die in der Sequenz $s_j$ vorangehen, und in seinem rechten Kindteilbaum nur Elemente, die in der Sequenz $s_j$ nachfolgen. Dadurch kann durch eine geordnete Traversierung des Binärbaums die ursprüngliche Sequenz wieder hergestellt werden.

a) Geben Sie den kartesischen Baum zur Sequenz $S = \langle 9, 3, 1, 5, 4, 2, 7, 8 \rangle$ an! **(8 Punkte)**

Lösungsvorschlag



b) Füllen Sie die Methode `insert` in Pseudocode aus, sodass die Funktion `makeCartesianTree` einen kartesischen Baum aus der Sequenz $S$ konstruiert! **(12 Punkte)**

```
Struct Node
{
    s : int
    l : Node
    r : Node
}
```

**Function** makeCartesianTree(S : [int; N]) : Node

```
root ← ⊥
for s ∈ S do
 |  root ← insert(root, Node(s[i], ⊥, ⊥))
end
return root
```

```
// Füge new.s in den Heap ein, gebe neuen Wurzelknoten zurück
Function insert(root: Node, new: Node) : Node
{
```

Lösungsvorschlag

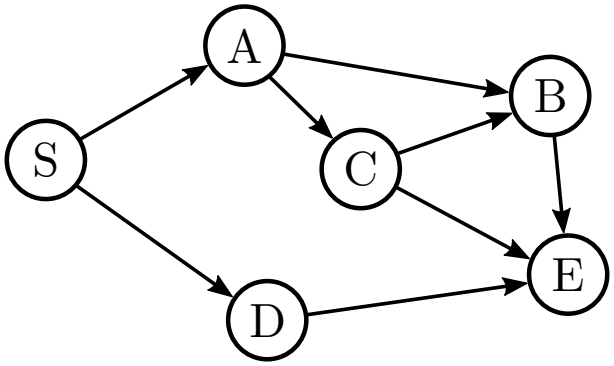
```
    if root = ⊥ or new.s < root.s then
    |   new.l ← root
    |   return new
    end
    else
    |   root.r ← insert(root.r, new)
    |   return root
    end
```

```
}
```

Name: LÖSUNGSVORSCHLAG Matrikelnummer: _____

**Aufgabe 5: Graphen (22 Punkte)**

Gegeben sei der folgende gerichtete Graph:




a) Geben Sie die Adjazenzfeldrepräsentation und Adjazenzmatrix des Graphen an! Beachten Sie die Sortierung $S, A, B, \ldots, E$ der Knoten!

i) Adjazenzfeld: **(4 Punkte)**

V: | | | | | | | | | | | | | | |

E: | | | | | | | | | | | | | | |

Weiterer Platz, falls Sie Ihre Antwort korrigieren wollen. Falls Sie diesen nutzen, kennzeichnen Sie eindeutig, welche Lösung gewertet werden soll:

V: | | | | | | | | | | | | | | |

E: | | | | | | | | | | | | | | |

ii) Adjazenzmatrix: **(4 Punkte)**

$$\begin{pmatrix} & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \end{pmatrix}$$

**Lösungsvorschlag**

i) V:

| 0 | 2 | 4 | 5 | 7 | 8 |
|---|---|---|---|---|---|

E:

| 1 | 4 | 2 | 3 | 5 | 2 | 5 | 5 |
|---|---|---|---|---|---|---|---|
| A | D | B | C | E | B | E | E |

Kein Abzug für 1-Indizierung (betrifft nur Werte in $V$).
$A, D$ sowie $B, C$ und $B, E$ in Einträgen ohne Trennlinien dürfen jeweils auch vertauscht auftreten.

ii)

$$\begin{pmatrix} 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

b) Vergleichen Sie die Adjazenzmatrix und Adjazenfeldrepräsentation für einen Graphen $G = (V, E)$, indem Sie in $\mathcal{O}$-Notation für jede Repräsentation möglichst enge Schranken für den Speicherbedarf sowie die Worst Case-Laufzeit für das Einfügen einer neuen Kante angeben! **(4 Punkte)**

| | Speicherbedarf | Einfügen |
|---|---|---|
| Adjazenzmatrix | | |
| Adjazenzfeld | | |

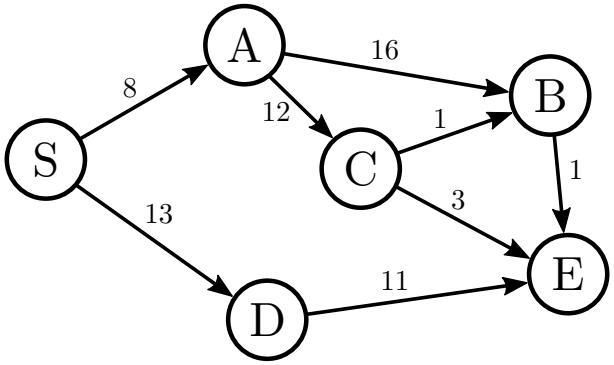
**Lösungsvorschlag**

| | Speicherbedarf | Einfügen |
|---|---|---|
| Adjazenzmatrix | $\mathcal{O}(\|V\|^2)$ | $\mathcal{O}(1)$ |
| Adjazenzfeld | $\mathcal{O}(\|V\| + \|E\|)$ | $\mathcal{O}(\|V\| + \|E\|)$ |

$\mathcal{O}(|V| + |E|) = \mathcal{O}(\max(|V|, |E|))$

c) Führen Sie auf dem folgenden Graphen Dijkstras Algorithmus aus, beginnend mit Knoten $S$! Füllen Sie dazu untenstehende Tabelle aus! Geben Sie pro Schritt die bisher gefundenen Entfernungen der Knoten an und markieren Sie den aktuellen Knoten, wie beispielhaft für den ersten Schritt angegeben! Sie müssen nur Werte angeben, die sich ändern. **(10 Punkte)**

Name: LÖSUNGSVORSCHLAG Matrikelnummer: _____




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| S | ⓪ | | | | | | | | | |
| A | ∞ | 8 | | | | | | | | |
| B | ∞ | | | | | | | | | |
| C | ∞ | | | | | | | | | |
| D | ∞ | 13 | | | | | | | | |
| E | ∞ | | | | | | | | | |

**Lösungsvorschlag**

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| S | ⓪ | | | | | | | | | |
| A | ∞ | ⑧ | | | | | | | | |
| B | ∞ | | 24 | | ㉑ | | | | | |
| C | ∞ | | 20 | ⑳ | | | | | | |
| D | ∞ | 13 | ⑬ | | | | | | | |
| E | ∞ | | | 24 | 23 | ㉒ | | | | |

## Aufgabe 6: Minimale Spannbäume (MST) (25 Punkte)

a) Erklären Sie die folgenden zwei Eigenschaften minimaler Spannbäume (MST)! **(4 Punkte)**

i) Schnitt-Eigenschaft:

**Lösungsvorschlag**

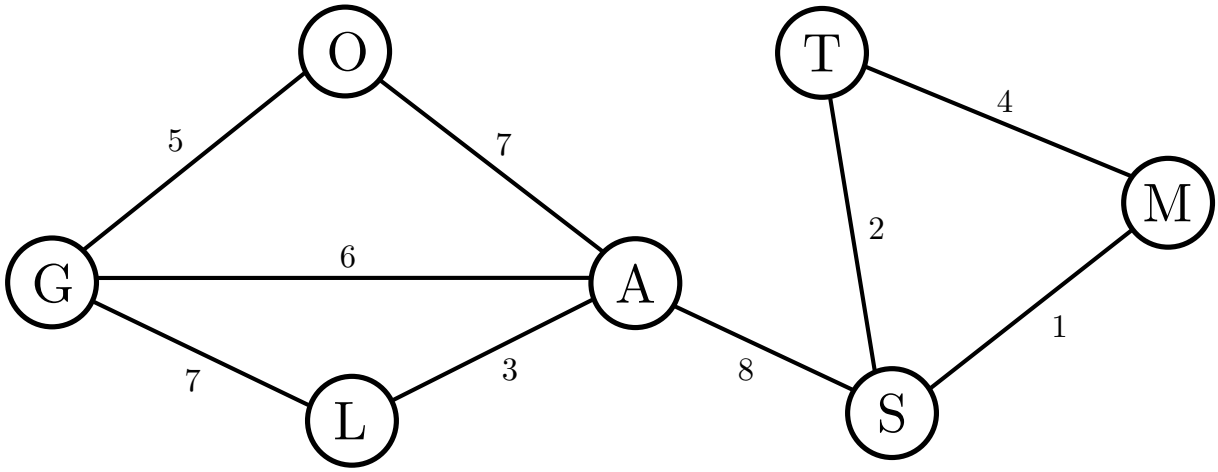
Die leichteste Kante in einem Schnitt kann in einem MST verwendet werden.

ii) Kreis-Eigenschaft:

**Lösungsvorschlag**

Die schwerste Kante eines Kreises wird nicht für einen MST benötigt.

b) Bestimmen Sie den minimalen Spannbaum des folgenden Graphen mit Hilfe des Kruskal-Algorithmus! Geben Sie in der Tabelle pro Schritt die betrachtete Kante an sowie ob diese zum MST hinzugefügt wurde! **(9 Punkte)**




| Schritt | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Betrachtete Kante | { , } | { , } | { , } | { , } | { , } |
| Im MST (✓/X) | | | | | |

| Schritt | 6 | 7 | 8 | 9 |
|---|---|---|---|---|
| Betrachtete Kante | { , } | { , } | { , } | { , } |
| Im MST (✓/X) | | | | |

Name: LÖSUNGSVORSCHLAG Matrikelnummer: _____

**Lösungsvorschlag**

| Schritt | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Betrachtete Kante | {M,S} | {S,T} | {A,L} | {M,T} | {G,O} |
| Im MST (✓/X) | ✓ | ✓ | ✓ | X | ✓ |

| Schritt | 6 | 7 | 8 | 9 |
|---|---|---|---|---|
| Betrachtete Kante | {A,G} | {A,O} | {G,L} | {A,S} |
| Im MST (✓/X) | ✓ | X | X | ✓ |

{A,O} und {G, L} dürfen vertauscht sein, ansonsten ist die Lösung eindeutig.

c) Welche Datenstrukturen verwendet man für eine effiziente Implementierung des Kruskal-Algorithmus? Wofür werden diese Datenstrukturen verwendet? **(4 Punkte)**

i)

**Lösungsvorschlag**

Mithilfe einer *sortierten Liste der Kanten* kann man die *Kante mit minimalem Gewicht auswählen*.

ii)

**Lösungsvorschlag**

Mithilfe der *Union-Find*-Datenstruktur kann man *herausfinden, ob eine Kante zwei Teilbäume verbindet*.

d) Gegeben sei der Algorithmus von Jarník und Prim zur Bestimmung eines minimalen Spannbaums eines gewichteten, gerichteten Graphen $G = (V, E)$ mit Knoten $V : [int; n]$ und Kanten $E : [(int, int); m]$:

**Function** MSTJarnikPrim(V : [int; n], E : [(int, int); m]) : [int; n-1]

```
d : [float; n] ← [∞]
s : int ← V[0]
parent : [int; n]

parent[s] ← s
d[s] ← 0

Q ← ...      // siehe i)
Q.insert(s)
while Q not empty do
    u ← Q.deleteMin()
    d[i] ← 0
    for each e ← (v,v') ∈ E where v=u do
        if weight(e) < d[v'] then
            d[v'] ← weight(e)
            parent[v'] ← u
            if v ∈ Q then Q.decreaseKey(v')
            else Q.insert(v')
return [(parent[v], v) : v ∈ V \ {s}]
```

i) Geben Sie eine für Q geeignete Datenstruktur an, welche die im Algorithmus benötigten Operationen effizient unterstützt! **(2 Punkte)**

**Lösungsvorschlag**

adressierbare Priority Queue / Prioritätsliste / binären Heaps / Fibonacci Heaps
ab-Baum

ii) Welche asymptotischen Laufzeiten haben die im Algorithmus auf Q ausgeführten Operationen für die Datenstruktur aus i) pro Aufruf? Wie oft werden die Operationen asymptotisch aufgerufen? Geben Sie möglichst enge Schranken in $\mathcal{O}$-Notation an! **(6 Punkte)**

Name: LÖSUNGSVORSCHLAG Matrikelnummer: _____

| **Operation** | **Laufzeit** | **Aufrufe** |
|---|---|---|
| `insert` | | |
| `decreaseKey` | | |
| `deleteMin` | | |

**Lösungsvorschlag**

Für binäre Heaps/ab-Baum:

| **Operation** | **Laufzeit** | **Aufrufe** |
|---|---|---|
| `insert` | $\mathcal{O}(\log n)$ | $\mathcal{O}(n)$ |
| `decreaseKey` | $\mathcal{O}(\log n)$ | $\mathcal{O}(m)$ |
| `deleteMin` | $\mathcal{O}(\log n)$ | $\mathcal{O}(n)$ |

Fibonacci Heaps:

- insert: Worst Case $\mathcal{O}(1)$
- decreaseKey: amortisiert $\mathcal{O}(1)$, Worst Case $\mathcal{O}(n)$
- deleteMin: amortisiert $\mathcal{O}(\log n)$, Worst Case $\mathcal{O}(n)$

## Aufgabe 7: 2-SAT-Problem (30 Punkte)

Das 2-SAT-Problem ist ein Erfüllbarkeitsproblem für aussagenlogische Formeln, die Konjunktionen ($\wedge$) von *maximal zweistelligen* Disjunktionen ($\vee$) sind. Im Gegensatz zum allgemeinen SAT-Problem sind für das 2-SAT-Problem Polynomialzeitalgorithmen bekannt.

Diese machen sich bekannte Graphenalgorithmen zu Nutze, um zu berechnen, ob für eine gegebene aussagenlogische Formel $F$ eine Belegung der Variablen mit Werten aus $\{0, 1\}$ so existiert, dass $F$ erfüllt ist. Im Folgenden bezeichnen wir die Menge der Variablen einer Formel $F$ mit $\mathbb{X}_F$.

**Kleiner aussagenlogischer Evaluationsleitfaden:** Eine Konjunktion ($\wedge$, *"und"*) ist genau dann *erfüllt*, wenn *alle* ihre Argumente den Wert 1 annehmen. Eine Disjunktion ($\vee$, *"oder"*) ist genau dann *nicht erfüllt*, wenn *alle* ihre Argumente den Wert 0 annehmen. Die Negation $\overline{v}$ einer Variablen $v$ ist genau dann erfüllt, wenn $v$ den Wert 0 annimmt. Die doppelte Negation $\overline{\overline{v}}$ vereinfachen wir zu $v$. Eine Implikation ($\rightarrow$, *"impliziert"*) ist genau dann *nicht* erfüllt, wenn die linke Seite den Wert 1 und die rechte Seite den Wert 0 annimmt.

a) Geben Sie für die 2-SAT-Formel $F_a := (a \vee b) \wedge (\overline{a} \vee \overline{b}) \wedge c$ alle erfüllenden Belegungen der Variablen $\mathbb{X}_F = \{a, b, c\}$ an! **(4 Punkte)**

Lösungsvorschlag

$$a = 1, b = 0, c = 1 \text{ und } a = 0, b = 1, c = 1$$

b) Der Implikationsgraph $G_F = (V, E)$ einer 2-SAT-Formel $F$ wird wie folgt erzeugt: Für jede Variable $v \in \mathbb{X}_F$ gibt es zwei Knoten $v$ und $\overline{v}$. Analog zu den Äquivalenzen $(a \vee b) \equiv (\overline{a} \rightarrow b) \equiv (\overline{b} \rightarrow a)$, werden für jede Disjunktion $(a \vee b)$ aus $F$ zwei Kanten $(\overline{a}, b)$ und $(\overline{b}, a)$ eingeführt. Konkret:

$$V := \mathbb{X}_F \cup \{\overline{x} \mid x \in \mathbb{X}_F\}$$
$$E := \{(\overline{a}, b), (\overline{b}, a) \mid (a \vee b) \in F\}$$

Zeichnen Sie den Implikationsgraphen der Formel $F_a$ aus Aufgabenteil a)! Beachten Sie dabei, dass einstellige Formelglieder durch zweistellige Disjunktionen ersetzt werden können, also z.B. $c \equiv c \vee c$ ! **(6 Punkte)**

**Lösungsvorschlag**

$a \leftrightarrow \overline{b}$
$b \leftrightarrow \overline{a}$
$\overline{c} \rightarrow c$
*Anmerkung:* Es gibt in dem Graphen eine Kante von $\overline{c}$ nach $c$, aber *keine* Kante von $c$ nach $\overline{c}$. Das liegt daran, dass $c$ durch die Disjunktion $c \vee c$ ersetzt werden kann. Damit ist $c \vee c \in F$ (rechter Teil der Definition von $E$). Durch das erste Muster $(\overline{a}, b)$ (linker Teil der Definition von $E$) enthält $E$ eine Kante, die beim negierten ersten Element $\overline{c}$ der Disjunktion $c \vee c$ startet und beim zweiten Element $c$ aufhört. Das ist $(\overline{c}, c)$. Durch das zweite Muster $(\overline{b}, a)$ enthält $E$ eine Kante, die beim negierten zweiten Element $\overline{c}$ der Disjunktion $c \vee c$ startet und beim ersten Element $c$ aufhört. Das ist ebenfalls $(\overline{c}, c)$. Das heißt, insgesamt erzeugt die Disjunktion $c \vee c$ *nur die eine Kante* $(\overline{c}, c)$ in $E$.

c) Schiefsymmetrie

Sei $G_F$ Implikationsgraph einer beliebigen 2-SAT-Formel $F$. Zeigen Sie: Gibt es in einen Pfad $P := (k_0, \ldots, k_n)$ in $G_F$, dann gibt es auch einen Pfad $(\overline{k}_n, \ldots, \overline{k}_0)$ in $G_F$. **(8 Punkte)**

**Lösungsvorschlag**

Für jede Kante $(k_i, k_j) \in E$ gilt nach Definition von $E$: $(\overline{k}_i \vee k_j) \in F$. Weiterhin ist dann nach Definition auch die Kante $(\overline{k}_j, \overline{k}_i) \in E$. Es gibt einen Pfad $(k_0, \ldots, k_n)$ in $G_F$, also auch Kanten $(k_0, k_1)$, $\ldots$, $(k_{n-1}, k_n)$ in $E$. Dann enthält $E$ auch die Kanten $(\overline{k}_1, \overline{k}_0)$,... $(\overline{k}_n \vee \overline{k}_{n-1})$, und damit $G_F$ den Pfad $(\overline{k}_n, ..., \overline{k}_0)$.

d) Transitivität

Sei $G_F$ Implikationsgraph einer beliebigen 2-SAT-Formel $F$. Zeigen Sie: Wenn es einen Pfad $(k_0, \ldots, k_n)$ in $G_F$ gibt, dann sind alle erfüllenden Belegungen von $F$ auch erfüllende Belegungen der Formel $K := k_0 \rightarrow k_n$, d.h. $K$ ist eine logische Konsequenz von $F$. **(12 Punkte)**

**Lösungsvorschlag**

Beweis durch Widerspruch: Sei $M$ erfüllende Belegung von $F$, die $k_0 \rightarrow k_n$ nicht erfüllt, d.h. unter $M$ gilt $k_0 = 1$ und $k_n = 0$. Es gibt aber Kanten $(k_0, k_1)$, $\ldots$, $(k_{n-1}, k_n)$ im Graphen, daraus folgt, es gibt Disjunktionen $(\overline{k}_0 \vee k_1)$, $\ldots$, $(\overline{k}_{n-1} \vee k_n)$ in der Formel. Aus $k_0 = 1$ folgt wegen $(\overline{k}_0 \vee k_1)$, dass in $M$ $k_1 = 1$ u.s.w., dann folgt aber auch aus $(\overline{k}_{n-1} \vee k_n)$, dass $k_n = 1$, im Widerspruch zur Annahme. Also gilt die Behauptung.

Lösungsvorschlag

Vorwärts-Lösung:
Annahme: Sei $(k_0, \ldots, k_n)$ ein Pfad in $G_F$.
$\Rightarrow \forall i = 0 \ldots n-1$ ist $(k_i, k_{i+1})$ eine Kante in $G_F$.
$\Rightarrow \forall i = 0 \ldots n-1$ ist $k_i \rightarrow k_{i+1}$ (alternativ $\overline{k}_i \vee k_{i+1}$) Teil der Formel $F$, die $G_F$ erzeugt.
$\Rightarrow$ alle erfüllende Belegungen von $F$ erfüllen insbesondere $k_i \rightarrow k_{i+1}$ (alternativ $\overline{k}_i \vee k_{i+1}$) für alle $i = 0 \ldots n-1$.
(Es gilt: $\overline{k}_i \vee k_{i+1}$ ist äquivalent zu $k_i \rightarrow k_{i+1}$)
Aus der Transitivit der Implikation folgt: für alle erfüllenden Belegungen von $F$ ist $k_0 \rightarrow k_n$ erfüllt.

Name: LÖSUNGSVORSCHLAG Matrikelnummer: _____

**Aufgabe 8: Sortierte Listen (34 Punkte)**

a) Welche Invariante ermöglicht die effiziente Suche in einem binären Suchbaum? **(4 Punkte)**

**Lösungsvorschlag**

Für alle inneren Knoten mit Schlüssel $k$, linkem Teilbaum $l$ und rechtem Teilbaum $r$ gilt: alle über $l$ erreichbaren Blätter sind $\leq k$, alle über $r$ erreichbaren Blätter sind $> k$.

b) Sei $h$ die Höhe eines $(a, b)$-Baums mit $n \geq 2$ Elementen und $2 \leq a < b$. Zeigen Sie: $h \in \Theta(\log n)$! Betrachten Sie dabei $a$ und $b$ als Konstanten! **(10 Punkte)**

**Lösungsvorschlag**

**1.** $h \in \mathcal{O}(\log n)$
Jeder innere Knoten hat mindestens $a$ Kindknoten (bzw. die Wurzel mindestens 2): $2 \cdot a^{h-1} \leq n + 1 \Rightarrow h \leq 1 + \log_a \frac{n+1}{2} \Rightarrow h \in \mathcal{O}(\log n)$
**2.** $h \in \Omega(\log n)$
Jeder innere Knoten hat höchstens $b$ Kindknoten: $b^h \geq n+1 \Rightarrow h \geq \log_b n + 1 \Rightarrow h \in \Omega(\log n)$
aus 1. und 2. folgt $h \in \Theta(\log n)$

c) Nennen Sie *je eine* Operation aus der Vorlesung, die für ein sortiertes Array oder einen $(a, b)$-Baum (ohne Augmentierungen) eine bessere asymptotische Laufzeit hat! **(4 Punkte)**

effizient für sortiertes Array:

effizient für $(a, b)$-Baum:

**Lösungsvorschlag**

besser für Array: select, rangeSize
besser für $(a, b)$-Baum: insert, remove

d) Nennen Sie *je einen* Vorteil von $(a, b)$-Bäumen und Hashtabellen gegenüber der jeweils anderen Datenstruktur! Gehen Sie davon aus, dass nur von beiden unterstützte Operationen benötigt werden und nehmen Sie eine gut gewählte Hashfunktion an! **(4 Punkte)**

Vorteil $(a, b)$-Baum:

**Lösungsvorschlag**

$(a, b)$-Bäume: deterministisch, Worst Case besser

Vorteil Hashtabelle:

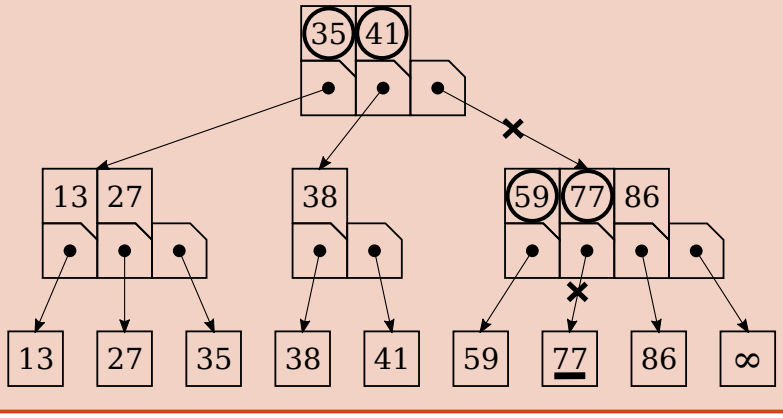
**Lösungsvorschlag**

Hashtabellen: erwartete Laufzeit besser, einfachere Implementierung durch weniger Sonderfälle

e) Führen Sie im folgenden $(2, 5)$-Baum die Operation *locate* mit den jeweils gegebenen Schlüsseln aus! Umkreisen Sie dazu alle betrachteten Spalt-Schlüssel, kreuzen Sie die verwendeten Kanten im Baum an und unterstreichen Sie das Ergebnis! **(4 Punkte)**

`locate(77):`

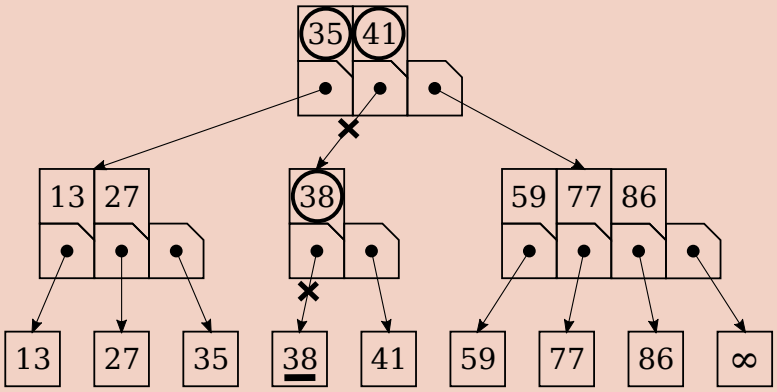
**Lösungsvorschlag**




`locate(37):`

Name: LÖSUNGSVORSCHLAG Matrikelnummer: _____

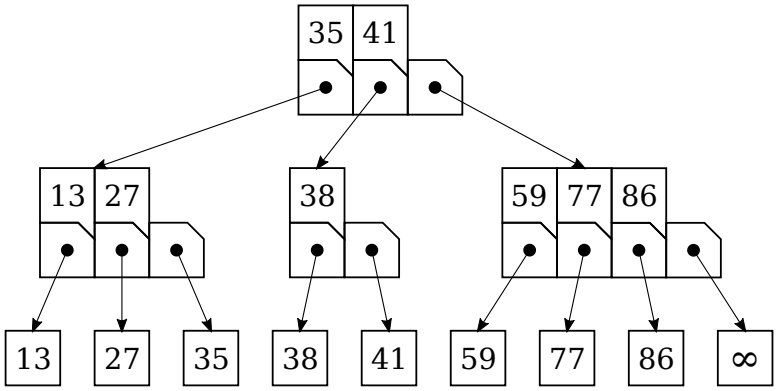
**Lösungsvorschlag**




**Lösungsvorschlag**

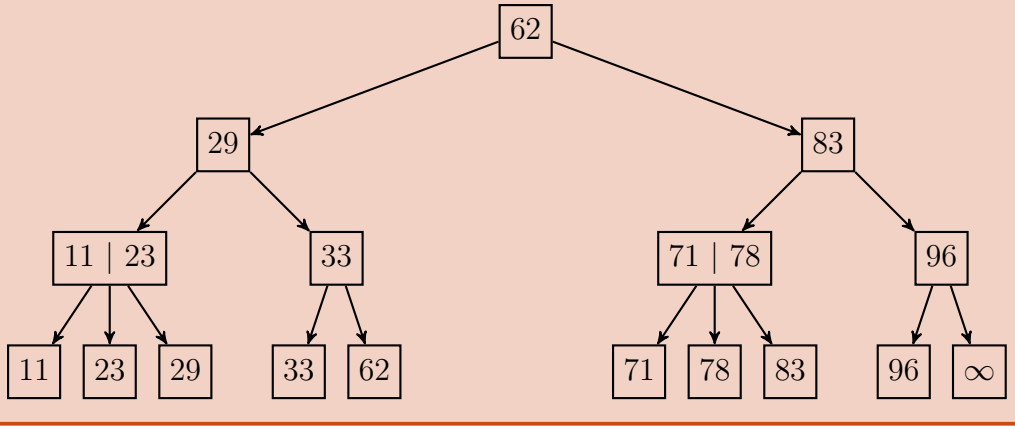
Weiterer Platz, falls Sie Ihre Antwort korrigieren wollen. Falls Sie diesen nutzen, kennzeichnen Sie eindeutig, welche Lösung gewertet werden soll:

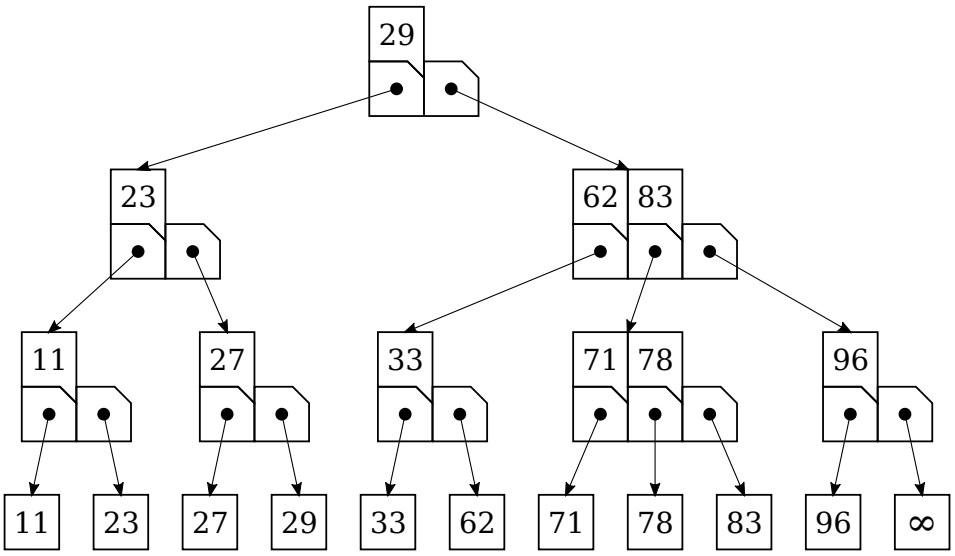
`locate( ):`




f) Führen Sie auf dem folgenden $(2, 5)$-Baum die Operation `remove(27)` aus und geben Sie den resultierenden Baum an! **(8 Punkte)**

**Lösungsvorschlag**




29
23
62
83
11
27
33
71
78
96
11
23
27
29
33
62
71
78
83
96
∞

Name: LÖSUNGSVORSCHLAG Matrikelnummer: _____

**Aufgabe 9: Edit-Distance (44 Punkte)**

Die Levenshtein-Distanz $L_{a,b}$ der beiden Strings $a$ und $b$ mit Länge $|a|$ und $|b|$ kann verwendet werden, um die Ähnlichkeit von $a$ und $b$ zu bestimmen. Sie quantifiziert, wie viele Operationen (z.B. Einfügen von Zeichen) nötig sind, um $a$ in $b$ zu überführen und ist wie folgt rekursiv definiert:

$$L_{a,b} = l_{a,b}(|a|, |b|)$$

$$l_{a,b}(i,j) = \begin{cases} max(i,j) & \text{wenn } min(i,j) = 0, \\ min \begin{cases} l_{a,b}(i-1,j)+1 \\ l_{a,b}(i,j-1)+1 \\ l_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & sonst \end{cases}$$

wobei

$$1_{(x \neq y)} = \begin{cases} 0 & \text{wenn } x = y, \\ 1 & sonst. \end{cases}$$

a) Die Levenshtein-Distanz kann effizient mit Dynamischer Programmierung (DP) berechnet werden. Vervollständigen Sie hierzu die vorgegebene DP-Matrix für $a =$ "passiv" und $b =$ "aktiv"! **(6 Punkte)**

| a \ b | | a | k | t | i | v |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| p | 1 | | | | | |
| a | 2 | | | | | |
| s | 3 | 2 | 2 | 3 | 4 | 5 |
| s | 4 | 3 | 3 | 3 | 4 | 5 |
| i | 5 | | | | | |
| v | 6 | | | | | |

**Lösungsvorschlag**

| a \ b | | a | k | t | i | v |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| p | 1 | 1 | 2 | 3 | 4 | 5 |
| a | 2 | 1 | 2 | 3 | 4 | 5 |
| s | 3 | 2 | 2 | 3 | 4 | 5 |
| s | 4 | 3 | 3 | 3 | 4 | 5 |
| i | 5 | 4 | 4 | 4 | 3 | 4 |
| v | 6 | 5 | 5 | 5 | 4 | 3 |

b) Die Levenshtein-Distanz definiert drei Operationen, um $a$ in $b$ zu überführen.

1. $R(i)$ Löschen von Buchstabe $a_i$
2. $I(i,j)$ Einfügen von Buchstabe $b_j$ an der Position $i$ in $a$
3. $X(i,j)$ Ersetzen von Buchstabe $a_i$ durch $b_j$

Nun soll "passiv" mit einer minimalen Anzahl Operationen in "aktiv" umgewandelt werden. Bestimmen Sie mithilfe der ausgefüllten Tabelle aus Teilaufgabe a):

☐ i) Wie kann die Anzahl der mindestens benötigten Operationen mithilfe der ausgefüllten DP-Matrix effizient (in konstanter Zeit) ermittelt werden? **(3 Punkte)**

**Lösungsvorschlag**

Ablesen des Tabelleneintrags an der Stelle $(|a|, |b|)$ (alternativ: Ablesen des Tabelleneintrags in der unteren rechten Zelle)

☐ ii) Wie ermittelt man effizient die benötigten Operationen mithilfe der ausgefüllten DP-Matrix? Geben Sie eine Folge von Operationen gemäß der DP-Matrix in der Kurznotation $(R(i), I(i,j), X(i,j))$ an, sodass "passiv" in "aktiv" umgewandelt wird! **(6 Punkte)**

Name: LÖSUNGSVORSCHLAG Matrikelnummer: _____

**Lösungsvorschlag**

- In Matrix von rechts unten starten
- Betrachte drei Nachbarzellen: oben, links, oben links
  - 1. Möglichkeit: Gehe zu einem Minimum, falls möglich diagonal
  - 2. Möglichkeit: Gehe zu einer Zelle, die Wert in aktueller Zelle erzeugt hat
- Operation ergibt sich daraus, welcher Nachbar gewählt wurde
- verfolge bis zur
  - 1. Möglichkeit: Zelle ganz oben links
  - 2. Möglichkeit: ersten 0

mögliche Operationen:
absolute Indizes:
1: R(0)
2: X(2,1)
3: X(3,2)
relative Indizes:
1: R(0)
2: X(1,1)
3: X(2,2)
absolute/relative Indizes rückwärts:
1: X(3,2)
2: X(2,1)
3: R(0)

iii) Ist die Folge an Operationen, welche den einen String in den Anderen umwandelt, eindeutig? Begründen Sie Ihre Antwort!**(3 Punkte)**

Lösungsvorschlag

Alternativen:

- Ja, in diesem Fall ist die Folge, die sich aus der Traversierung der Matrix ergibt, eindeutig
- Nein, andere Permutationen der Folge wandeln den gegebenen String in gleich vielen Schritten auch in den gesuchten String um
- Nein, es kann mehrere Möglichkeiten für die Traversierung geben (mit anderen Wörtern)
- Nein, es gibt beliebig viele Folgen (weil in der Aufgabenstellung keine Folge mit minimaler Anzahl an Operationen gefordert war)

c) Bei Smartphone-Tastaturen werden oft für Korrekturvorschläge die $n$ besten Worte aus einem Wörterbuch bestimmt. Die Anzahl der Worte im Wörterbuch ist dabei sehr groß. In dieser Aufgabe sollen Sie einen Algorithmus in Pseudocode entwerfen, welcher beliebige Eingabestrings entgegennimmt und das Wort mit der geringsten Levenshtein-Distanz aus dem Wörterbuch findet. Nutzen Sie die Sortiertheit aus!

Im folgenden Pseudocode soll `a:[char]` der String `a` mit beliebiger Länge sein. Sie können mit `|a|` die Länge des Strings auslesen und mit `a[i]` für $i \in [0, |a| - 1]$ auf das i-te Zeichen im String zugreifen.

Gehen Sie beim Entwurf wie folgt vor:

i) Entwerfen Sie eine Funktion in Pseudocode, um die Länge des längsten gemeinsamen Präfixes zweier Strings $a$ und $b$ zu bestimmen! **(6 Punkte)**

**Function** longestPrefixLength(a : [char], b : [char]) : int
{

Lösungsvorschlag

i ← 0
**while** $i <$ *min(|a|, |b|) && a[i] == b[i]* **do**
 | i++
**end**
**return** $i$

}

ii) Entwerfen Sie eine Funktion in Pseudocode, um die DP-Matrix zur Bestimmung der Levenshtein-Distanz zu berechnen! Dafür erhalten Sie als Eingabe eine DP-Matrix, die bereits teilweise gefüllt ist und sollen die fehlenden Werte berechnen. Die Funktion erhält folgende Parameter:

- Strings $a$ und $b$
- Länge $l$ eines Präfixes von $b$
- 2D-Array *mat*, um das Ergebnis der DP-Matrix zu speichern

Die übergebene DP-Matrix *mat* enthält schon den Teil des Ergebnisses bis zur Spalte $l$ (siehe Abb.). **(8 Punkte)**

**Achtung: Die gedruckte Version der Klausur**

Es gilt $M \geq |b|, N \geq |a|$ und $l \in [0..|b|]$. Wird für die Länge des Präfixes $l = 0$ übergeben, dann sind keine Einträge vorberechnet.

**wurde korrigiert zu**

Es gilt $M > |b|, N > |a|$ und $l \in [-1..|b|]$. Wird für die Länge des Präfixes $l = -1$ übergeben, dann sind keine Einträge vorberechnet.

**Function** computeDP(a : [char], b : [char], l : int, mat : [[int; M]; N])
{

Lösungsvorschlag

**Der Lösungsvorschlag ist für die korrigierte Version angegeben.**

```
// Diese Initialisierung muss im Prinzip nur gemacht werden, wenn l = −1, ist aber
für jede DP-Matrix gleich:
for i ∈ 0../a/ do
  | mat[i][0] ← i
end

// Diese Initialisierung ist notwendig, da die Spalten ab l noch nicht unbedingt
gefüllt sind:
for i ∈ max(0, l)../b/ do
  | mat[0][i] ← i
end

for j ∈ max(l+1,1)../b/ do
  |  for i ∈ 1../a/ do
  |  |  mat[i][j] ← min(
  |  |  mat[i][j-1] + 1,
  |  |  mat[i-1][j] + 1,
  |  |  mat[i-1][j-1] + (a[i-1] == b[j-1])?0:1
  |  |  )
  |  end
end
```

}

iii) Entwerfen Sie eine Funktion in Pseudocode für die Suche nach einem Wort mit der geringsten Levenshtein-Distanz im sortierten Wörterbuch `dict` zum Eingabewort `input`, und geben Sie das gefundene Wort zurück! Nutzen Sie die Funktionen aus i) und ii)! Berechnen Sie die Levenshtein-Distanz zwischen der Eingabe und allen Wörtern im Wörterbuch! Machen Sie sich zu Nutze, wenn zwei aufeinanderfolgende Worte im Wörterbuch ein gemeinsames Präfix besitzen! **(12 Punkte)**

```
// l_max: Länge des längsten Wortes im Wörterbuch
Function bestWord(dict : [[char]; N], input : [char], l_max : int) : [char]
{
```

Lösungsvorschlag

```
mat ← [[int; l_max]; |input|]
id ← 0
mindist ← |input| + l_max // kürzeste bisher gefundene Distanz
lp ← 0 // longest prefix
for x ← 0..N do
    computeDP(a, dict[x], lp, mat)
    d ← mat[|input|-1][|dict[x]|-1] // DP-Matrix unten rechts auslesen
    if d < mindist then
        mindist ← d
        id ← x
    end
    if x < N then
        lp ← longestPrefixLength(dict[x],dict[x+1])
    end
end
return dict[id]
```