Prof. Dr. Ir. Joost-Pieter Katoen

Jonathan Heinen, Sabrina von Styp

Hinweise:

- Die Übungsblätter sollen in Gruppen von je 3 Studierenden aus der gleichen Kleingruppenübung bearbeitet werden.
- Die Lösungen müssen bis Montag, den 26. April um 11:00 Uhr in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55).
- Namen und Matrikelnummern der Studenten sowie die Nummer der Übungsgruppe sind auf jedes Blatt der Abgabe zu schreiben. Heften bzw. tackern Sie die Blätter!

Aufgabe 1 (Laufzeit): (3+2+7 Punkte)

Gegeben sei der folgende Suchalgorithmus:

```
bool search(int E[], int n, int K) {
  int left = 0, right = n - 1;
  while (left <= right) {
    if (E[left] == K || E[right] == K) {
      return true;
    }
    left = left + 1;
    right = right - 1;
  }
  return false;
}
```

Als Eingabe erhält er ein Array E mit n Einträgen sowie einen Schlüssel K für den er überprüft, ob er in dem Eingabe-Array enthalten ist. Gehen Sie davon aus, dass der gesuchte Schlüssel mit einer Wahrscheinlichkeit von 0.3 im Array enthalten ist und jeder Schlüssel maximal einmal im Array vorkommt.

Bestimmen Sie in Abhängigkeit von n ...

a) die exakte Anzahl der Vergleiche im Worst-Case.

b) die exakte Anzahl der Vergleiche im Best-Case.

c) die exakte Anzahl der Vergleiche im Average-Case.

Lösung:

In Dieser Lösung Gehen Wir Davon Aus, Dass Nur Vergleiche Mit Einträgen Im Array Bei Bestimmung Der Laufzeit gezählt werden, nicht aber der Vergleich, der in der der Schleifenbedingung stattfindet. Darüber hinaus gehen wir von einer nicht strikten Auswertung von booleschen Ausdrücken aus (wie es auch bei Java üblich ist), d.h. insbesondere, dass wenn die linke Seite eines oder-Ausdruckes (||) zu *true* ausgewertet wurde die rechte Seite nicht mehr ausgewertet wird. In dieser Aufgabe findet somit innerhalb der if-Bedingung der rechte Vergleich nicht statt, wenn bereits der linke Vergleich zutrifft.

a) Worst-case Analyse $W(n)$
Die schlechteste Laufzeit ergibt sich, wenn das Element nicht im Array vorhanden ist. In diesem Fall wird die while-Schleife durchlaufen bis die Schleifenbedinung verletzt wird. Die Bedingung der while-Schleife lässt sich umschreiben in $right - left + 1 > 0$. In jedem Schleifendurchlauf wird $left$ um eins erhöht und $right$ um eins erniedrigt. Somit wird $right - left + 1$ mit jeder Iteration um zwei kleiner. Zu Begin gilt

$right - left + 1 = n$ und somit ist die Schleifenbedingung nach $\lceil \frac{n}{2} \rceil$ Iterationen verletzt. Da in jeder Iteration zwei Vergleich stattfinden ergibt sich eine Worst-case Laufzeit von:

$$W(n) = \lceil \frac{n}{2} \rceil \cdot 2$$

b) Best-case Analyse $B(n)$
Die beste Laufzeit wird erreicht, wenn das Element an Position 0 steht. Dort wird es nach dem ersten Vergleich gefunden. Somit ergibt sich:

$$B(n) = 1$$

c) Average-case Analyse $A(n)$
Die Average-case Laufzeit ergibt sich wie folgt aus der Laufzeit $A_{K \in E}(n)$ für den Fall, dass K in E enthalten ist, sowie $A_{K \notin E}(n)$ für den Fall, dass K nicht in E enthalten ist:

$$A_{K \in E}(n) \cdot 0,3 + A_{K \notin E}(n) \cdot 0,7$$

Wie bereits in Aufgabenteil b) bestimmt gilt $A_{K \notin E}(n) = W(n) = \lceil \frac{n}{2} \rceil \cdot 2$. Für $A_{K \in E}(n)$ ergibt sich die folgende Laufzeit:

$$\begin{aligned}
A_{K \in E}(n) &= \sum_{i=0}^{\frac{n}{2}-1} Pr\{K = E[i] | K \in E\} \cdot t(K == E[i]) + Pr\{K = E[n-1-i] | K \in E\} \cdot t(K == E[n-1-i])) \\
&= \sum_{i=0}^{\frac{n}{2}-1} \left(\frac{1}{n} \cdot (2i+1) + \frac{1}{n} \cdot (2i+2) \right) \\
&= \frac{1}{n} \cdot \sum_{i=0}^{\frac{n}{2}-1} (4i+3) = \frac{1}{n} \cdot \left(4 \cdot \sum_{i=0}^{\frac{n}{2}-1} i + \sum_{i=0}^{\frac{n}{2}-1} 3 \right) \\
&= \frac{1}{n} \cdot \left(4 \cdot \left(\frac{(\frac{n}{2}-1) \cdot (\frac{n}{2}-1+1)}{2} \right) + \frac{n}{2} \cdot 3 \right) = \frac{4 \cdot \frac{n}{2} \cdot (\frac{n}{2}-1)}{2 \cdot n} + \frac{3 \cdot n}{2 \cdot n} \\
&= \frac{n}{2} - 1 + \frac{3}{2} = \frac{n+1}{2}
\end{aligned}$$

Somit ergibt sich eine Average-case Laufzeit von:

$$A(n) = \frac{n+1}{2} \cdot 0,3 + \lceil \frac{n}{2} \rceil \cdot 1,4$$

Aufgabe 2 ($\mathcal{O}$-Notation): (3+3+3+3+3 Punkte)

Zeigen oder widerlegen Sie die folgenden Aussagen:

a) $g(n) = 347n^3 + 44n^2 + 749 \in \Theta(n^3)$
b) $2^{n+1} \in \Theta(2^n)$
c) $\log n \in \mathcal{O}(\sqrt{n})$
d) $max(f(n),\ g(n)) \in \Theta(f(n) + g(n))$
e) $g(n) + f(n) \in \mathcal{O}(g(f(n)))$

Lösung:

Lösung:

(a) Die Aussage gilt: $g(n) = 347n^3 + 44n^2 + 749 \in \Theta(n^3)$

$$\begin{aligned} & g(n) \in \Theta(n^3) \\ \Leftrightarrow \quad & \exists c_1, c_2 \in \mathbb{R}, n_0 \in \mathbb{N} : c_1 \cdot n^3 \leq g(n) \leq c_2 \cdot n^3 \ \forall n > n_0 \quad | \text{ nach Definition} \end{aligned}$$

Diese Aussage ist für $n_0 = 1, c_1 = \frac{1}{c_2}$ und $c_2 = 347 + 44 + 749 = 1140$ erfüllt.

q.e.d

(b) Die Aussage gilt: $2^{n+1} \in \Theta(2^n)$

$$\begin{aligned} & 2^{n+1} \in \Theta(2^n) \\ \Leftrightarrow \quad & \exists c_1, c_2 \in \mathbb{R}, n_0 \in \mathbb{N} : c_1 \cdot 2^n \leq 2^{n+1} \geq c_2 \cdot 2^n \ \forall n > n_0 \quad | \text{ nach Definition} \\ \Leftrightarrow \quad & \exists c_1, c_2 \in \mathbb{R}, n_0 \in \mathbb{N} : c_1 \cdot 2^n \leq 2 \cdot 2^n \geq c_2 \cdot 2^n \ \forall n > n_0 \\ \Leftarrow \quad & 2 \cdot 2^n \leq 2 \cdot 2^n \geq 2 \cdot 2^n \ \forall n > 1 \quad | \ c_1 = 2, c_2 = 2, n_0 = 1 \end{aligned}$$

q.e.d

Alternative Lösung:

$$\lim_{n\to\infty} \frac{2^{n+1}}{2^n} = \lim_{n\to\infty} 2 = 2$$

Hieraus folgt dass $2^{n+1} \in \Theta(2^n)$.

(c) Die Aussage gilt. Beweise:

Wir müssen zeigen dass $\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} \geq 0$ und $\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} < \infty$.

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} \overset{\text{L'Hôspital}}{=} \lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = \lim_{n\to\infty} \frac{2\sqrt{n}}{n} = 0$$

q.e.d

(d) Die Aussage gilt.

Es ist zu zeigen, dass Folgendes gilt:

$$\exists c_1, c_2 \in \mathbb{R}_{\geq 0}, \ n_0 \in \mathbb{N} : \ c_1 \cdot (f(n) + g(n)) \leq max(f(n), g(n)) \leq c_2 \cdot (f(n) + g(n)) \quad \forall n \geq n_0$$

für $c_1, c_2 \in \mathbb{R}_{\geq 0}$ und $n \geq n_0$ für ein hinreichend großes n_0 mit $n_0 \in \mathbb{N}$.

Wir beginnen mit $max(f(n), g(n)) \leq c_2 \cdot (f(n) + g(n))$:

Da $f(n)$ und $g(n)$ positive Funktionen sind gilt, $(f(n) + g(n)) \geq max(f(n), g(n))$. Aus dieser Abschätzung erhalten wir dann:

$$\exists c_2 \in \mathbb{R}_{\geq 0}, \ n_0 \in \mathbb{N} : \ max(f(n), g(n)) \leq c_2 \cdot max(f(n), g(n)) \quad \forall n \geq n_0$$

Dies gilt offensichtlich für beliebe $c_2 \geq 1$ und $n \geq n_0$.

Um den 2. Teil zu zeigen schätzen wir den Ausdruck $f(n) + g(n)$ nach oben hin ab und erhalten

$$f(n) + g(n) \leq 2 \cdot max(f(n), g(n)).$$

Nach einsetzen in die Ungleichung ergibt sich dann:

$$c_1 \cdot 2 \cdot max(f(n), g(n)) \leq max(f(n), g(n)).$$

Wähle nun z.B. $c_1 = \frac{1}{4}$, dann gilt:

$$\frac{1}{2} \cdot max(f(n), g(n)) \leq max(f(n), g(n)).$$

q.e.d

(e) Die Aussage gilt nicht. Gegenbeispiel:
Sei $g(n) = 1$ und $f(n) = n^2$ dann ist $n^2 \notin \mathcal{O}(1)$ und das gilt, da es kein $c \in \mathbb{R}_{\geq 0}$ gibt so dass:

$$n^2 \leq c \cdot 1$$

denn egal wie groß die Konstante c gewählt wird für $n \geq n_0$ für hinreichend große n_0 mit $n_0 \in \mathbb{N}$ gilt $n^2 > c$.

q.e.d

Aufgabe 3 (Beweise): (5+5 Punkte)

Zeigen oder widerlegen Sie die folgenden Aussagen:

a) $o(f(n)) \cap \omega(f(n)) = \emptyset$
b) Aus $f(n) \in \Omega(g(n))$ und $g(n) \in \Omega(h(n))$ folgt $f \in \Omega(h(n))$

Lösung:

Lösung:
(a) Die Aussage gilt.
Beweis durch Widerspruch: Angenommen der Schnitt sei nicht leer dann gibt es eine Funktion $g(n)$ für die gilt:

$$g(n) \in o(f(n)) \ \wedge \ g(n) \in \omega(f(n)).$$

Aus $g(n) \in o(f(n))$ folgt:

$$\forall c \in \mathbb{R}_{>0}, \exists n_1 \in \mathbb{N} \text{ mit } \forall n \geq n_1 : 0 \leq g(n) < c \cdot f(n) \quad (1)$$

Aus $g(n) \in \omega(f(n))$ folgt:

$$\forall c \in \mathbb{R}_{>0}, \exists n_2 \in \mathbb{N} \text{ mit } \forall n \geq n_2 : c \cdot f(n) < g(n) \quad (2)$$

Aus den beide Gleichungen (1) und (2) folgt, dass für $n_0 = max(n_1, n_2)$ gilt:

$$\forall c \in \mathbb{R}_{>0}, \forall n \geq n_0 : 0 \leq g(n) < c \cdot f(n) < g(n)$$

Hiermit ergibt sich ein Widerspruch woraus folgt, dass die Annahme $o(f(n)) \cap \omega(f(n)) \neq \emptyset$ falsch ist.

q.e.d

(b) Beweis:
Aus $f(n) \in \Omega(g(n))$ folgt:

$$\exists c_1 \in \mathbb{R}_{>0}, n_1 \in \mathbb{N}: \quad 0 \leq c_1 \cdot g(n) \leq f(n)\ \forall n \geq n_1\ \forall n > n_1$$
$$\exists c_1 \in \mathbb{R}_{>0}, n_1 \in \mathbb{N}: \quad 0 \leq g(n) \leq \frac{1}{c_1} \cdot f(n)\ \forall n \geq n_1\ \forall n > n_1$$

Aus $g(n) \in \Omega(h(n))$ folgt: $\exists c_2 \in \mathbb{R}_{>0}, n_2 \in \mathbb{N}: \ 0 \leq c_2 \cdot h(n) \leq g(n)\ \forall n \geq n_2$.
Es gilt folgendes:

$$\begin{aligned}
& 0 \leq c_2 \cdot h(n) \leq g(n) \quad \forall n \geq n_2 \\
\Leftrightarrow \quad & 0 \leq c_2 \cdot h(n) \leq g(n) \leq \frac{1}{c_1} \cdot f(n) \quad \forall n \geq \underbrace{max(n_1, n_2)}_{n_0} \\
\Leftrightarrow \quad & 0 \leq \underbrace{c_1 \cdot c_2}_{c_3} \cdot h(n) \leq f(n) \quad \forall n \geq n_0 \\
\Rightarrow \ \exists c_3 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}: \quad & f(n) \in \Omega(h(n)) \quad \forall n \geq n_0
\end{aligned}$$

q.e.d

Prof. Dr. Ir. Joost-Pieter Katoen

Jonathan Heinen, Sabrina von Styp

Hinweise:

- Die Übungsblätter sollen in Gruppen von je 3 Studierenden aus der gleichen Kleingruppenübung bearbeitet werden.
- Die Lösungen müssen bis Montag, den 3. Mai um 11:00 Uhr in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55).
- Namen und Matrikelnummern der Studenten sowie die Nummer der Übungsgruppe sind auf jedes Blatt der Abgabe zu schreiben. Heften bzw. tackern Sie die Blätter!
- Bitte beachten Sie, dass am 4. Mai wegen der Fachschaftsvollversammlung von 10:00 - 14:00 Uhr keine Übungen stattfinden. Für die Gruppen 6-9 wird es Ausweichtermine geben, die ab Mittwoch den 28. April auf der Internetseite zur Vorlesung angekündigt werden.

Aufgabe 1 (Baumeigenschaften): (3+5+4 Punkte)

Beweisen Sie folgende, in der Vorlesung eingeführte, Fakten für Binärbäume:

a) Ein Binärbaum enthält höchstens 2^d Knoten in Ebene d.

b) Ein Binärbaum mit Höhe h kann maximal $2^{h+1} - 1$ Knoten enthalten.

c) Ein Binärbaum mit n Knoten hat mindestens die Höhe $\lceil \log_2(n+1) \rceil - 1$.

Lösung:

Beweisen Sie folgende, in der Vorlesung eingeführte, Fakten für Binärbäume:
Ein Binärbaum...

a) enthält höchstens 2^d Knoten in Ebene d.

b) mit Höhe h kann maximal $2^{h+1} - 1$ Knoten enthalten.

c) hat, wenn er n Knoten enthält, mindestens Höhe $\lceil \log_2(n+1) \rceil - 1$.

zu (a):

Der Beweis erfolgt per Induktion über d.

Induktionsanfang: $d = 0$

$$2^0 = 1$$

Induktionsvorraussetzung: Die Aussage gilt für d.

Für Ebene $d+1$ haben wir dann:

$$2^{d+1} \text{ Knoten} = \underbrace{2^d}_{\text{Anzahl der Knoten in Ebene d}} \cdot 2$$

Da jeder Knoten maximal 2 Nachfolger hat folgt daraus dass Ebene $d+1$ maximal $2 \cdot n$ Knoten hat, wobei n die Anzahl der Knoten in Ebene d ist. Somit haben wir gezeigt, dass ein Baum auf Ebene höchstens 2^d Knoten hat.

q.e.d

zu (b):

Der Beweis erfolgt per Induktion über h.

Induktionsanfang: $h = 0$:

$$2^{0+1} - 1 = 1$$

Induktionsvorraussetzung: Die Aussage gilt für h.

Zu zeigen ist, dass die Aussage auch für $h+1$ gilt dh. Höhe $h+1$ hat maximal $2^{h+2} - 1$ Knoten.

$$\begin{aligned} & 2^{h+2} - 1 \\ = \; & 2^{h+1} \cdot 2 - 1 \\ = \; & \underbrace{2^{h+1} - 1}_{(1)} + \underbrace{2^{h+1}}_{(2)} \end{aligned}$$

(1) ist per Induktionsvorraussetzung die maximale Anzahl der Knoten für einen Baum der Höhe h und (2) ist die maximale Anzahl der Knoten in Ebene $h+1$. Somit kann ein Baum der Höhe $h+1$ nicht mehr als $2^{h+1} - 1 + 2^{h+1}$ Knoten haben und somit gilt die Aussage auch für $h+1$

q.e.d

zu (c):

Aus Teil (b) wissen wir, dass ein Baum mit Höhe h maximal $n = 2^{h+1} - 1$ viele Knoten hat. Durch Äquivalenzumformung erhalten wir dann:

$$\begin{aligned} n &= 2^{h+1} - 1 \\ \Leftrightarrow \; n - 1 &= 2^{h+1} \\ \Leftrightarrow \; \log_2(n-1) &= h + 1 \\ \Leftrightarrow \; \log_2(n-1) - 1 &= h \end{aligned}$$

Da die Höhe immer ganzzahlig ist müssen wir den berechneten Wert noch auf den nächst hören ganzzahligen Wert erhöhen. Es ergibt sich:

$$h = \lceil \log_2(n+1) \rceil - 1$$

q.e.d

Aufgabe 2 (Alternative Queue-Implementierung): (10 Punkte)

Wir haben in der Vorlesung Queues kennengelernt. Bei der vorgestellten Implementierung kann der letzte freie Eintrag im Array nicht mehr gefüllt werden, da eine komplett volle Queue nicht von einer leeren zu unterscheiden wäre. Sie sollen nun eine alternative Implementierung entwickeln, die diese Einschränkung nicht besitzt. Ersetzten Sie hierzu die Blöcke `A`, `B`, `C`, `D` und `E` in der folgenden Java-Implementierung:

```
public class Queue {
  public Queue(int N) {
    data = new int[N];
  }
  public boolean isEmpty() {
    return head == tail;                          A
  }
  public boolean isFull() {
    return head == (tail + 1) % data.length;      B
  }
  public void enqueue(int e) {
    data[tail] = e;                               C
    tail = (tail + 1) % data.length;
  }
  public int dequeue() {
    int e = data[head];                           D
    head = (head + 1) % data.length;
    return e;
  }
  private int head, tail;                         E
  private int[] data;
}
```

Lösung:

In der angegebenen Lösung ist bekannt wo sich Kopf und Schwanz der Schlang befinden. Es ergibt sich jedoch das Problem, dass man nicht zwischen einer vollen sowie leeren Schlange unterscheiden kann. In beiden Fällen zeigen Kopf und Schwanz auf die selbe Position.
Alternativ kann man sich auch die Position des Kopfes sowie die Anzahl der vorhandenen Elemente merken. Die Position des Schwanzes lässt sich dann aus diesen Werten berechnen. Für die Überprüfung ob die Warteschlange leer oder voll ist reicht es dann die Anzahl der vorhandenen Elemente zu überprüfen:

Block A:
```
    return size == 0;
```

Block B:
```
    return size >= data.length;
```

Block C:
```
    data[(head+size) % data.length] = e;
    size++;
```

Block D:
```
    int e = data[head];
    head = (head + 1) % data.length;
    size--;
    return e;
```

Block E:
```
  private int head, size;
```

Aufgabe 3 (Laufzeitanalyse): (6 Punkte)

Bestimmen Sie die Komplexitätsklasse für den Aufruf `berechne(n)` in Abhängigkeit von n. Gehen Sie davon aus, dass die Grundrechenarten +, -, *, / in konstanter Zeit O(1) ausgeführt werden, ebenso die Zuweisungen = und Vergleiche <=.

```
int berechne(int k){

    int result = k;

    for(int i = k; i > 0; i = i/2){
        result = k * result;
        for(int j = 0; j < i; j++){
            result++;
        }
    }

    return result;
}
```

Lösung:

Das vorgegeben Programm besitzt eine verschachtelte Schleife. Die Zählervariable i der äußeren startet bei n und wird in jedem Durchlauf halbiert und besitzt somit im i-ten Durchlauf den Wert $\frac{n}{2^i}$. D.h. die äußere Schleife wird exakt $\lfloor log_2(n) \rfloor$ mal durchlaufen bevor sie den Wert $\frac{n}{2^{log_2(n+1)}} = \frac{n}{n+1} < 1$ annimmt und damit die Schleifenbedingung verletzt wird.
Die innere Schleife wird jeweils von 0 bis zur Zählervariable der äußeren Schleife hochgezählt, wird also in der $k$ten Iteration $\frac{n}{2^k}$ Mal durchlaufen. Alle weiteren Anweisungen haben konstante Laufzeit. Somit ergibt sich eine Laufzeit von:

$$\frac{n}{2^0} + \frac{n}{2^1} + \frac{n}{2^2} \cdots + \frac{n}{2^{log_2 n}} = \sum_{i=0}^{log_2 n} \frac{n}{2^i} = n \cdot \sum_{i=0}^{log_2 n} \frac{1}{2^i} = n \cdot \frac{2^{log_2 n+1} - 1}{2^{log_2 n}} = n \cdot \frac{2 \cdot n - 1}{n} = 2n - 1 \in \Theta(n)$$

Aufgabe 4 (Effiziente Implementierung): (10 Punkte)

Schreiben Sie einen Algorithmus der den kleinsten und größten Schlüsselwert eines übergebenen Arrays E zurückgibt. Hierbei sei n die Anzahl der Elemente im Array. Der Algorithmus soll im Worst-Case $W(n) = 1,5 \cdot n + c$ Schlüsselwerte (mit konstanten Wert c) vergleichen.

Hinweis:

- Es sollen nur Vergleiche zwischen Schlüsselwerten gezählt werden, nicht aber Vergleiche zwischen Zählervariablen wie sie in Schleifenbedingungen stattfinden.
- Versuchen Sie mit nur **vier** Vergleichen aus vier Werten den maximalen so wie minimalen herauszufinden und erweitern Sie diesen Ansatz dann auf beliebig viele Werte.

Lösung:

Der folgende Algorithmus bestimmt den kleinsten und größten Schlüsselwert eines Arrays E der Länge N mit einer Laufzeit von $3 \cdot \frac{n-1}{2}$ indem er jeweils Min und Max zweier Elemente mit einem Vergleich bestimmt und zwei weiter Vergleiche nutzt um diese dann gegen das globale Minimum und Maximum zu vergleichen.

```
(Key, Key) findMinAndMax(Array E, int N) {
    Key min, max;
    min = max = E[N-1];

    for (int i = 0; i < N-1; i+=2) {

        if (E[i] < E[i+1]) {
            min = min(E[i], min);
            max = max(E[i+1], max);
        } else {
            min = min(E[i+1], min);
            max = max(E[i], max);
        }
    }
    return (min, max)
}
```

Der oben angegebene Algorithmus als Java - Implementierung:

```
public class MinMax{

    public static void main(String[] args){
        minmax(new int[]{1,3,4,5,6,72,3,2,32,4,32,56,6,7,3,4,51,0});
    }

    public static void minmax(int[] array){

        int min, max;
        min = max = array[array.length-1];

        for(int i = 0; i < array.length -1; i+=2){

            if(array[i] < array[i+1]){
                min = (array[i] < min)?array[i] :min;
                max = (array[i+1] > max)?array[i+1]:max;
            }else{
                min = (array[i+1] < min)?array[i+1]:min;
                max = (array[i] > max)?array[i] :max;
            }
        }
        System.out.println("Maximum: " + max + ", minimum: " + min);
    }
}
```

Prof. Dr. Ir. Joost-Pieter Katoen

Jonathan Heinen, Sabrina von Styp

Hinweise:

- Die Übungsblätter sollen in Gruppen von je 3 Studierenden aus der gleichen Kleingruppenübung bearbeitet werden.
- Die Lösungen müssen bis Montag, den 10. Mai um 11:00 Uhr in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55).
- Namen und Matrikelnummern der Studenten sowie die Nummer der Übungsgruppe sind auf jedes Blatt der Abgabe zu schreiben. Heften bzw. tackern Sie die Blätter!
- Bitte beachten Sie, dass am 4. Mai wegen der Fachschaftsvollversammlung von 10:00 - 14:00 Uhr keine Übungen stattfinden. Die Ausweichtermine wurden per Email den betroffenen Studierenden mitgeteilt.

Aufgabe 1 (Maximale Laufzeit): (8 Punkte)

Geben Sie einen Algorithmus an, der in $\mathcal{O}(n)$ für eine Folge von n ganzen Zahlen (gegeben als Array) eine maximale Teilfolge findet. Eine Teilfolgen wird hierbei von beliebig vielen (maximal n) *aufeinanderfolgender* Zahlen gebildet. Sie ist maximal, wenn die Summe ihrer Elemente maximal ist, d.h. wir suchen aus allen möglichen Teilfolgen eine mit maximaler Summe.
Die Teilfolge soll dabei als *Startindex*, *Endindex* sowie *Summe* der Folge ausgegeben werden. Die Eingangsfolge

$$12,\ -34,\ 56,\ -5,\ -6,\ 78,\ -32,\ 8$$

liefert beispielsweise die Indizes 3 und 6 sowie die Summe $56 - 5 - 6 + 78 = 123$.

Lösung:

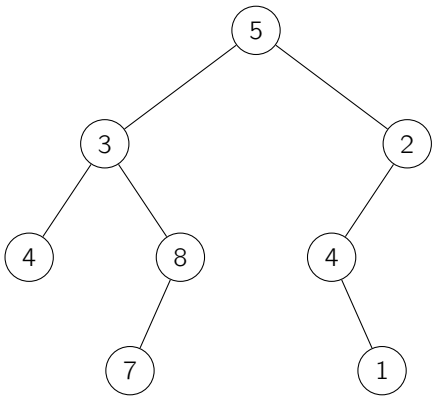
Der folgende Algorithmus bestimmt zu einer Folge die Teilfolge mit maximaler Summe.

```
public static void maxSum(int[] array){

    // die folgenden drei Variablen benoetigen wir um uns die
        Eckdaten der
    // maximalen Folge zu merken.
    int max = 0;
    int maxStart = 0;
    int maxEnd = -1;

    // start ist die Startposition der aktuell betrachteten Folge,
        sum ihre Summe.
    int start = 0;
    int sum = 0;

    for(int i = 0; i < array.length; i++){

        // Wir erweitern die aktuell betrachtete Folge um die
            Position i
        sum += array[i];

        // Ist die Summe der aktuellen Folge kleiner Null, so hat
            jede Folge,
        // die dahinter startet eine hoehere Summe als die aktuelle
            auf diese Position
        // erweitert.
```

```
            if(sum <= 0){
                sum = 0;
                start = i+1;
            }

            // Wenn eine neue maximale Folge gefunden wurde merken wir
                 uns die Eckdaten dieser
            if(sum > max){
                max = sum;
                maxStart = start;
                maxEnd = i;
            }

        }
        // Die Eckdaten der gefundenen Teilfolge werden ausgegeben
        System.out.printf("Das Intervall von %d bis %d besitzt die
             maximale Summe %d ", maxStart+1, maxEnd+1, max);
    }
```

Aufgabe 2 (Baumtraversierung): (3+4+3 Punkte)

a) Geben Sie jeweils das Ergebnis der in-, pre- und post-order Traversierung des folgenden Baumes an:



b) Bestimmen Sie zu den folgenden Paaren von Linearisierungen den jeweils zugehörigen Baum:

(i) in-order: 8 4 5 7 1 2 6 9 3
pre-order: 1 4 8 7 5 3 6 2 9

(ii) in-order: 3 2 6 1 5 4 7 9 8
post-order: 3 6 1 2 4 8 9 7 5

c) Geben Sie ein minimales Beispiel für zwei unterschiedliche Bäume an, die sowohl die gleiche pre- als auch post-order Linearisierung besitzen. Minimal bedeutet hier mit der kleinstmöglichen Anzahl von Elementen wobei jedes Element maximal einmal pro Baum enthalten sein darf.

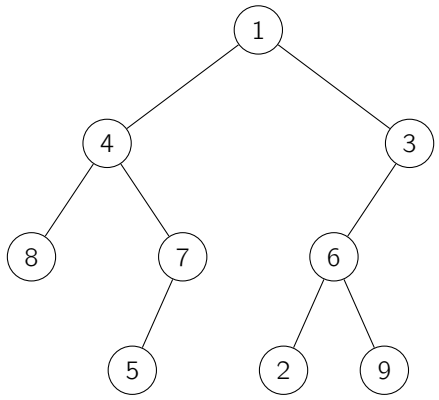
Lösung:

a) Die Linearisierung sind wie folgt:

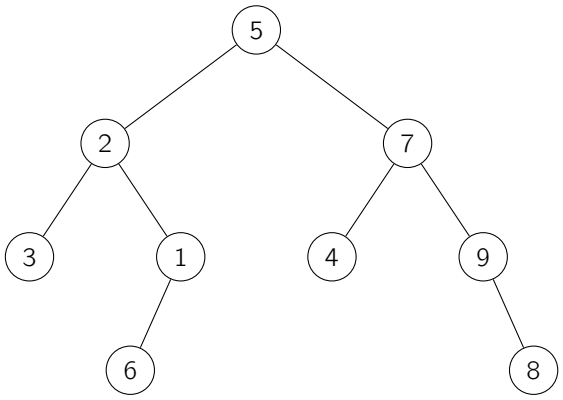
preorder:	5 3 4 8 7 2 4 1
inorder:	4 3 7 8 5 4 1 2
postorder:	4 7 8 3 1 4 2 5

b) Es ergeben sich die folgenden Graphen:

(i)



(ii)



c) Für die folgenden Bäume ist die preorder Linearisierung >1 2< und die postorder Linearisierung >2 1<:

Aufgabe 3 (Analyse rekursiven Codes): (3+3+3 Punkte)

Geben Sie für die folgenden Programme die Laufzeitkomplexität als Rekursionsgleichung in Abhängigkeit des Eingabeparameters n an:

a)

```
int berechne1(int n){

    int sum = 0;

    for(int i = 0; i < n/2; i++){
        sum += 2*i - 1;
    }
    if(n <= 0)
        return sum;
    else
        return sum + 4 * berechne1(n-1) + 5;
}
```

b)

```
int berechne2(int n){

    if(n <= 0)
        return n*n;
```

```
    int wert = berechne2(n-3) * (berechne2(n-3) + 2);

    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            sum += i - j;
        }
    }
    return wert * berechne2(n-3);
}
```

c)

```
int berechne3(int n){

    if(n <= 0)
        return 5;
    else
        return berechne3(n-1) * berechne3(n-2) * berechne3(n-4);
}
```

Lösung:

zu a)

$$T(n) = T(n-1) + c_1 \cdot \frac{n}{2} + c_2$$

zu b)

$$T(n) = 3 \cdot T(n-3) + c_1 \cdot n^2 + c_2 \cdot n + c_3$$

zu c)

$$T(n) = T(n-1) + T(n-2) + T(n-4) + c$$

Aufgabe 4 (Rekursionsgleichungen): (3+3+3+3 Punkte)

a) Zeigen Sie mit Hilfe der Substitutionsmethode, dass für $T(n) = 2 \cdot T(\sqrt{n}) + log_2 n$ mit $T(1) = 1$ gilt, dass $T(n) = log_2 n \cdot log_2 log_2 n$

b) Raten Sie die Komplexitätsklasse von $T(n) = 2 \cdot T(n/2) + 4 \cdot T(n/4) + n$ mit $T(1) = 1$ und zeigen Sie mit Hilfe der Substitutionsmethode die Korrektheit Ihrer geratenen Lösung.

c) Raten Sie mit Hilfe des entsprechenden Rekursionsbaum die Komplexitätsklasse zu $T(n) = 3 \cdot T(n-1) + 3n + 5$ mit $T(1) = 1$ und zeigen Sie mit Hilfe der Substitutionsmethode die Korrektheit Ihrer geratenen Lösung.

d) Nutzen sie die Methode der Variablentransformation um die exakte Lösung der Rekursionsgleichung $T(n) = T(\sqrt{n}) + 1$ mit $T(1) = 1$ zu bestimmen.

Lösung:

zu a)
Wir setzen die Angenommene Lösung $T(n) = log_2 n \cdot log_2 log_2 n$ in $T(n)$ ein und überprüfen das Ergebnis:

$$
\begin{aligned}
& T(n) = 2 \cdot T(\sqrt{n}) + log_2 n && \text{| Einsetzen der Behauptung für } T(\sqrt{n}) \\
\Leftrightarrow \quad & T(n) = 2 \cdot log_2 \sqrt{n} \cdot log_2 log_2 \sqrt{n} + log_2 n \\
\Leftrightarrow \quad & T(n) = 2 \cdot \frac{1}{2} \cdot log_2 n \cdot log_2 (\frac{1}{2} \cdot log_2 n) + log_2 n \\
\Leftrightarrow \quad & T(n) = log_2 n \cdot (log_2 \frac{1}{2} + log_2 log_2 n) + log_2 n \\
\Leftrightarrow \quad & T(n) = log_2 n \cdot (-1 + log_2 log_2 n) + log_2 n \\
\Leftrightarrow \quad & T(n) = -log_2 n + log_2 n \cdot log_2 log_2 n + log_2 n \\
\Leftrightarrow \quad & T(n) = log_2 n \cdot log_2 log_2 n
\end{aligned}
$$

q.e.d.

Die Angenommene Lösung ist korrekt.
$T(n) = 2 \cdot T(\sqrt{n}) + \log_2 n$ für $n > 1$, und $T(1) = 1$.
Vermutung: Lösung ist $T(n) \in \mathcal{O}(\log n \cdot \log \log n)$
Dazu müssen wir zeigen: $\exists c \in \mathbb{R}_{>0}$, $n_0 \in \mathbb{N}$ so dass: $T(n) \leq c \cdot \log n \log \log n$ für alle $n \geq n_0$

$$
\begin{aligned}
T(n) &= 2 \cdot T(\sqrt{n}) + log_2 n && \text{| Induktionshypothese} \\
&\leq c \cdot (2 \cdot \log \sqrt{n} \cdot \log \log \sqrt{n}) + \log n \\
&= c \cdot (\log n \cdot \log \log \sqrt{n}) + \log n \\
&= c \cdot \log n \cdot ((\log \log n) - \log(2)) + \log n \\
&= c \cdot \log n \cdot \log \log n - c \cdot \log(n) \log(2) + \log n && \text{| } \log \equiv \log_2 \\
&= c \cdot \log n \cdot \log \log n + \log(n) \underbrace{(1 - \log(2))}_{=0} && \text{| } c > 1 \\
&\leq c \cdot \log n \cdot \log \log n
\end{aligned}
$$

q.e.d

zu b)
$T(n) = 2 \cdot T(\frac{n}{2}) + 4 \cdot T(\frac{n}{4}) + n$ für $n > 1$, und $T(1) = 1$. Vermutung: Lösung ist $T(n) \in \mathcal{O}(n^2 \cdot \log n)$
Dazu müssen wir zeigen: $\exists c \in \mathbb{R}_{>0}$, $n_0 \in \mathbb{N}$ so dass: $T(n) \leq c \cdot n^2 \log n$ für alle $n \geq n_0$

$$
\begin{aligned}
T(n) &= 2 \cdot T(\frac{n}{2}) + 4 \cdot T(\frac{n}{4}) + n && \text{|Induktionshypothese} \\
&\leq c \cdot \frac{n^2}{2} \cdot \log \frac{n}{2} + c \cdot \frac{n^2}{4} \cdot \log \frac{n}{4} + n \\
&= c \cdot \frac{n^2}{2} \cdot \log \frac{n}{2} + c \cdot \frac{n^2}{4} \cdot \log \frac{n}{4} + n && \text{|log-Rechnung: } (\log \equiv \log_2) \\
&= c \cdot \frac{n^2}{2} \cdot \log n - c \cdot \frac{n^2}{2} \cdot \log 2 + c \cdot \frac{n^2}{4} \cdot \log n - c \cdot \frac{n^2}{4} \cdot \log 4 + n \\
&\leq c \cdot \frac{3}{4} n^2 \cdot \log n - c \cdot n^2 + n \\
&\leq c \cdot n^2 \cdot \log n
\end{aligned}
$$

q.e.d

zu c)
$T(n) = 3 \cdot T(n-1) + 3n + 5$ für $n > 1$ und $T(1) = 1$.
Vermutung: Lösung ist von der Form $c \cdot 3^n \cdot n$

$$\begin{aligned} T(n) &= 3 \cdot T(n-1) + 3n + 5 \\ &\leq 3 \cdot (c \cdot 3^{n-1} \cdot (n-1)) + 3n + 5 \\ &= c \cdot (3^n \cdot n - 3^n) + 3n + 5 \\ &= c \cdot 3^n n - c \cdot 3^n + 3n + 5 \qquad \mid \text{für hinreichend große } c \text{ und } n \\ &\leq c \cdot 3^n n \end{aligned}$$

q.e.d

zu d)
Fehler in der Aufgabenstellung: statt $T(1) = 1$ muss gelten $T(2) = 2$

$$\begin{aligned} & T(n) = T(\sqrt{n}) + 1 && \mid \text{Variablentransformation } m = \log n \\ \Leftrightarrow \quad & T(2^m) = T(2^{\frac{m}{2}}) + 1 && \mid \text{Umbenennung } T(2^m) = S(m) \\ \Leftrightarrow \quad & S(m) = S(\frac{m}{2}) + 1 && \mid \text{Lösung bekannter Rekursionsgleichung} \\ \Leftrightarrow \quad & S(m) = \log m + S(1) && \mid \text{da } S(1) = T(2) = 2 \text{ (Anfangsbedingung)} \\ \Leftrightarrow \quad & S(m) = \log m + 2 && \mid m = \log n \\ \Leftrightarrow \quad & T(n) = \log \log n + 2 \end{aligned}$$

q.e.d

Prof. Dr. Ir. Joost-Pieter Katoen

Jonathan Heinen, Sabrina von Styp

Hinweise:

- Die Übungsblätter sollen in Gruppen von je 3 Studierenden aus der gleichen Kleingruppenübung bearbeitet werden.
- Die Lösungen müssen bis Montag, den 17. Mai um 11:00 Uhr in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55).
- Namen und Matrikelnummern der Studenten sowie die Nummer der Übungsgruppe sind auf jedes Blatt der Abgabe zu schreiben. Heften bzw. tackern Sie die Blätter!
- Am Freitag den 14.05.2010 findet keine Vorlesung statt.

Aufgabe 1 (Verbesserter Insertionsort): (4 + 5 Punkte)

Der folgende Algorithmus stellt eine Variante von Insertionsort dar, bei der die lineare Suche nach der Einfügeposition durch eine binären Suche ersetzt wurde.

```
int binaereSuche (int E[], int key, int left, int right){

    if (left == right)
        return left;

    int mid = left + ((right - left) / 2);

    if (key > E[mid])
        return binaereSuche (E, key, mid + 1, right);
    else if (key < E[mid])
        return binaereSuche (E, key, left, mid);

    return mid;
}


void sort(int[] E){

    int ins, i, j;
    int tmp;

    for (i = 1; i < E.length; i++) {

        ins = binaereSuche (E, E[i], 0, i);
        tmp = E[i];

        for (j = i - 1; j >= ins; j--)
            E[j + 1] = E[j];

        E[ins] = tmp;
    }
}
```

a) Bestimmen Sie jeweils $\Theta(T_{binaereSuche(E,k,0,n)})$ in Abhängigkeit von `n` im Worst- und Best-Case.

b) Nutzen sie das Ergebnis Ihrer Analyse aus **a)** um $\Theta(T_{sort(E)})$ in Abhängigkeit von der Größe des übergebenen Arrays `E.length = n` für den Worst- sowie Best-Case zu bestimmen.

Lösung:

a) Die beste Laufzeit wird erreicht, wenn sich das gesuchte Element an der Position $mid = n/2$, d.h. an der mittleren Position befindet. Dann hat die binäre Suche eine konstante Laufzeit d.h. $T_{binaereSuche(E,k,0,n)} \in \Theta(1)$.

Die schlechteste Laufzeit wird erreicht, wenn das Element nicht im übergebenen Array enthalten ist oder aber es erst im letzten Durchlauf gefunden wird. In diesem Fall gilt $T_{binaereSuche(E,k,0,n)} = T(n) = T(n/2) + c \in \Theta(\log n)$.

Die schlechteste Laufzeit wird insbesondere erreicht, wenn sich das gesuchte Element direkt an der Grenze eines Intervalls befindet, d.h. insbesondere auch beim ersten und letzten Element.

b) Die äußere Schleife wird in jedem Fall n Mal durchlaufen. Der Schleifenkörper besteht, neben Anweisungen die jeweils eine konstanten Laufzeit besitzen, aus zwei Teilen. Der erste Teil ist die binäre Suche nach der Einfügeposition im bereits sortiertem Bereich von 0 bis zur Laufvariable i. Da sich der gesuchte Schlüssel außerhalb des sortierten Bereichs befindet wird immer die Worst-Case Laufzeit in $\Theta(\log i)$ benötigt, egal an welcher Position das Element einzufügen ist. Der zweite Teil übernimmt das Verschieben der Elemente, das nötig ist um das Element einfügen zu können. Diese Schleife wird rückwärts von Position $i-1$ bis zur Einfügeposition durchlaufen. Sei j die Einfügeposition mit $0 \leq j < i$ dann ergibt sich die Laufzeit L des Schleifenkörpers in Abhängigkeit von der Größe des bereits sortierten Bereichs i und der Einfügeposition j als:

$$L(i,j) \in \Theta(\log i + (i-j))$$

Offensichtlich ist $\Theta(\log i + (i-j))$ (bei festem i) minimal wenn $j = i$, d.h. wenn das Element an der Position einzufügen ist, an der es bereits steht. Somit wird die Best-Case Laufzeit erreicht, wenn das Array bereits sortiert ist. Die Worst-Case Laufzeit wird hingegen erreicht, wenn $j = 0$ ist, d.h. wenn das Array invers sortiert ist.

Es ergibt sich folgende Best-Case Laufzeit:

$$B(n) = \sum_{i=1}^{n} (\log i) = \log \prod_{i=1}^{n} i = \log n!$$

Es ergibt sich folgende Worst-Case Laufzeit:

$$W(n) = \sum_{i=1}^{n} (\log i + i) = \log n! + \frac{n^2 + n}{2} \in \Theta(n^2)$$

Aufgabe 2 (Sortieralgorithmus): (5 + 3 Punkte)

```
sort(int [ ] A, int l, int r) {
  if (A[l] > A[r]){
    exchange(A[l], A[r]);
  }
  if (l < r-1){
    k:= (r-l+1) div 3;
    sort(A, l, r-k);
    sort(A, l+k, r);
    sort(A, l, r-k);
  }
}
```

a) Bestimmen Sie für den gegebenen Sortieralgorithmus die Komplexitätsklasse Θ im Best-, Worst- und Average-Case für den Aufruf `sort(A,0,n)` in Abhängigkeit von `n`, der Anzahl der zu sortierenden Elemente aus dem Array `A`.

b) Der vorgestellte Algorithmus ist nicht stabil. Ändern Sie den Algorithmus so ab, dass er stabil ist.

Lösung:

a) Die Laufzeit von `sort` ist im Best-, Worst- und Average-Case gleich. Sie kann mit folgender Rekursionsgleichung beschrieben werden:

$$T(n) = 3 \cdot T(\frac{2}{3}n) + 1$$

Dabei steht $3 \cdot T(\frac{2}{3}n)$ für die Komplexität der Rekursionsaufrufe und 1 für die Komplexität der Vergleiche.

Zur Bestimmung der Komplexitätsklasse verwenden wir das Mastertheorem.

Es gilt $b = 3$, $c = 1{,}5$ und $f(n) = c \in \mathcal{O}(n^0)$. Daraus ergibt sich $E = \log_{1,5} 3$, so dass $f(n) \in \mathcal{O}(n^{E-\epsilon})$. Dies ist der 1. Fall und wir erhalten:

$$T(n) \in \Theta(n^{\log_{1,5} 3})$$

b) Der Suchalgorithmus `sort` ist nicht stabil, da die Elemente vor den Rekursionsaufrufen vertauscht werden. Ändert man den Code so ab, dass die Elemente nach den Rekursionsaufrufen vertauscht werden, ist der Algorithmus stabil.

```
sort2(int [ ] A, int l, int r) {
  if (l < r-1){
    k:= (r-l+1) div 3;
    sort2(A, l, r-k);
    sort2(A, l+k, r);
    sort2(A, l, r-k);
  } else {
    if (A[l] > A[r]) then {
      exchange(A[l], A[r]);
    }
  }
}
```

Aufgabe 3 (Rekursionsbäume): (5 + 6 Punkte)

Erstellen Sie zu den folgenden Rekursionsgleichungen die entsprechenden Rekursionsbäume und lesen Sie eine möglichst kleine obere Schranke der Lösung ab. Überprüfen Sie die Schranke mit Hilfe des Substitutionsverfahrens.

a)

$$T(n) = 4 \cdot T(\frac{n}{3}) + \frac{n}{2} + 3 \text{ mit } T(0) = 1$$

b)

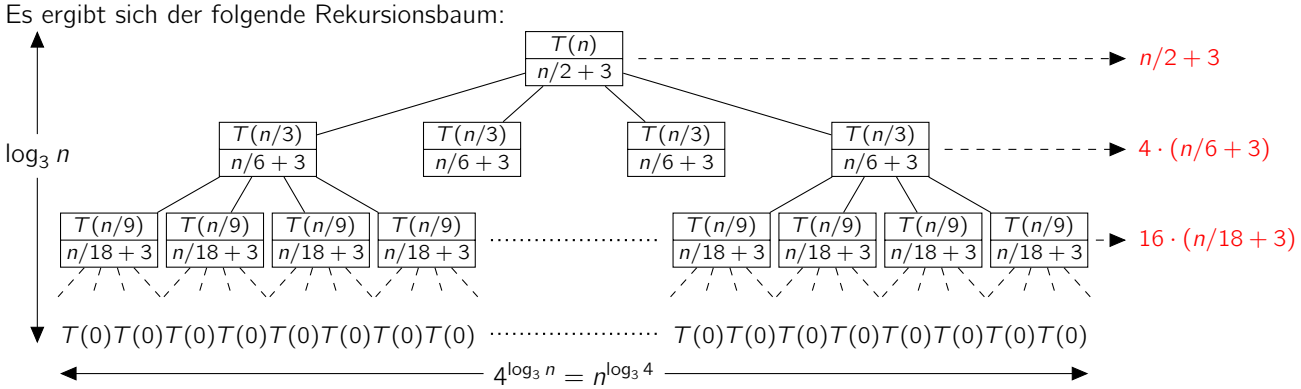
$$T(n) = T(n-2) + T(n-1) + 7 \text{ mit } T(0) = 1 \text{ und } T(1) = 7$$

Lösung:

a)

$$T(n) = 4 \cdot T(\frac{n}{3}) + \frac{n}{2} + 3$$

Es ergibt sich der folgende Rekursionsbaum:



Für Ebene i können wir eine Laufzeit von $4^i \cdot (\frac{n}{2 \cdot 3^i} + 3)$ ablesen. Bei einer Höhe des Rekursionsbaumes von $\log_3 n$ ergibt sich die Laufzeit als:

$$\sum_{i=0}^{\log_3 n} (4^i \cdot (\frac{n}{2 \cdot 3^i} + 3)) + n^{\log_3 4} \cdot T(0)$$

Lösen wir diese Gleichung auf:

$$
\begin{aligned}
&\sum_{i=0}^{\log_3 n}(4^i \cdot (\frac{n}{2\cdot 3^i}+3)) + n^{\log_3 4}\cdot T(0)\\
=&\sum_{i=0}^{\log_3 n}(\frac{4^i\cdot n}{2\cdot 3^i}+3\cdot 4^i) + n^{\log_3 4}\cdot 1\\
=&\frac{1}{2}\cdot n\cdot \sum_{i=0}^{\log_3 n}\left(\frac{4}{3}\right)^i + 3\cdot\sum_{i=0}^{\log_3 n}4^i + n^{\log_3 4}\\
=&\frac{1}{2}\cdot n\cdot \sum_{i=0}^{\log_3 n}\left(\frac{4}{3}\right)^i + 3\cdot\sum_{i=0}^{\log_3 n}4^i + n^{\log_3 4} \qquad \Big| \quad \sum_{i=0}^{n}x^i = \frac{x^{n+1}-1}{x-1}\\
=&\frac{1}{2}\cdot n\cdot \frac{\left(\frac{4}{3}\right)^{\log_3 n+1}-1}{\frac{1}{3}} + 3\cdot\frac{4^{log_3 n+1}-1}{3} + n^{\log_3 4}\\
=&\frac{3}{2}\cdot n\cdot (\frac{4}{3}\cdot\left(\frac{4}{3}\right)^{\log_3 n}-1) + 4\cdot 4^{\log_3 n} - 1 + n^{\log_3 4}\\
=&\frac{3}{2}\cdot n\cdot (\frac{4}{3}\cdot\frac{4^{\log_3 n}}{3^{\log_3 n}}-1) + 4\cdot 4^{\log_3 n} + n^{\log_3 4} - 1\\
=&\frac{3}{2}\cdot n\cdot (\frac{4}{3}\cdot\frac{n^{\log_3 4}}{n}-1) + 4\cdot n^{\log_3 4} + n^{\log_3 4} - 1\\
=&2\cdot n\cdot\frac{n^{\log_3 4}}{n} - 1\frac{1}{2}\cdot n + 5\cdot n^{\log_3 4} - 1\\
=&2\cdot n^{\log_3 4} - 1\frac{1}{2}\cdot n + 5\cdot n^{\log_3 4} - 1\\
=&7\cdot n^{\log_3 4} - 1\frac{1}{2}\cdot n - 1
\end{aligned}
$$

Mit Hilfe der Substitutionsmethode können wir zeigen, dass $7\cdot n^{\log_3 4} - 1\frac{1}{2}\cdot n - 1$ tatsächlich eine Oberschranke von $T(n)$ ist. Betrachten wir zuerst die Basisfälle:

Für $n = 0$ ist die Abschätzung offensichtlich -1 und somit kleiner als $T(0) = 1$, gleiches gilt für $T(1)$. Deshalb legen wir $n_0 = 2$ fest und zeigen, dass die Funktion eine obere Schranke für $T(1) = 4 * T(0) + 1/2 + 3 - 1 = 6,5$ liefert.

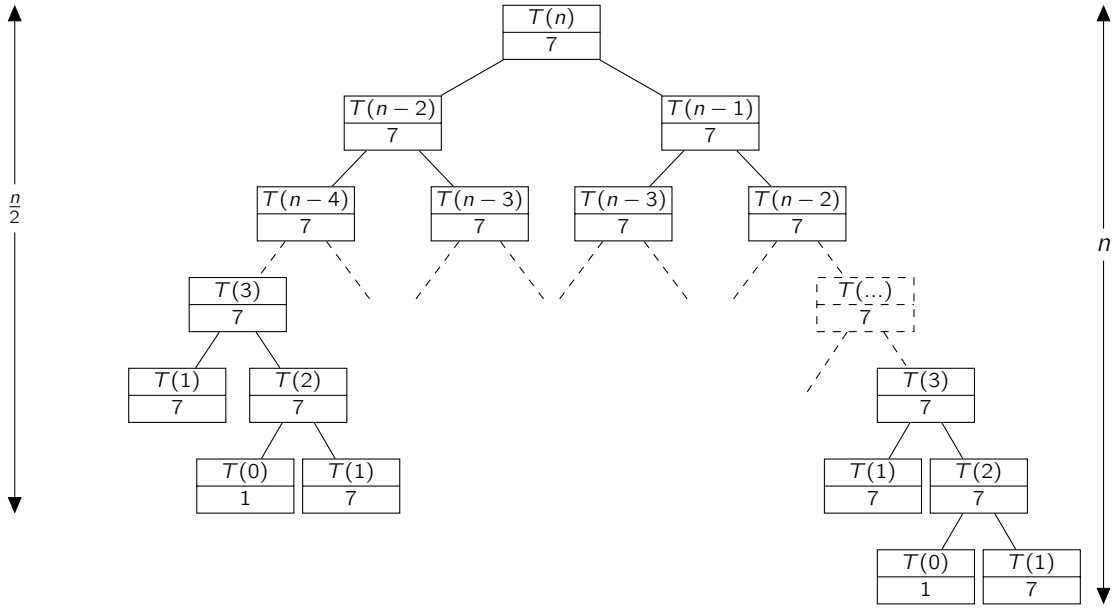
Wir zeigen nun, dass $7\cdot n^{\log_3 4} - 1\frac{1}{2}\cdot n - 1$ für alle $n > 0$ eine obere Schranke für $T(n)$ ist.

$$
\begin{aligned}
T(n) &= 4\cdot T(\frac{n}{3}) + \frac{n}{2} + 3\\
&\leq 4\cdot(7\cdot\left(\frac{n}{3}\right)^{\log_3 4} - 1\frac{1}{2}\cdot\frac{n}{3} - 1) + \frac{n}{2} + 3\\
&= 4\cdot 7\cdot\left(\frac{n}{3}\right)^{\log_3 4} - 2n - 4 + \frac{n}{2} + 3\\
&= 4\cdot 7\cdot\frac{n^{\log_3 4}}{3^{\log_3 4}} - \frac{3}{2}\cdot n - 1\\
&= 7\cdot n^{\log_3 4} - \frac{3}{2}\cdot n - 1
\end{aligned}
$$

q.e.d.

b)

$$T(n) = T(n-2) + T(n-1) + 7$$



Der Baum hat keine einheitliche Höhe. Das linkeste Blatt liegt auf der Ebene $\frac{1}{2}n$, das rechteste auf Ebene n. Alle anderen Blätter befinden sich zwischen diesen Ebenen, die Tiefe nimmt von Links nach Rechts hin zu. Wir können davon ausgehen, dass alle Blätter gleichmäßig auf die Tiefen zwischen $\frac{1}{2}n$ und n aufgeteilt sind. Somit können wir die Anzahl der Knoten und Blätter im Baum durch die Anzahl der Knoten und Blätter eines Baumes der Höhe $\frac{n+\frac{1}{2}n}{2} = \frac{3}{4}n$ abschätzen. Es gibt somit ungefähr $2^{\frac{3}{4}n+1} - 1$ viele Knoten und Blätter im Baum. Jeder dieser Knoten und Blätter hat eine konstante Laufzeit von maximal 7. Somit ergibt sich eine Laufzeit von $(2^{\frac{3}{4}n+1} - 1) \cdot 7$

Wir zeigen nun mit Hilfe der Substitutionsmethode, dass $(2^{\frac{3}{4}n+1} - 1) \cdot 7$ eine obere Schranke für T(n) ist:

$$\begin{aligned}
T(n) &= T(n-2) + T(n-1) + 7 && \text{|Einsetzen: } T(n) \leq (2^{\frac{3}{4}n+1} - 1) \cdot 7 \\
&\leq (2^{\frac{3}{4}(n-1)+1} - 1) \cdot 7 + (2^{\frac{3}{4}(n-2)+1} - 1) \cdot 7 + 7 \\
&= (2^{\frac{3}{4}(n-1)+1} - 1 + 2^{\frac{3}{4}(n-2)+1} - 1 + 1) \cdot 7 \\
&= (2^{\frac{3}{4}n+\frac{1}{4}} + 2^{\frac{3}{4}n-\frac{2}{4}} - 1) \cdot 7 \\
&= (2^{\frac{3}{4}n} \cdot \underbrace{(2^{\frac{1}{4}} + 2^{-\frac{2}{4}})}_{\approx 1{,}89} - 1) \cdot 7 \\
&\leq (2^{\frac{3}{4}n} \cdot 2 - 1) \cdot 7 \\
&= (2^{\frac{3}{4}n+1} - 1) \cdot 7
\end{aligned}$$

Es bleibt zu zeigen, dass die abgelesene Laufzeit auch eine obere Schranke für die Basisfälle $T(0)$ und $T(1)$ darstellen:

$T(0) = 1 \leq (2^1 - 1) \cdot 7 = 7$

$T(1) = 7 \leq (2^{3/4+1} - 1) \cdot 7 \approx 16{,}5$

q.e.d.

Aufgabe 4 (Mastertheorem): (3 + 2 + 2 + 2 Punkte)

Geben Sie eine möglichst gute untere und obere Schranke für nachfolgende Rekursionsgleichungen an. Verwenden Sie dafür das Mastertheorem und begründen Sie Ihre Antwort.

a) $T(n) = 2T(\sqrt{n}) + \mathrm{ld}n$

b) $T(n) = 4T(\frac{n}{5}) + n \cdot \log n$

c) $T(n) = 8T(\lceil \frac{n}{2} \rceil) + 16n$

d) $T(n) = 47T(\lfloor \frac{n}{47} \rfloor) + 47n$

Lösung:

a) $T(n) = 2T(\sqrt{n}) + \mathrm{ld}n$

Damit wir das Mastertheorem anwenden können führen wir zuerst folgende Substitution durch:

$$m := \mathrm{ld}n \quad \Rightarrow \quad n = 2^m$$

Nun schreiben wir die Formel wie folgt um:

$$\begin{aligned} T(n) &= 2T(\sqrt{2^m}) + m \\ &= 2T(2^{\frac{m}{2}}) + m \qquad | \; S(m) = T(2^m) \\ S(m) &= 2S(\frac{m}{2}) + m \end{aligned}$$

Die Koeffizienten für das Mastertheorem sind nun: $b = 2$, $c = 2$ und $f(m) = m$. Somit erhalten wir $E = \log_2 2 = 1$. Da $m^E = m$ und $f(m) \in \Theta(m)$ wenden wir den 2. Fall des Mastertheorems an und erhalten: $S(m) \in \mathcal{O}(m \cdot \mathrm{ld}m)$. Setzen wir dieses Ergebnis nun in $T(n)$ ein, so erhalten wir die Lösung:

$$T(n) \in \Theta(\mathrm{ld}(n) \cdot \mathrm{ld}(\mathrm{ld}(n)))$$

q.e.d.

b) $T(n) = 4T(\frac{n}{5}) + n \cdot \log n$

Es gilt $b = 4$, $c = 5$ und somit ist $E = log_5 4 \leq 1$. Da $f(n) \in \mathcal{O}(n \cdot \log n)$ ist folgt nach dem 3. Fall des Mastertheorems:

$$T(n) \in \Theta(n \cdot \log n)$$

q.e.d.

c) $8T(\lceil \frac{n}{2} \rceil) + 16n$

Es gilt $b = 8$, $c = 2$ und somit ist $E = log_2 8 = 3$. Da $f(n) \in \mathcal{O}(n^3)$ wenden wir den 1. Fall des Mastertheorem an und erhalten:

$$T(n) \in \Theta(n^3)$$

q.e.d.

d) $T(n) = 47T(\lfloor \frac{n}{47} \rfloor) + 47n$.

Die Koeffizienten des Mastertheorems sind: $b = 47$, $c = 47$ und somit ist $E = \log_{47} 47 = 1$ und $f(n) \in \Theta(n)$. Der 2. Fall des Mastertheorem liefert dann:

$$T(n) \in \Theta(n \cdot \log n)$$

q.e.d.

$\square$.

Prof. Dr. Ir. Joost-Pieter Katoen

Jonathan Heinen, Sabrina von Styp

Hinweise:

- Die Übungsblätter sollen in Gruppen von je 3 Studierenden aus der gleichen Kleingruppenübung bearbeitet werden.
- Die Lösungen müssen bis Montag, den 31. Mai um 11:00 Uhr in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55).
- Namen und Matrikelnummern der Studenten sowie die Nummer der Übungsgruppe sind auf jedes Blatt der Abgabe zu schreiben. Heften bzw. tackern Sie die Blätter!

Aufgabe 1 (Min- und Max-Heaps): (4+4 Punkte)

a) Bestimmen Sie, ob folgende Arrays Heaps (Max-Heaps) sind. Falls nicht, geben Sie an wo die Heapeigenschaft verletzt ist.

1.)

56	47	56	10	20	50	51	1	2	3	18

2.)

56	56	47	10	20	50	51	1	2	3	18

3.)

56	47	56	10	51	20	50	1	2	3	18

4.)

56	47	56	10	5	20	50	1	2	3	18

b) In der Vorlesung wurden so genannte Max-Heaps vorgestellt. D.h jedes Element ist größer/gleich seiner Kinder. Ein *Min-Heap* ist ein Heap bei dem jedes Element kleiner/gleich seiner Kinderelemente ist.
Bestimmen sie, ob die folgenden Arrays Min-Heaps sind, und falls nicht, geben sie an, wo die Heapeigenschaft verletzt ist

1.)

1	5	1	8	10	4	15	11	12	14	11

2.)

0	1	5	8	10	4	15	11	12	14	11

3.)

1	5	1	11	8	4	15	11	12	14	10

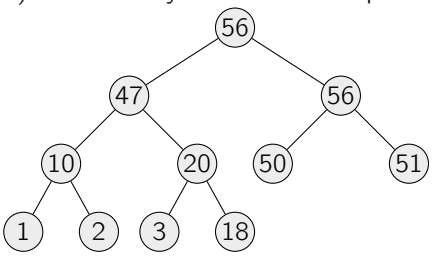
4.)

1	5	1	11	15	4	8	11	12	14	10

Lösung:

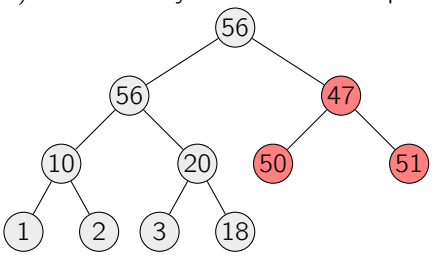
a) Die folgenden Heaps sind in den gegebenen Arrays repräsentiert:

1) Dieses Array ist ein Max-Heap.



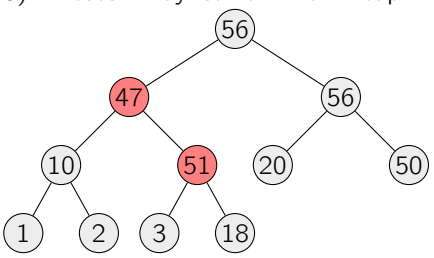
56	47	56	10	20	50	51	1	2	3	18

2) Dieses Array ist kein Max-Heap.



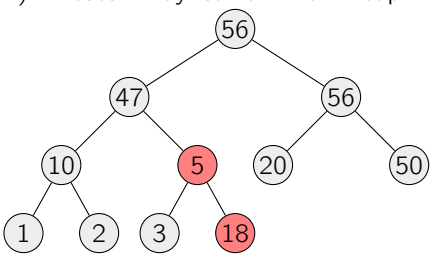
56	56	47	10	20	50	51	1	2	3	18

3) Dieses Array ist kein Max-Heap.



56	47	56	10	51	20	50	1	2	3	18

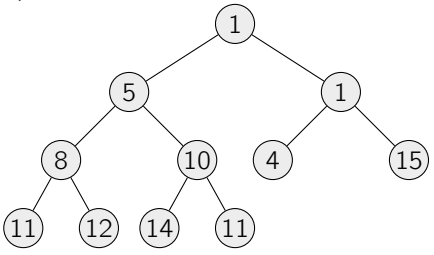
4) Dieses Array ist kein Max-Heap.



56	47	56	10	5	20	50	1	2	3	18

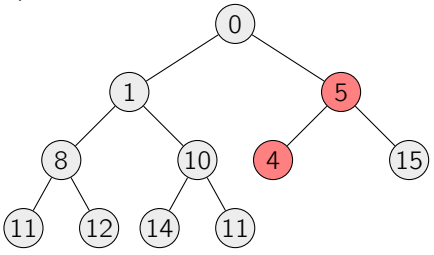
b) Die folgenden Heaps sind in den gegebenen Arrays repräsentiert:

1) Dieses Array ist ein Min-Heap.



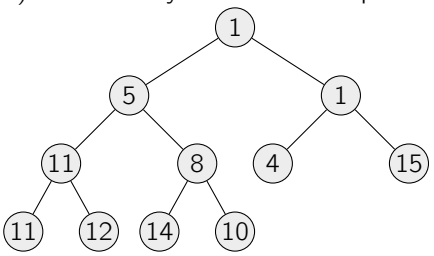
1	5	1	8	10	4	15	11	12	14	11

2) Dieses Array ist kein Min-Heap.



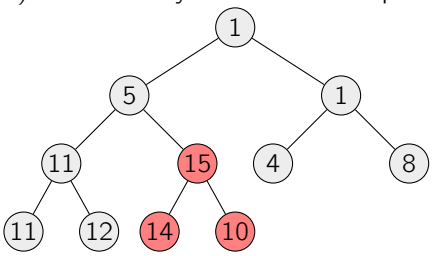
0	1	5	8	10	4	15	11	12	14	11

3) Dieses Array ist ein Min-Heap.



1	5	1	11	8	4	15	11	12	14	10

4) Dieses Array ist kein Min-Heap.



1	5	1	11	15	4	8	11	12	14	10

Aufgabe 2 (Insertionsort): (5 Punkte)

Nutzen Sie Insertionsort, um die Schlüsselwerte des folgenden Arrays aufsteigend zu sortieren. Geben Sie den Zustand des Arrays nach jeder Einfügeoperation an, sowie die Anzahl der Schlüsselwert-Vergleiche und Array-Zuweisungen, die für den Sortierschritt benötigt wurden.

1	**8**	**5**	**9**	**0**	**2**	**3**	**4**	**7**	**6**

Lösung:

Array zu Beginn:

1	**8**	**5**	**9**	**0**	**2**	**3**	**4**	**7**	**6**

1. Schritt: Ein Vergleich und eine Zuweisung:

1	**8**	**5**	**9**	**0**	**2**	**3**	**4**	**7**	**6**

2. Schritt: Zwei Vergleiche und zwei Zuweisungen:

1	**5**	**8**	**9**	**0**	**2**	**3**	**4**	**7**	**6**

3. Schritt: Ein Vergleich und eine Zuweisung:

1	**5**	**8**	**9**	**0**	**2**	**3**	**4**	**7**	**6**

4. Schritt: Vier Vergleiche und fünf Zuweisungen:

0	**1**	**5**	**8**	**9**	**2**	**3**	**4**	**7**	**6**

5. Schritt: Vier Vergleiche und vier Zuweisungen:

0	**1**	**2**	**5**	**8**	**9**	**3**	**4**	**7**	**6**

6. Schritt: Vier Vergleiche und vier Zuweisungen:

0	1	2	3	5	8	9	4	7	6

7. Schritt: Vier Vergleiche und vier Zuweisungen:

0	1	2	3	4	5	8	9	7	6

8. Schritt: Drei Vergleiche und drei Zuweisungen:

0	1	2	3	4	5	7	8	9	6

9. Schritt: Vier Vergleiche und vier Zuweisungen:

0	1	2	3	4	5	6	7	8	9

Aufgabe 3 (Heapsort): (5 + 8 Punkte)

Das folgende Array soll mit Hilfe des Heapsort-Algorithmus sortiert werden:

5	3	4	7	0	2	3	6	1	8

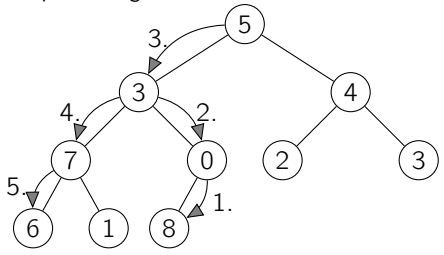
a) Stellen Sie das gegeben Array als Baum dar und skizzieren Sie (durch nummerierte Pfeile) die nötige Versickerungen. Geben Sie den resultierenden Heap sowohl als Baum als auch als Array an.

b) Nutzen Sie Heapsort um, ausgehend von dem Heap aus Aufgabenteil **a)**, das gegebene Array zu sortieren. Geben Sie für jeden Sortierschritt den neuentstandenen Heap als Array sowie als Baum an und skizzieren Sie nötige Versickerungen in der Baumdarstellung. Geben Sie auch die Reihenfolge der Versickerungen an.

Lösung:

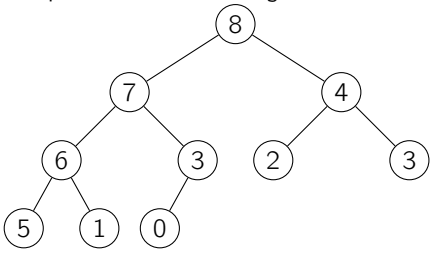
a) Das Array zu beginn (mit den nötigen Versickerungsschritten) sowie das resultierende Array mit Heapeigenschaft:

Heap zu Beginn:



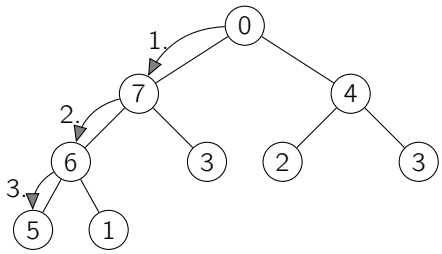
5	3	4	7	0	2	3	6	1	8

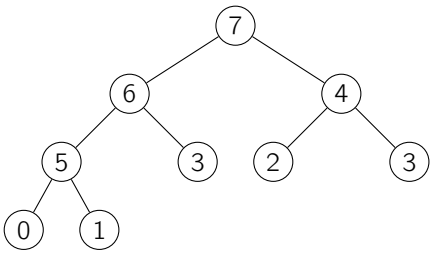
Heap nach Initialisierung:



8	7	4	6	3	2	3	5	1	0

1. Sortierschritt

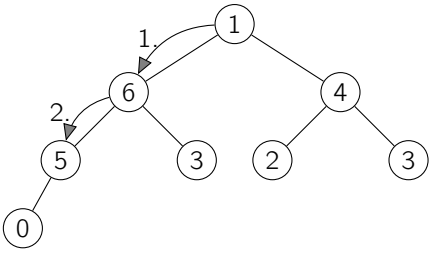


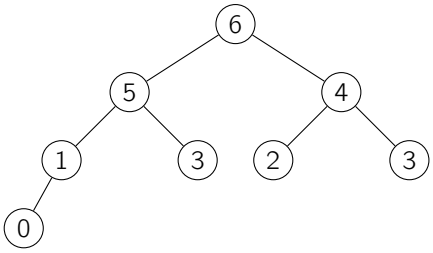


0	7	4	6	3	2	3	5	1	8

7	6	4	5	3	2	3	0	1	8

2. Sortierschritt

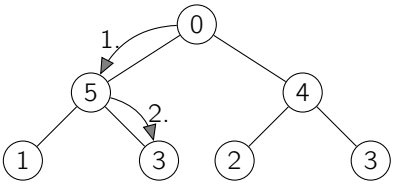


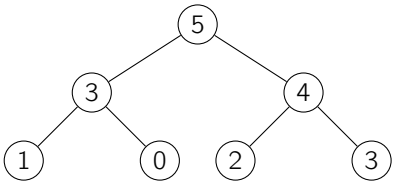


1	6	4	5	3	2	3	0	7	8

6	5	4	1	3	2	3	0	7	8

3. Sortierschritt

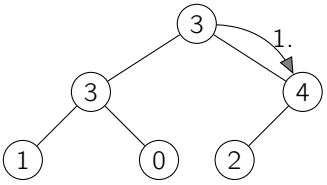


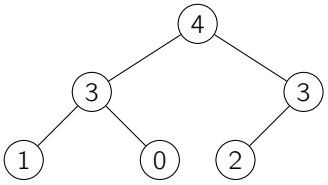


0	5	4	1	3	2	3	6	7	8

5	3	4	1	0	2	3	6	7	8

4. Sortierschritt

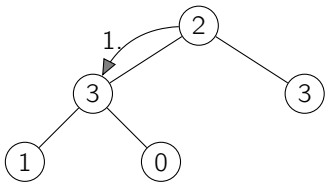


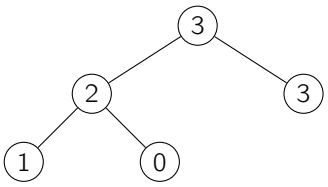


3	3	4	1	0	2	5	6	7	8

4	3	3	1	0	2	5	6	7	8

5. Sortierschritt

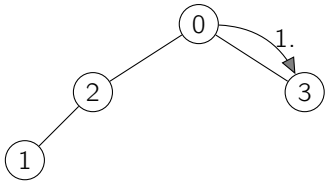


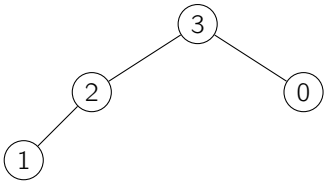


2	3	3	1	0	4	5	6	7	8

3	2	3	1	0	4	5	6	7	8

6. Sortierschritt

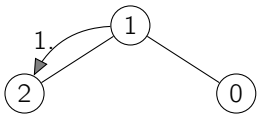


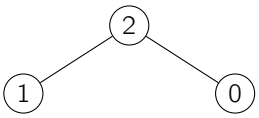


0	2	3	1	3	4	5	6	7	8

3	2	0	1	3	4	5	6	7	8

7. Sortierschritt

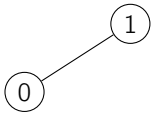


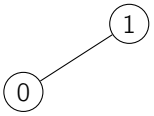


1	2	0	3	3	4	5	6	7	8

2	1	0	3	3	4	5	6	7	8

8. Sortierschritt





0	1	2	3	3	4	5	6	7	8

1	0	2	3	3	4	5	6	7	8

Endresultat

Aufgabe 4 (Ein weiterer Sortieralgorithmus): (2+2+5+5 Punkte)

Gegeben sei der folgende Sortieralgorithmus:

```
void sort(int E[]) {
  int i,j,m;
  for (i = 0; i < E.length; i++) {
    m = i;
    for (j = i + 1; j < E.length; j++) {
      if(E[j] <= E[m]){
          m = j;
      }
    }
    int v = E[i];
    E[i] = E[m];
    E[m] = v;
  }
}
```

a) Geben Sie in wenigen Worten wieder, wie der gegebene Algorithmus funktioniert. In der Vorlesung wurden mehrere Sortierverfahren genannt (siehe Folien). Welcher Name passt zu diesem Algorithmus?

b) Ist der Sortieralgorithmus stabil, oder kann er so angepasst werden, dass er stabil wird?

c) Nutzen Sie den gegebene Algorithmus, um das folgende Array zu sortieren. Geben Sie den Zustand des Arrays nach jedem Durchlauf der äußeren Schleife an.

1	3	2	7	0	4	8	5	7	6

d) Welche Average-Case Laufzeit besitzt der gegebene Sortieralgorithmus für eine Eingabe der Länge n? Geben Sie die Komplexitätsklasse $\Theta(T_{sort(E)})$ für Array E mit Länge `n = E.length` an und begründen Sie Ihre Antwort.

Lösung:

a) Der Algorithmus sortiert das Array von vorne nach hinten indem er jeweils das kleinste Element aus dem noch zu sortierenden Teil sucht und dieses mit dem ersten Element dieses Bereiches tauscht. Dadurch wird von vorne nach Hinten ein sortiertes Array aufgebaut.

Selectionsort ist ein geeigneter Name für dieses Verfahren, da jeweils das **kleinste** Element aus den übrigen Elementen **ausgesucht** wird.

b) Der Algorithmus ist nicht stabil. Um einen stabilen Algorithmus zu erhalten können wir das Vertauschen der Elemente durch ein Einfügen ersetzen, wie wir es von Insertionsort kennen.

c) In den folgenden Schritten sortiert der Algorithmus das Array:

1	3	2	7	0	4	8	5	7	6
0	3	2	7	1	4	8	5	7	6
0	1	2	7	3	4	8	5	7	6
0	1	2	7	3	4	8	5	7	6
0	1	2	3	7	4	8	5	7	6
0	1	2	3	4	7	8	5	7	6
0	1	2	3	4	5	8	7	7	6
0	1	2	3	4	5	6	7	7	8
0	1	2	3	4	5	6	7	7	8
0	1	2	3	4	5	6	7	7	8

d) Unabhängig von den jeweiligen Schlüsseln und ihrer Verteilung im Array wird die äußere Schleife n-fach durchlaufen. Die innere Schleife jedes mal von der aktuellen Position bis zum Ende. Das Vertauschen der Werte geschieht in konstanter Zeit, da bei diesem Algorithmus das Aufschieben der Elemente entfällt. Es ergibt sich eine Laufzeitkomplexität von:

$$T_{sort(E)} \in \Theta(\sum_{i=0}^{n} i) = \Theta(n^2)$$

Prof. Dr. Ir. Joost-Pieter Katoen

Jonathan Heinen, Sabrina von Styp

Hinweise:

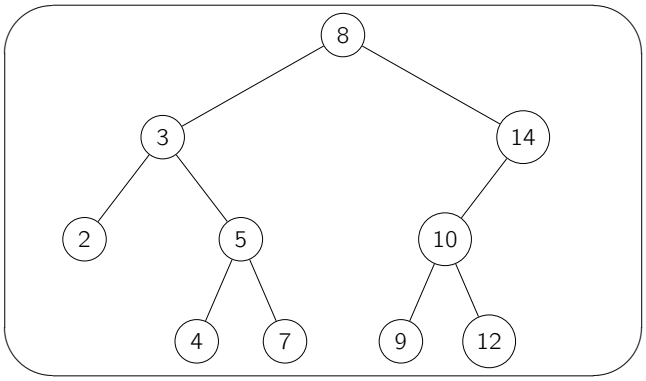
- Die Übungsblätter sollen in Gruppen von je 3 Studierenden aus der gleichen Kleingruppenübung bearbeitet werden.
- Die Lösungen müssen bis Montag, den 7. Juni um 11:00 Uhr in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55).
- Namen und Matrikelnummern der Studenten sowie die Nummer der Übungsgruppe sind auf jedes Blatt der Abgabe zu schreiben. Heften bzw. tackern Sie die Blätter!
- Am Dienstag den 1. Juni findet, wegen dem RWTH SPORTS DAY, keine Vorlesung statt. Für die Übungsgruppen wird es Ausweichtermine geben.

Aufgabe 1 (Binärer Suchbaum): (3 + 4 + 4 + 6 + 5 Punkte)

a) Geben Sie den binären Suchbaum an, der entsteht, wenn die folgenden Zahlen in *gegebener Reihenfolge* in einen *leeren Suchbaum* einfügt werden:

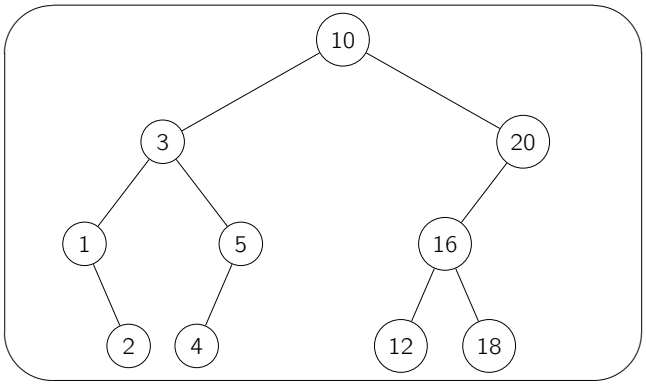
5 2 3 8 6 7 10 4

b) Gegeben sei der folgende binäre Suchbaum:

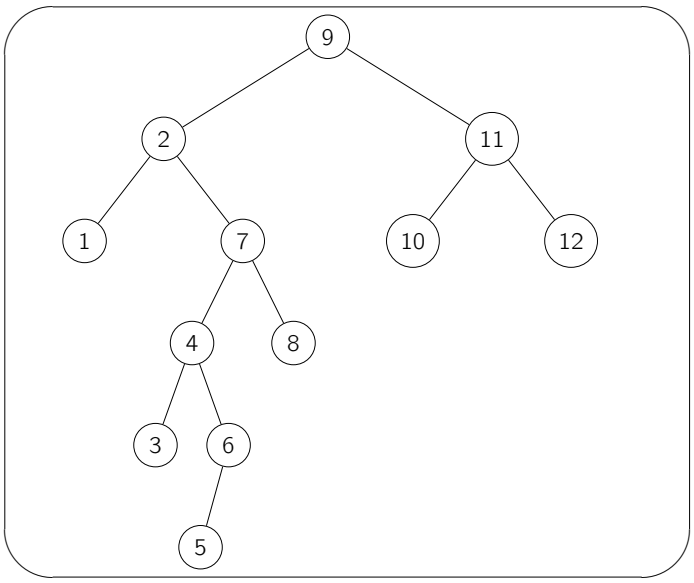


Geben Sie die binären Suchbäume an, die entstehen, wenn sie erst **14**, dann **3** und schließlich die **12**, entsprechend dem in der Vorlesung vorgestellten Verfahren, aus dem Baum heraus löschen.

c) Geben Sie für den folgenden binären Suchbaum die Binärbäume an, die entstehen, wenn eine **Rechtsrotation** auf den Knoten mit dem Wert **20** ausgeführt wird, dann eine **Linksrotation** auf den Knoten mit Wert **3** und zuletzt eine **Linksrotation auf die Wurzel** des entstandenen Baumes:



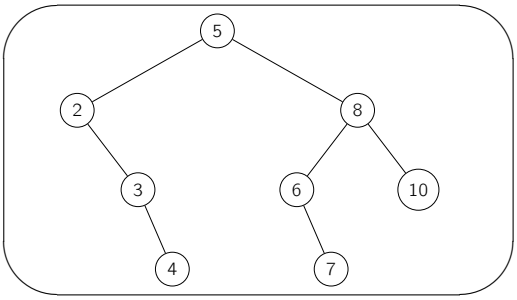
d) Geben Sie **maximal vier Rotationen** an, die den folgenden Baum der **Höhe fünf** in einen Baum der **Höhe drei** transformieren. Geben Sie auch den Zustand des Baumes nach jeder Rotation an.



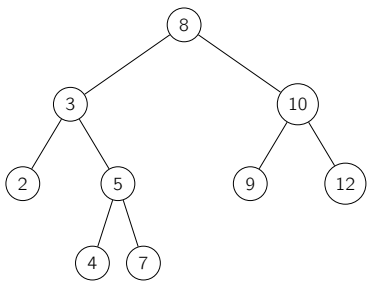
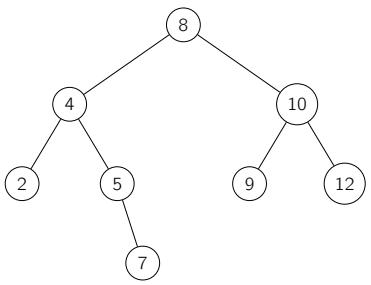
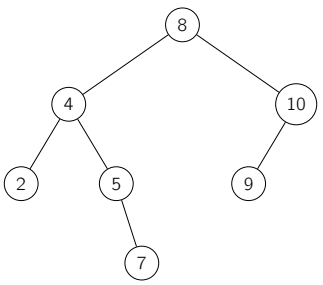
e) In der Vorlesung wurde ein Verfahren zur *Bestimmung des Nachfolgers* eines Elementes im binären Suchbaum vorgestellt. Hierfür wird der Pfad zwischen den beiden Elementen zurückgelegt. Wie lang ist dieser Pfad maximal für einen Baum mit n Elementen? Beweisen Sie Ihre Aussage.

Lösung:

a) Wir fügen **5 2 3 8 6 7 10 4** in einen leeren Suchbaum ein und erhalten:

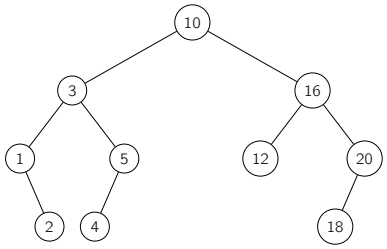


b) Es ergeben sich die folgenden Bäume:

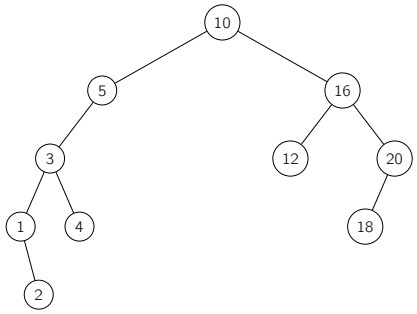
Löschen des Schlüssels 14:	Löschen des Schlüssels 3:	Löschen des Schlüssels 12
		

c) Es ergeben sich folgende Bäume durch Rotation:

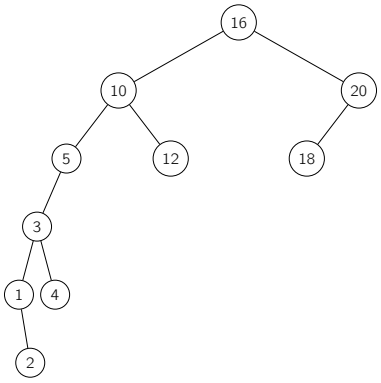
Rechtsrotation auf Knoten 20:



Linksrotation auf Knoten 3:

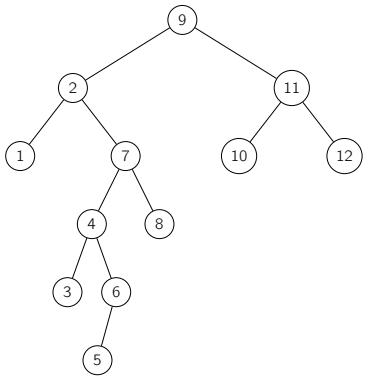


Linksrotation auf Wurzel:

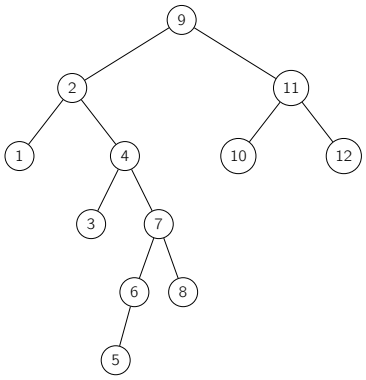


d) Wir können den gegebenen Baum mit vier Rotationen in einen Baum der Höhe drei transformieren. Hierzu führen wir zuerst eine Rechtsrotation auf den Knoten 7 aus und anschließend eine Linksrotation auf den Knoten 2. Durch diese beiden Rotationen erhalten wir einen Baum der Höhe vier:

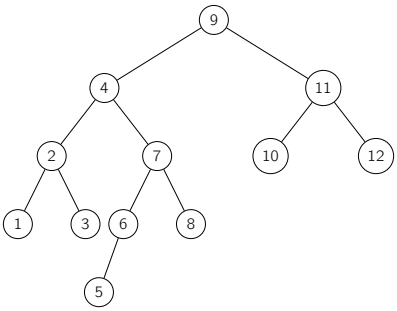
Ursprünglicher Baum:



Rechtsrotation auf Knoten 7:

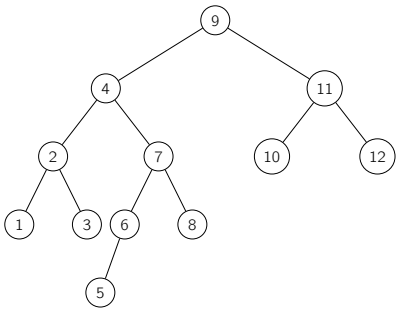


Linksrotation auf Knoten 2

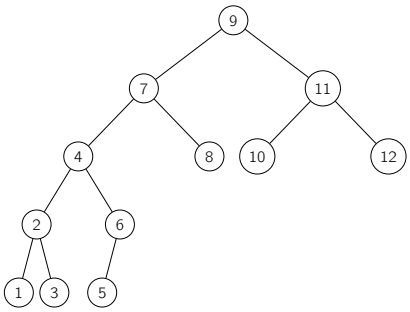


Auf diesen Baum führen wir nun noch eine Linksrotation auf den Knoten 4 durch, gefolgt von einer Rechtsrotation um die Wurzel und erhalten so einen Baum der Höhe drei:

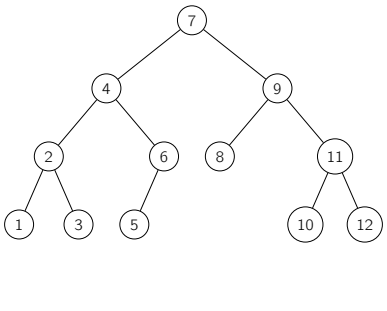
Baum der Höhe vier:



Linksrotation auf Knoten 4:

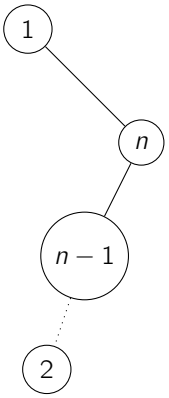


Rechtsrotation auf Wurzel:



e) *Behauptung:* Der längsten Pfad mögliche Pfad in einem Baum mit n Elementen besitzt die Länge $n-1$.

Da in einem Pfad kein Knoten doppelt vorkommen darf, umfasst jeder Pfad in Einem Baum mit n Elementen maximal n Knoten und besitzt somit maximal eine Länge von $n-1$.
Im folgenden Baum hat der Pfad zwischen der Wurzel und ihrem Nachfolger die Länge $n-1$:



Aufgabe 2 (Quicksort): (3+3+4 Punkte)

a) Sortieren Sie das folgende Array mit Hilfe von *Quicksort*. Skizzieren Sie den Zustand des Arrays jeweils nach der Wahl eines neuen *Pivotelements* und geben Sie dieses an.

3	7	1	5	8	2	4	6

b) Zeigen Sie, anhand eines geeigneten Beispiels, dass die in der Vorlesung vorgestellte *Implementierung von Quicksort nicht stabil* ist.

Geben Sie eine geeignete Eingabe an, machen Sie gleiche Elemente durch geeignete Markierung unterscheidbar und führen Sie dann Quicksort auf der gewählten Eingabe aus, wobei Sie in jedem Schritt den Zustand der Folge sowie das Pivotelement angeben.

c) Modifizieren Sie die in der Vorlesung vorgestellte *Implementierung für Quicksort* so, dass sie *stabil* ist. Die *Komplexitätsklasse* der Laufzeit soll sich nicht verändern.

Hilfestellung: Möchte man, mit Hilfe eines *nicht stabilen* Sortieralgorithmus, *stabil* sortieren, so gibt es die Möglichkeit die ursprüngliche Position der Elemente als *sekundäres Sortierkriterium* hinzuziehen. Werden dann zwei Elemente mit gleichem *Primärschlüssel* verglichen wird die Sortierung anhand des *sekundären Sortierkriteriums* - also der ursprünglichen Position im Array - vorgenommen.

Lösung:

a) Die Sortierung des Arrays erfolgt in den folgenden Schritten:

3	7	1	5	8	2	4	6
3	4	1	5	2	6	7	8
1	2	3	5	4	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

b) Folgendes Beispiel zeigt das Quicksort nicht stabil ist:

8	5	1'	1''	4
1''	1'	4	8	5
1''	1'	4	8	5
1''	1'	4	5	8

c) Der folgende Algorithmus ermöglicht die stabile Sortierung mit Quicksort. Er nutzt den Hinweis des Aufgabenblattes und verwende ein zweites Array P, um in P[i] die ursprüngliche Position von Element E[i] zu speichern.

```
int partition(int E[], int P[], int left, int right){

    // Waehle einfaches Pivotelement
    int ppos = right, pivot = E[ppos];
    while (true){

        // Bidirektionale Suche
        while ( left < right && (E[left] < pivot
             || (E[left] == pivot && P[left] < P[ppos])))
          left++;

        while (left < right && (E[right] > pivot
             || (E[left] == pivot && P[left] >= P[ppos]))
          right--;

        if (left >= right){
          break;
        }

        swap(E[left], E[right]);
        swap(P[left], P[right]);
    }

    swap(E[left], E[ppos]);
    swap(P[left], P[ppos]);

    // gib Splitpunkt zurueck
    return left;
}
```

```
void quickSort(int E[], int P[], int left, int right){

    if (left < right){

        // i ist Position des Split-punktes (Pivot)
        int i = partition(E, P, left, right);
        // sortiere den linken Teil
        quickSort(E, P, left, i - 1);
        // sortiere den rechten Teil
        quickSort(E, P, i + 1, right);
    }
}

void quickSort(int E[]){

    int P[E.length];
    for(int i=0; i<P.length; ++i)
        P[i] = i;

    quickSort(E,P,0,E.length-1);
}
```

Aufgabe 3 (Höhergradige Heaps): (3+2+4+3 Punkte)

In Übung 5 haben Sie Heapsort, basierend auf *binären Heaps*, zum Sortieren genutzt. In binären Heaps hat jedes Element bis zu zwei Kinderelemente. Das Konzept der binären Heaps lässt sich auf *d-Heaps* erweitern. Ein *d*-Heap ist ein Baum mit Verzweigungsgrad d (d.h. jedes Element besitzt bis zu d Kinder), der die *max-Heapeigenschaft* (alle Kinderelemente haben *niedrigere* Werte) erfüllt.

a) Geben Sie die Formel an mit der, für die Arraydarstellung eines *d*-Heaps, die Position der Kinder des Elementes an Position i bestimmt werden kann.

In welchem Bereich des Arrays ist die Heapeigenschaft immer erfüllt?

b) Stellen Sie den im folgenden Array repräsentierten *3-Heap* als Baum dar.

40	32	24	36	28	30	5	12	21

c) Geben Sie eine modifizierte Version des in der Vorlesung vorgestellten Algorithmus `sink` an, der Elemente in einem *d*-Heap versickern lässt. Der Grad d wird hierbei als Parameter übergeben.

d) Sortieren Sie das in *b)* gegebene Array entsprechend der *Heapsortmethode für 3-Heaps*.

Lösung:

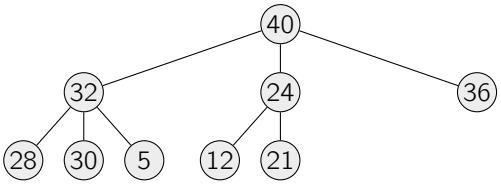
a) Sei `a` das gegebene Array mit Wurzel des Heaps in `a[0]`. Das j-te Kind ($j \in \{1, \ldots, d\}$) von `a[i]` liegt dann in `a[d * i + j]`.

Die Position des erste Kind eines Knotens liegt genau dann außerhalb des zulässigen Bereichs ($\geq n$), wenn der Knoten ein Blatt ist. Somit gilt für alle Positionen i an denen Blätter stehen:

$$\begin{array}{rrcl} & d \cdot i + 1 & \geq & n \\ \Leftrightarrow & d \cdot i & \geq & n - 1 \\ \Leftrightarrow & i & \geq & \frac{n-1}{d} \end{array}$$

Da i eine ganzzahliger Wert ist gilt für die Position i jedes Blattes, dass $i \geq \lceil \frac{n-1}{d} \rceil$ und somit sind alle Knoten ab Position $\lceil \frac{n-1}{d} \rceil$ Blätter und das Array erfüllt ab Position $\lceil \frac{n-1}{d} \rceil$ die Heapeigenschaft.

b) Es ergibt sich der folgende 3-Heap, der offensichtlich ein max-Heap ist:



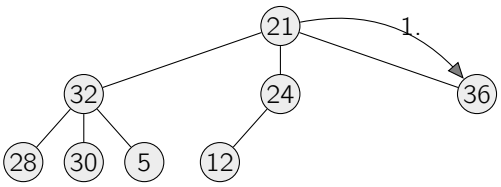
c) Mit Hilfe des folgenden Algorithmus können Versickerungen in einem d-Heap realisiert werden:

```
void sink(int E[], int n, int pos, int d) {
  int firstChild = d * pos + 1;
  while (firstChild < n) {
    // finde das größte Kind
    int max = firstChild;
    for(int i = 1; i < d; i++){
      if (firstChild + i < n && E[firstChild + i] > E[max]) {
        max = firstChild + i;
      }
    }
    // überpruefen, ob keine Verletzung vorliegt
    if (E[pos] > E[max]) {
      break;
    }
    // austauschen der Elemente - versickern
    swap(E[pos], E[max]);
    // im Baum absteigen
    pos = max;
    firstChild = d * pos + 1;
  }
}
```

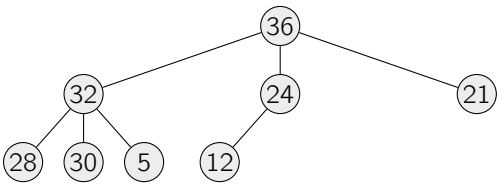
d) Da es sich, wie bereits in **b)** festgestellt, um einen max-Heap handelt, ist eine Initialisierung nicht nötig.

Sortierphase:

1. Sortierschritt

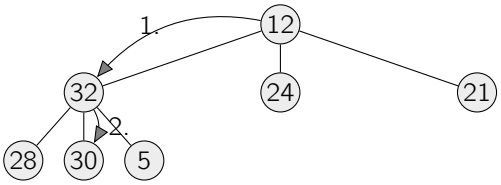


21	32	24	36	28	30	5	12	40

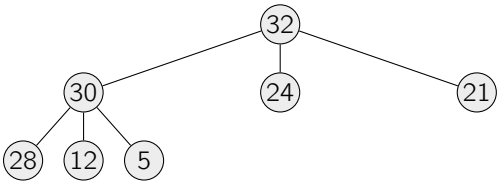


36	32	24	21	28	30	5	12	40

2. Sortierschritt

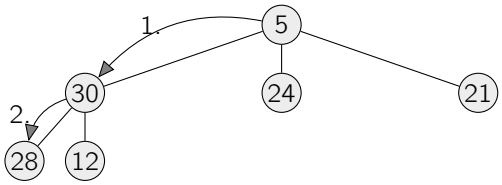


12	32	24	21	28	30	5	36	40

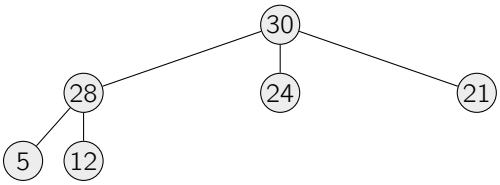


32	30	24	21	28	12	5	36	40

3. Sortierschritt

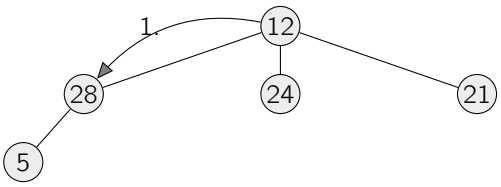


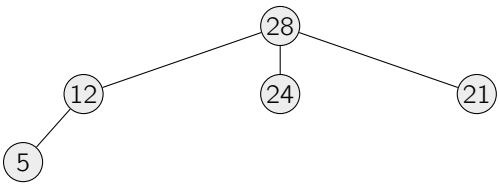
5	30	24	21	28	12	32	36	40



30	28	24	21	5	12	32	36	40

4. Sortierschritt

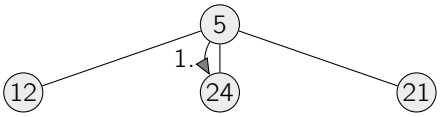


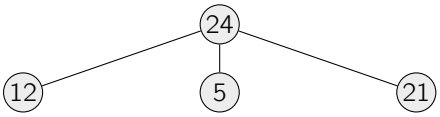


12	28	24	21	5	30	32	36	40

28	12	24	21	5	30	32	36	40

5. Sortierschritt

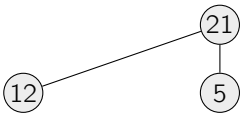


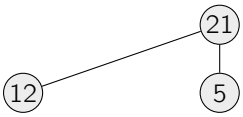


5	12	24	21	28	30	32	36	40

24	12	5	21	28	30	32	36	40

6. Sortierschritt

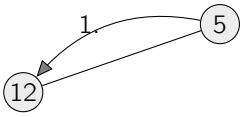


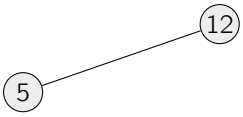


21	12	5	24	28	30	32	36	40

21	12	5	24	28	30	32	36	40

7. Sortierschritt

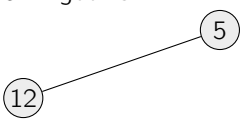




5	12	21	24	28	30	32	36	40

12	5	21	24	28	30	32	36	40

8. Ergebnis:



5	12	21	24	28	30	32	36	40

5	12	21	24	28	30	32	36	40

Aufgabe 4 (Optimales Sortierverfahren): (3 Punkte)

Bei der Kommunikation über Netzwerke kann nicht immer sichergestellt werden, dass die einzelnen Pakete beim Empfänger in der gleichen Reihenfolge ankommen wie sie versendet wurden. Um dadurch entstehende Probleme zu lösen, wird jedes Paket mit einer eindeutigen, aufsteigenden ID versehen. Eine Veränderung der Reihenfolge ist jedoch nicht der Regelfall, sondern eher die Ausnahme. Um die ursprüngliche Reihenfolge der Pakete wieder herzustellen werden diese, nach dem Empfangen aller Pakete, auf der Empfängerseite sortiert. Welcher Sortieralgorithmus ist für diese Aufgabe optimal geeignet? Begründen Sie Ihre Antwort.

Lösung:

Wir gehen davon aus, dass die Folge der Pakete nahezu sortiert ist und falsch positionierte Pakete nicht weit von ihrer richtigen Position entfernt sind. Unter diesen Voraussetzungen empfiehlt es sich Insertionsort zu nutzen. Insertionsort hat auf nahezu sortierte Folgen eine nahezu lineare Laufzeitkomplexität. Quicksort, wie wir ihn kennengelernt haben, hat hingegen auf sortierte Folgen eine quadratische Laufzeit (Worst-Case).

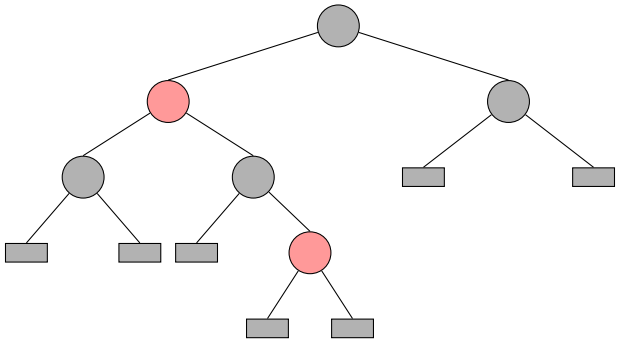
Prof. Dr. Ir. Joost-Pieter Katoen

Jonathan Heinen, Sabrina von Styp

Hinweise:

- Die Übungsblätter sollen in Gruppen von je 3 Studierenden aus der gleichen Kleingruppenübung bearbeitet werden.
- Die Lösungen müssen bis Montag, den 14. Juni um 11:00 Uhr in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55).
- Namen und Matrikelnummern der Studierenden sowie die Nummer der Übungsgruppe sind auf jedes Blatt der Abgabe zu schreiben. Heften bzw. tackern Sie die Blätter!
- Am Freitag den 18. Juni findet, keine Vorlesung statt, da der Audimax fremd belegt ist.

In dieser Übung werden zwei Höhenbegriffe genutzt. Einerseits die Schwarzhöhe und anderseits die Höhe eines Baumes. Für den folgenden Rot-Schwarz-Baum (in dem die Schlüsselwerte nicht angegeben sind) ist die Schwarzhöhe zwei (vergleiche Definition in der Vorlesung) und die Höhe vier (Längster Pfad von der Wurzel bis zu einem (externen) Blatt).

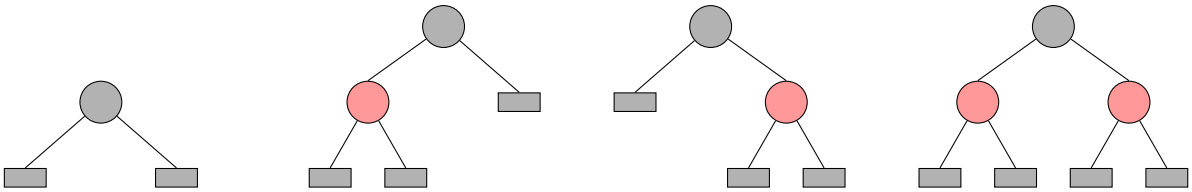


Aufgabe 1 (Rot-Schwarz-Bäume): (8 Punkte)

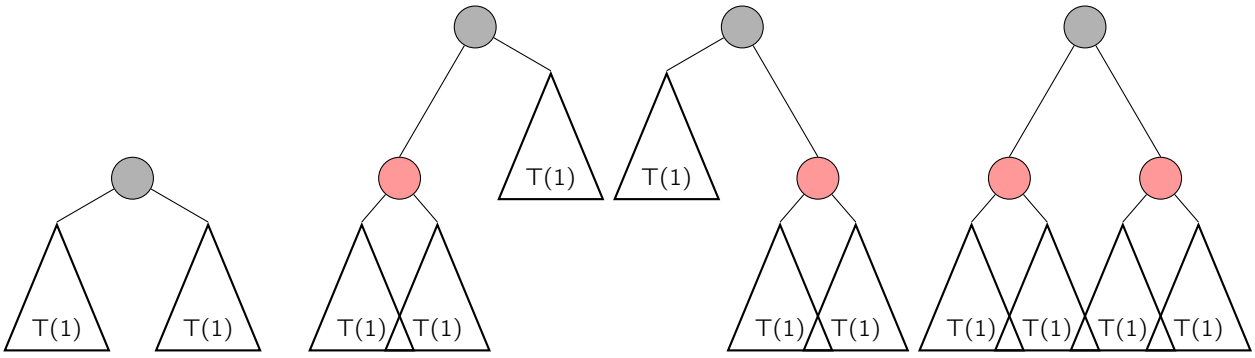
Bestimmen Sie die Anzahl verschiedener Rot-Schwarz-Bäume mit Schwarzhöhe *zwei*, wenn die Schlüsselwerte ignoriert werden. D.h. lediglich die Farbe eines Knotens und seine Position im Baum sind entscheidend. Begründen Sie Ihre Antwort.

Lösung:

Es gibt 4 Rot-Schwarz-Bäume der Höhe 1:



Wir können nun wie folgt Bäume der Schwarzhöhe *zwei* konstruieren (hierbei ist T(1) ein Rot-Schwarz-Baum der Höhe eins):



Für die ersten Konstruktionsmöglichkeit gibt es offensichtlich $4^2 = 16$ mögliche Belegungen, für die zweite $4^3 = 64$ und für die letzte Konstruktionsmöglichkeit $4^4 = 256$. Somit ergibt sich eine Gesamtzahl von $16+2\cdot64+256 = 400$ möglichen Rot-Schwarz-Bäumen mit Schwarzhöhe zwei.

Aufgabe 2 (Einfügen in einen Rot-Schwarz-Baum): (10+8 Punkte)

a) Die folgenden Werte sollen in einen leeren Rot-Schwarz-Baum eingefügt werden:

3, 2, 6, 13, 7

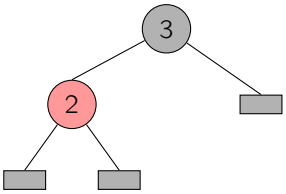
Zeichnen Sie den Baum jeweils nach dem Einfügen und nach jeder erfolgten Rotation und Umfärbung.

b) Zeigen Sie, dass die Schwarz-Höhe $bh(T)$ eines Rot-Schwarz-Baumes T bei jedem Einfügen eines Knotens gleich bleibt, oder sich um *genau eins* erhöht.

Lösung:

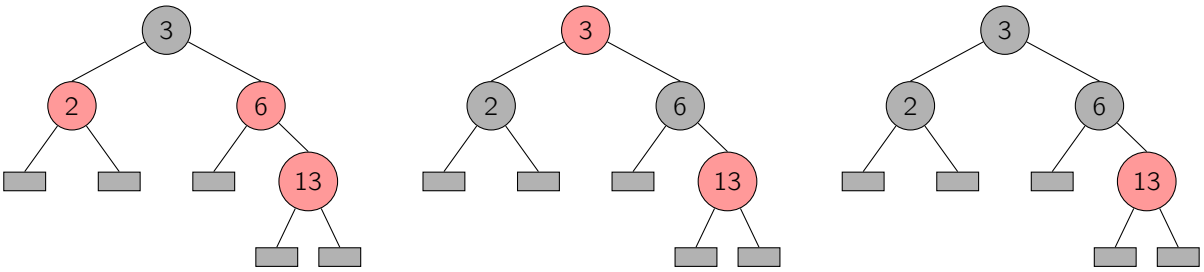
a) Einfügen von Schlüsselwert 3:
3 wird als neuer roter Knoten eingefügt. Da er die Wurzel ist wird er umgefärbt:

Einfügen von Schlüsselwert 2:
2 wird als neuer roter Knoten als linkes Kind der Wurzel eingefügt. Eine Umfärbung ist nicht nötig:

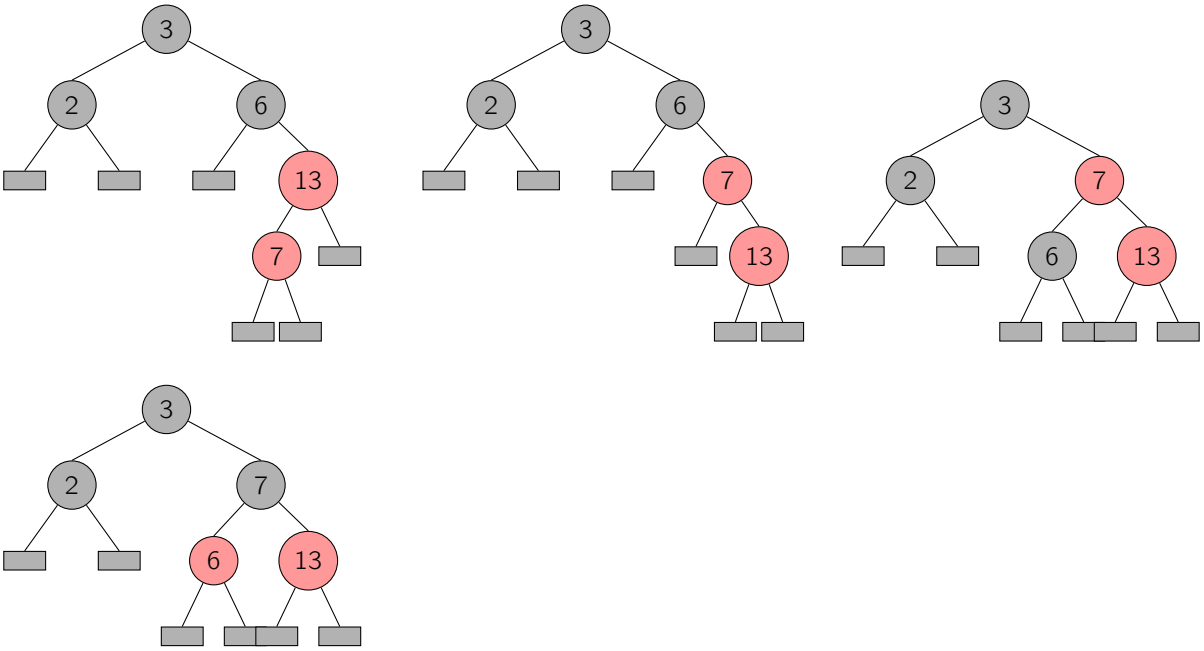


Ebenso kann der Schlüsselwert 6 eingefügt werden:

Durch das Einfügen des Schlüsselwert 13 erhalten wir zwei aufeinanderfolgende Rot-Knoten. Da der Onkel des neueingefügten Knoten Rot ist können Wir das Problem durch einfaches Umfärben lösen. In einem weiteren Schritt wird die Wurzel wieder schwarz gefärbt:



Nach dem Einfügen des Wertes 7 gibt es wieder aufeinaderfolgende Rot-Knoten. Durch eine erste Rechtsrotation überführen wir diesen Fall (2) erst in einen andern Fall (3). Danach können wir die neue Situation durch eine Linksrotation und Umfärbung auflösen:

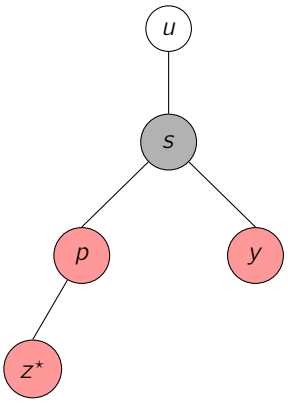


b) Der einzufügende Knoten wird zunächst wie bei einem gewöhnlichen binären Suchbaum eingefügt. Dann wird der Knoten rot gefärbt. Da ein roter Knoten eingefügt wurde, ändert sich die Schwarz-Höhe offensichtlich nicht.

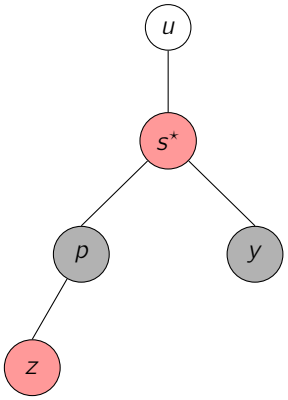
Anschließend wird RB-INSERT-FIXUP aufgerufen, um die Rot-Schwarz-Eigenschaften wiederherzustellen.

RB-INSERT-FIXUP durchläuft eine Schleife und unterscheidet innerhalb dieser 3 Fälle:

1. Fall: Der aktuell betrachtete (— der im 1. Schleifendurchlauf zu betrachtende Knoten ist hierbei der neu eingefügte —) und Eigenschaft-4-verletzende Knoten z ist rot, hat einen roten Vater p und einen roten Onkel y, (gezwungenermaßen) einen schwarzen Großvater s, und einen Urgroßvater unbekannter Farbe u. Der Urgroßvater hat vor der Behandlung des Falls die Schwarz-Höhe $bh'(u) = bh'(p) + 1 = bh'(y) + 1 = bh'(s) + 1$.



In diesem Fall werden der Vater p und der Onkel y schwarz gefärbt, sowie der Großvater s rot gefärbt. Als als nächstes zu betrachtender Knoten wird s bestimmt.



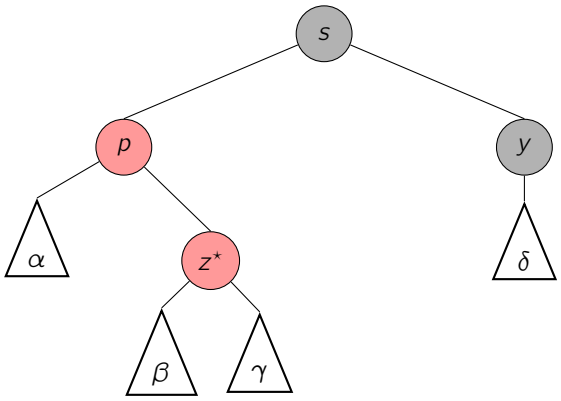
Die Schwarzhöhen nach der Behandlung des Falls sind:

$$\begin{aligned} bh(p) &= bh'(p) \\ bh(y) &= bh'(y) \\ bh(s) &= bh'(s) + 1 \\ bh(u) &= bh(s) \;=\; bh'(s) + 1 = bh'(u) \end{aligned}$$

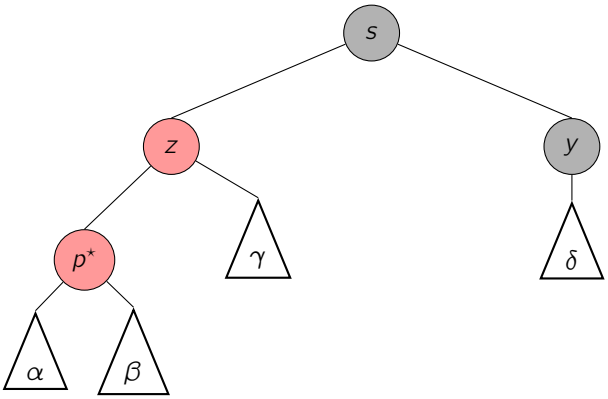
Der Urgroßvater hat nun nach der Behandlung des Falls also eine unveränderte Schwarz-Höhe.

Lediglich, wenn s die Wurzel von T ist (und s somit keinen Vater mit unveränderter Schwarz-Höhe besitzt), so hat sich die Schwarz-Höhe $bh(T) = bh(s)$ wegen $bh(s) = bh'(s) + 1$ um genau 1 erhöht.

2. Fall: Der aktuell betrachtete und Eigenschaft-4-verletzende Knoten z ist rot, ist ein rechtes Kind eines roten Vaters p, hat einen schwarzen Onkel y, und (gezwungenermaßen) einen schwarzen Großvater s.

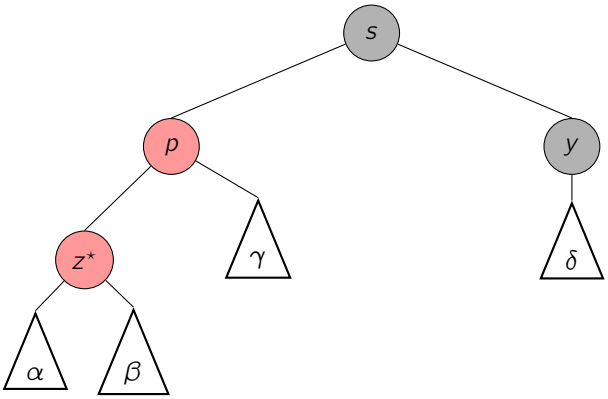


In diesem Fall wird der Vater p linksrotiert. Als nächster zu betrachtender Knoten wird p bestimmt. (Dies führt unweigerlich zu Fall 3 beim nächsten Schleifendurchlauf).



Es ändert sich offensichtlich keine der Schwarz-Höhen.

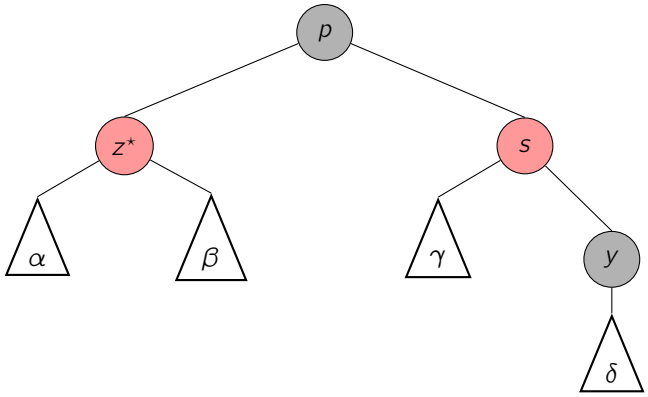
3. Fall: Der aktuell betrachtete und Eigenschaft-4-verletzende Knoten z ist rot, ist ein linkes Kind eines roten Vaters p, hat einen schwarzen Onkel y, und (gezwungenermaßen) einen schwarzen Großvater s. s ist die Wurzel des relavanten Teilbaums.



Die Schwarz-Höhen vor Behandlung des Falles $bh'(\cdot)$ sind

$$\begin{aligned} bh'(z) &= bh'(\alpha) = bh'(\beta) \\ bh'(p) &= bh'(z) = bh'(\gamma) \\ bh'(y) &= bh'(\delta) \\ bh'(s) &= bh'(p) = bh'(y) + 1, \text{ da } y \text{ schwarz} \end{aligned}$$

In diesem Fall wird der Großvater s rechtsrotiert, p schwarz gefärbt und s rot gefärbt. Ein als nächstes zu Betrachtender Knoten muss nicht neu bestimmt werden, da die Schleife hier immer terminiert.



Durch die Rotation ist p nun die Wurzel des relevanten Teilbaums. Wenn wir also zeigen wollen, dass sich die Schwarz-Höhe bei der Behandlung von Fall 3 nicht ändert, müssen wir zeigen, dass bh(p) = bh'(s).

Betrachten wir nun die Schwarz-Höhen nach Behandlung des Falls:

$$\begin{aligned} bh(z) &= bh'(z) \\ bh(y) &= bh'(y) \\ bh(s) &= bh(y) + 1 = bh'(y) + 1 = bh'(s) \\ bh(p) &= bh(s) = bh'(s) \end{aligned}$$

Die Schwarz-Höhe des relevanten Teilbaums ändert sich also nicht.

Nach Terminieren der Schleife wird die Wurzel T schwarz gefärbt. Dies ändert die Schwarz-Höhe von T nicht.

Es ändert sich also in keinem (außer genau einem) der Fälle die Höhe des gerade betrachteten Teilbaums. Da der Zeiger auf den als nächstes zu betrachtenden Knoten immer nach oben wandert, oder auf derselben Ebene bleibt(, die Schleife aber dann nach höchstens einem weiteren Schritt terminiert), können wir leicht induzieren, dass sich in keinem (außer genau einem) der Fälle die Schwarz-Höhe des gesamten Baums T ändert.

Der viel erwähnte *eine* Fall tritt nur genau dann auf, wenn in Fall 1 der Großvater die Wurzel T ist. Dann nämlich(, wie oben gezeigt) hat sich die Schwarz-Höhe der Wurzel — und damit die des gesamten Baums — um genau 1 erhöht. Dieser eine Fall kann außerdem nur genau einmal auftreten, da in diesem Falle als als nächstes zu betrachtender Knoten die Wurzel bestimmt wird, in welchem Falle die Schleife kein weiteres mal durchlaufen wird.

Aufgabe 3 (Anzahl roter Knoten im Rot-Schwarz-Baum): (10 Punkte)

Zeigen Sie, dass zu jeder Höhe h ein Rot-Schwarz-Baum $B(h)$ existiert mit der folgenden Anzahl roter Knoten $r(B(h))$:

$$r(B(h)) = \sum_{i=2}^{h} \left((1 + (-1)^{i-h}) \cdot 2^{i-2}\right)$$

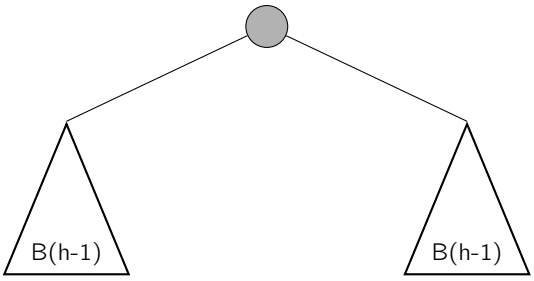
Lösung:

Wir konstruieren einen entsprechenden Baum mit Höhe h induktiv und zeigen, dass die Eigenschaft für jede Höhe erfüllt ist:

Induktionsverankerung:
Alle Bäume mit Höhe kleiner als zwei bestehen nur aus Wurzel und Blättern. Nach Definition besitzt er somit nur schwarze Knoten. Der Summenindex i startet bei 2 und läuft bis h somit ergibt sich für die Summe bei alle Höhen $h < 2$ Null. Somit ist die Induktionsverankerung für $h < 2$ gewährleistet.

Induktionsschritt:
Wir unterscheiden zwischen geraden und ungerader Höhe h:

1. Fall h ist ungerade:
   Konstruiere $B(h)$ wie folgt:

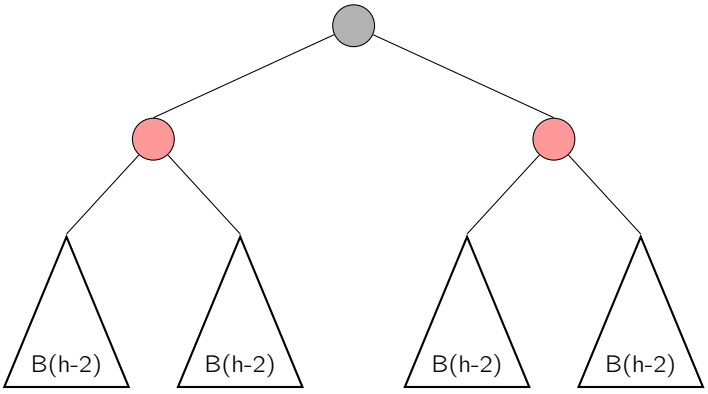


   Für den so konstruierten Baum $B(h)$ gilt für die Anzahl der roten Knoten $r(B(h))$:

$$\begin{aligned}
r(B(h)) &= 2 \cdot r(B(h-1)) && \text{|Induktionsvoraussetztung} \\
&= 2 \cdot \sum_{i=2}^{h-1} \left((1 + (-1)^{i-(h-1)}) \cdot 2^{i-2}\right) \\
&= \sum_{i=2}^{h-1} \left((1 + (-1)^{i-h+1)}) \cdot 2^{i-1}\right) && \text{|Indexverschiebung} \\
&= \sum_{i=3}^{h} \left((1 + (-1)^{i-h}) \cdot 2^{i-2}\right) && \text{|}h \text{ ungerade } \Rightarrow (1 + (-1)^{2-h}) = 0 \\
&= \sum_{i=2}^{h} \left((1 + (-1)^{i-h}) \cdot 2^{i-2}\right)
\end{aligned}$$

2. Fall h ist gerade:

Konstruiere $B(h)$ wie folgt:



Für den so konstruierten Baum $B(h)$ gilt für die Anzahl der roten Knoten $r(B(h))$:

$$
\begin{aligned}
r(B(h)) &= 2 + 4 \cdot r(B(h-2)) && \text{|Induktionsvoraussetztung} \\
&= 2 + 4 \cdot \sum_{i=2}^{h-2} \left((1 + (-1)^{i-(h-2)}) \cdot 2^{i-2} \right) \\
&= 2 + \sum_{i=2}^{h-2} \left((1 + (-1)^{i-h+2}) \cdot 2^{i} \right) && \text{|Indexverschiebung} \\
&= 2 + \sum_{i=4}^{h} \left((1 + (-1)^{i-h}) \cdot 2^{i-2} \right) && \text{|}h \text{ gerade } \Rightarrow (1 + (-1)^{3-h}) = 0 \wedge (1 + (-1)^{2-h}) = 2 \\
&= 2 + \sum_{i=2}^{h} \left((1 + (-1)^{i-h}) \cdot 2^{i-2} \right) - 2 \cdot 2^{2-2} \\
&= \sum_{i=2}^{h} \left((1 + (-1)^{i-h}) \cdot 2^{i-2} \right)
\end{aligned}
$$

Prof. Dr. Ir. Joost-Pieter Katoen

Jonathan Heinen, Sabrina von Styp

Hinweise:

- Die Übungsblätter sollen in Gruppen von je 3 Studierenden aus der gleichen Kleingruppenübung bearbeitet werden.
- Die Lösungen müssen bis Montag, den 21. Juni um 11:00 Uhr in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55).
- Namen und Matrikelnummern der Studierenden sowie die Nummer der Übungsgruppe sind auf jedes Blatt der Abgabe zu schreiben. Heften bzw. tackern Sie die Blätter!
- An den Freitagen **18.** und **25. Juni** findet, **keine** Vorlesung statt.
- Die Globalübung am 14. Juni findet ausnahmsweise im Raum H218 (Intzestrasse 5) statt.

Aufgabe 1 (Hashing): (4 + 12 + 2 + 2 + 2 Punkte)

In einem kleinen Studentenkino mit $m = 23$ Plätzen wird ein Film gezeigt. Um dem Ansturm Herr zu werden, sollen die Plätze mit einem offenen Hashverfahren auf die Wartenden verteilt werden. Als Schlüssel werden dabei nur die beiden letzten Ziffern der Matrikelnummer verwendet. Betrachten Sie die folgenden beiden Hashfunktionen:

- $h_1(x)$:= Quersumme von x
- $h_2(x)$:= x mod 23 (Division-Rest-Methode)

a) Diskutieren Sie, inwieweit h_1 und h_2 die Bedingungen, die an eine sinnvolle Hashfunktion gestellt werden, erfüllen.

b) Welche Platznummern erhalten die Besucher, wenn sie in der gegebenen Reihenfolge das Kino betreten?

$$6,\ 16,\ 61,\ 87,\ 69,\ 90,\ 4,\ 43,\ 57,\ 4,\ 12,\ 80,\ 46$$

Verwenden Sie jeweils folgende Hashfunktionen:

(i) lineares Sondieren mit h_1 und mit h_2 als Hashfunktion

(ii) quadratisches Sondieren mit h_1 und mit h_2 als Hashfunktion und mit $c_1 = 2$ und $c_2 = 3$ als Konstanten,

(iii) Doppelhashing mit h_1 als erster und h_2 als zweiter Hashfunktion. Inwieweit muss h_2 geändert werden, um Probleme zu vermeiden?

c) Warum ist ein geschlossenes Hashing, in dem geschildertem Szenario, nicht praktikabel bzw. sinnvoll?

d) Angenommen wir haben eine perfekte Hashfunktion und fügen die obigen Elemente in einen Hash der Größe 23 ein. Wieviele Sondierungen sind dann durchschnittlich bei erfolgreicher Suche nötig?

e) Angenommen wir haben eine perfekte Hashfunktion und fügen die obigen Elemente in einen Hash der Größe 23 ein. Wieviele Sondierungen sind dann durchschnittlich bei nicht erfolgreicher Suche nötig?

Lösung:

a) h_1 bildet alle Matrikelnummern auf den Bereich $0+0=0$ bis $9+9=18$ ab. Die Werte 19 bis 23 werden nicht genutzt. Darüber hinaus werden die Marikelnummern nicht gleichmäßig auf die Werte verteilt. So werden z.B. nur die Endziffern 00 auf 0, sowie 99 auf 18 abgebildet während auf z.B. 9 die Endziffern $09, 18, 27, 36, 45, 54, 63, 72, 81, 90$ abgebildet werden.

h_2 ist besser geeignet, da hier auf den gesamte Bereich abgebildet wird. Auch die Verteilung ist hier nahezu gleichmäßig. Auf die Werte zwischen 0 und 7 werden jeweils fünf verschiedene Werte abgebildet, auf die Werte von 8 bis 23 nur vier.

Aufgrund der schlechten Verteilung werden mit h_1 im Normalfall vielmehr Kollisionen auftreten als mit h_2.

b) Es ergeben sich die folgenden Hashwerte:

k	6	16	61	87	69	90	4	43	57	4	12	80	46
$h_1(k)$	6	7	7	15	15	9	4	7	12	4	3	8	10
$h_2(k)$	6	16	15	18	0	21	4	20	11	4	12	11	0

(i) Linear Sondiert mit $h(k,i) = (h_1(k) + i) \bmod 23$:

			12	4	4	6	16	61	90	43	80	57	46		87	69						

Linear Sondiert mit $h(k,i) = (h_2(k) + i) \bmod 23$:

69	46			4	4	6					57	12	80		61	16		87		43	90	

(ii) Quadratisch Sondiert mit $h(k,i) = (h_1(k) + 2 \cdot i + 3 \cdot i^2) \bmod 23$:

43			12	4		6	16	80	90	46		61		4	87		57			69		

Quadratisch Sondiert mit $h(k,i) = (h_2(k) + 2 \cdot i + 3 \cdot i^2) \bmod 23$:

69				4	46	6			4		57	12			61	16		87	80	43	90	

(iii) Doppeltes Hashing mit $h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod 23$:

						6	16								87							61

Die 69 kann nicht mehr sondiert werden, da $h_2(69) = 0$ und somit für jedes i auf $h_2(k)$ sondiert wird. Um dieses Problem zu vermeiden kann z.B. $h_2'(k) = (h_2(k) \bmod 22) + 1$ genutzt werden.

c) Das geschilderte Szenario ist ungeeignet für geschlossenes Hashing, da nicht mehrere Studenten auf einem Platz sitzen können und deswegen nicht mehrere Schlüsselwerte an die gleiche Stelle gehasht werden können.

d) Die erfolgreiche Suche hat die Komplexität $\mathcal{O}(\frac{1}{\alpha} \ln \frac{1}{1-\alpha})$ mit $\alpha = \frac{n}{m}$, wobei n die Anzahl der Hashwerte ist. Daraus ergibt sich folgende Anzahl an durchschnittlichen Sondierungen:

$$\frac{1}{\frac{13}{23}} \ln \frac{1}{1-\frac{13}{23}} \approx 1{,}47$$

Bemerkung: Da der einzige Grund für die $\mathcal{O}-$Notation die Komplexität zur Berechnung der Hashwerte ist und diese in $\mathcal{O}(1)$ liegt, darf hier die Konstante c weggelassen werden.

e) Die erfolglose Suche hat die Komplexität $\mathcal{O}(\frac{1}{1-\alpha})$ mit $\alpha = \frac{n}{m}$, wobei n die Anzahl der Hashwerte ist. Daraus ergibt sich folgende Anzahl an durchschnittlichen Sondierungen:

$$\frac{1}{1-\frac{13}{23}} = 2{,}3$$

Bemerkung: Da der einzige Grund für die $\mathcal{O}-$Notation die Komplexität zur Berechnung der Hashwerte ist und diese in $\mathcal{O}(1)$ liegt, darf hier die Konstante c weggelassen werden.

Aufgabe 2 (gutes Hashing): (5 Punkte)

Diskutieren Sie, welche Eigenschaften eine gute Hashfunktion für Strings haben sollte. Es sei m die Größe der Hashtabelle und $s = a_1 a_2 \dots a_n$ ein String, wobei wir die Zeichen mit ihren ASCII-Codes identifizieren. Wie gut sind die folgenden drei Hashfunktionen:

$$(\sum_{k=1}^{n} a_k) \mod m \qquad (\sum_{k=1}^{n} k \cdot a_k) \mod m \qquad a_1^{a_2^{a_3}} \mod m$$

Lösung:

- $(\sum_{k=1}^{n} a_k) \mod m$
  Bei dieser Hashfunktion werden kurze Wörter auch bei großem m auf einen sehr kleinen Bereich abgebildet. So gibt es z.B. $128^3 = 2\,097\,152$ verschiedene Wörter der Länge drei, die alle auf Werte zwischen 0 und $3 \cdot 128 = 384$ abgebildet werden. Für kurze Wörter ergeben sich so sehr viele Kollisionen.
- $(\sum_{k=1}^{n} k \cdot a_k) \mod m$
  Diese Hashfunktion ist etwas besser, da hier der Abbildungsbereich größer ist. Die 2 097 152 Wörter der Länge drei werden hier auf den Bereich von 0 bis $128 + 2 \cdot 128 + 3 \cdot 128 = 768$ abgebildet. Somit ist die Funktion immer noch nicht optimal für kurze Wörter, aber in jedem Fall besser als die erste Hashfunktion.
- $a_1^{a_2^{a_3}} \mod m$
  Diese Hashfunktion nutzt nur die ersten drei Zeichen der Zeichenkette, um den Hashwert zu berechnen. Somit werden Zeichenketten die mit den gleichen drei Buchstaben beginnen immer auf den gleich Wert abgebildet. Haben wir viele Zeichenketten mit den gleichen drei Anfangsbuchstaben erhalten wir sehr viele Kollisionen. Z.B. Prefix „auf“ in „aufgehen“, „aufheben“, „aufzeichnen“, $\cdots$

Aufgabe 3 (Countingsort): (6 + 2 Punkte)

a) Das folgendes Array ist mit Countingsort zu sortieren. Geben Sie das Histogramm- und das Positionsarray vor dem ersten Einfügen ins Ausgabearray an, sowie das Positions- und Ausgabearray nach jedem Einfügeschritt.

4	3	0	1	4	2	3	7	3

b) Der in der Vorlesung vorgestellte Algorithmus Countingsort fügt die Elemente des Eingabearrays von hinten nach vorne in das Ausgabearray ein. Welche Nachteile ergäben sich, wenn man das Eingabearray stattdessen von vorne nach hinten durchlaufen würde?

Lösung:

a)

Eingabearray

4	3	0	1	4	2	3	7	3

Histogramm

1	1	1	3	2	0	0	1
0	1	2	3	4	5	6	7

Positionen

1	2	3	6	8	8	8	9
0	1	2	3	4	5	6	7

Ausgabearray

0	1	2	3	4	5	6	7	8

Eingabearray

4	3	0	1	4	2	3	7	3

Positionen

1	2	3	5	8	8	8	9
0	1	2	3	4	5	6	7

Ausgabearray

					3			
0	1	2	3	4	5	6	7	8

Eingabearray

4	3	0	1	4	2	3	7	3

Positionen

1	2	3	5	8	8	8	8
0	1	2	3	4	5	6	7

Ausgabearray

					3			7
0	1	2	3	4	5	6	7	8

Eingabearray

4	3	0	1	4	2	3	7	3

Positionen

1	2	3	4	8	8	8	8
0	1	2	3	4	5	6	7

Ausgabearray

				3	3			7
0	1	2	3	4	5	6	7	8

Eingabearray

4	3	0	1	4	2	3	7	3

Positionen

1	2	2	4	8	8	8	8
0	1	2	3	4	5	6	7

Ausgabearray

		2		3	3			7
0	1	2	3	4	5	6	7	8

Eingabearray	**4**	**3**	**0**	**1**	**4**	**2**	**3**	**7**	**3**

Positionen	**1**	**2**	**2**	**4**	**7**	**8**	**8**	**8**
	0	1	2	3	4	5	6	7

Ausgabearray			**2**		**3**	**3**		**4**	**7**
	0	1	2	3	4	5	6	7	8

Eingabearray	**4**	**3**	**0**	**1**	**4**	**2**	**3**	**7**	**3**

Positionen	**1**	**1**	**2**	**4**	**7**	**8**	**8**	**8**
	0	1	2	3	4	5	6	7

Ausgabearray		**1**	**2**		**3**	**3**		**4**	**7**
	0	1	2	3	4	5	6	7	8

Eingabearray	**4**	**3**	**0**	**1**	**4**	**2**	**3**	**7**	**3**

Positionen	**0**	**1**	**2**	**4**	**7**	**8**	**8**	**8**
	0	1	2	3	4	5	6	7

Ausgabearray	**0**	**1**	**2**		**3**	**3**		**4**	**7**
	0	1	2	3	4	5	6	7	8

Eingabearray	**4**	**3**	**0**	**1**	**4**	**2**	**3**	**7**	**3**

Positionen	**0**	**1**	**2**	**3**	**7**	**8**	**8**	**8**
	0	1	2	3	4	5	6	7

Ausgabearray	**0**	**1**	**2**	**3**	**3**	**3**		**4**	**7**
	0	1	2	3	4	5	6	7	8

Eingabearray	**4**	**3**	**0**	**1**	**4**	**2**	**3**	**7**	**3**

Positionen	**0**	**1**	**2**	**3**	**6**	**8**	**8**	**8**
	0	1	2	3	4	5	6	7

Ausgabearray	**0**	**1**	**2**	**3**	**3**	**3**	**4**	**4**	**7**
	0	1	2	3	4	5	6	7	8

b) Fängt man beim ersten Element an, dann ist der Algorithmus nicht mehr stabil.

Prof. Dr. Ir. Joost-Pieter Katoen

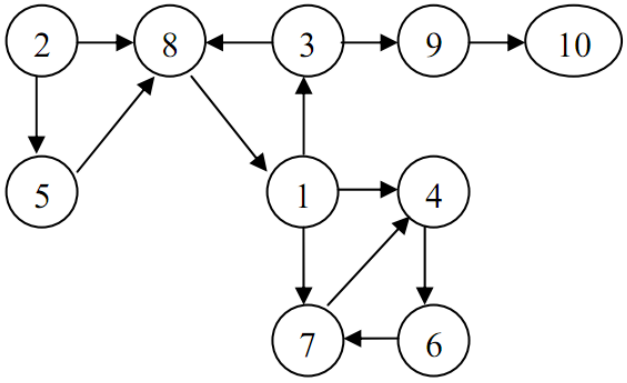
Jonathan Heinen, Sabrina von Styp

Hinweise:

- Die Übungsblätter sollen in Gruppen von je 3 Studierenden aus der gleichen Kleingruppenübung bearbeitet werden.
- Die Lösungen müssen bis Montag, den 28. Juni um 11:00 Uhr in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55).
- Namen und Matrikelnummern der Studierenden sowie die Nummer der Übungsgruppe sind auf jedes Blatt der Abgabe zu schreiben. Heften bzw. tackern Sie die Blätter!

Aufgabe 1 (Graphen): (4+4+8 Punkte)

Sei $G = (\{1, \ldots, 10\}, E)$ der folgende Graph:



a) Geben Sie den Graphen G als Adjazenzmatrix und in Adjazenzlistendarstellung an.

b) Führen Sie für den Graphen G die Breitensuche beginnend bei Knoten 1 durch. Geben Sie die Reihenfolge der Knotenbesuche an und heben Sie die entstehenden Baumkanten hervor.

c) Berechnen Sie mit Hilfe des in der Vorlesung vorgestellten Algorithmus die starken Zusammenhangskomponenten des obigen Graphen.

Lösung:

a) Der gegebene Graph hat die folgende Darstellung als Adjazenzmatrix:

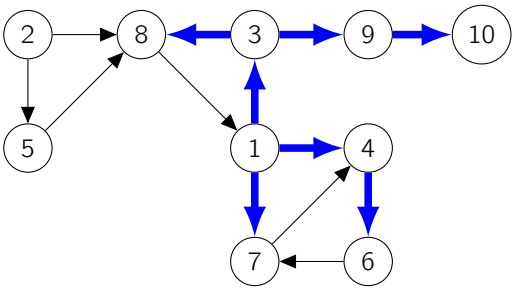
y-Achse: Von | x-Achse: Nach

-	1	2	3	4	5	6	7	8	9	10
1	0	0	1	1	0	0	1	0	0	0
2	0	0	0	0	1	0	0	1	0	0
3	0	0	0	0	0	0	0	1	1	0
4	0	0	0	0	0	1	0	0	0	0
5	0	0	0	0	0	0	0	1	0	0
6	0	0	0	0	0	0	1	0	0	0
7	0	0	0	1	0	0	0	0	0	0
8	1	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0

1 $\rightarrow 4 \rightarrow 7 \rightarrow 3$
2 $\rightarrow 5 \rightarrow 8$
3 $\rightarrow 8 \rightarrow 9$
4 $\rightarrow 6$
5 $\rightarrow 8$
6 $\rightarrow 7$
7 $\rightarrow 4$
8 $\rightarrow 1$
9 $\rightarrow 10$
10

b) Die Knoten werden in der folgenden Reihenfolge besucht (Knoten, die nur durch $\rightarrow$ voneinander getrennt sind werden gleichzeitig besucht):
$1 \Rightarrow 3 \rightarrow 4 \rightarrow 7 \Rightarrow 6 \rightarrow 8 \rightarrow 9 \Rightarrow 10$

Es ergibt sich der folgende Baum (dicke, blau eingezeichnete Kanten):



c) In dieser Lösung durchlaufen wir die Knoten in aufsteigender Reihenfolge. Ebenso durchlaufen wir die Nachfolger aufsteigend.

Nach dem ersten Durchlauf der Tiefensuche ergibt sich der folgende Stack:

8	10	9	3	7	6	4	1	5	2

$\leftarrow$ top

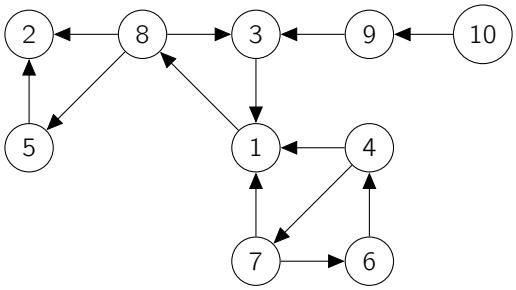
Die im folgenden Rot markierten Einträge werden als Leiter erkannt:

8	10	9	3	7	6	4	1	5	2

$\leftarrow$ top

Die folgenden Zuteilung zu SCCs wird in der zweiten Phase durch Tiefensuche, startend bei den Leitern, erkannt:

2	: 2
5	: 5
1	: 3, 8, 1
4	: 6, 7, 4
9	: 9
10	: 10



Aufgabe 2 (Backtracking): (4+10+4 Punkte)

Die Knoten eines ungerichteten Graphen sollen mit vier Farben so eingefärbt werden, dass zwei benachbarte Knoten (zwei Knoten, die durch eine Kante verbunden sind) nie die gleiche Farbe besitzen. Der Graph wird durch die Adjazenzmatrix `boolean[][] matrix` dargestellt. Das Array `int[] farbe` beinhaltet die Farbe, die dem jeweiligen Knoten zugewiesen ist. Die Farben werden durch die Zahlen 1, 2, 3 und 4 kodiert. Somit ist der Knoten `i` mit der Farbe `farbe[i]` eingefärbt. Mit der Zahl 0 kennzeichnen wir Knoten, die noch keine Färbung erhalten haben.

Hinweis: Knoten sind nie mit sich selbst über eine Kante verbunden.

a) Implementieren Sie die Java-Methode:

```
boolean test(boolean[][] matrix, int[] farbe),
```

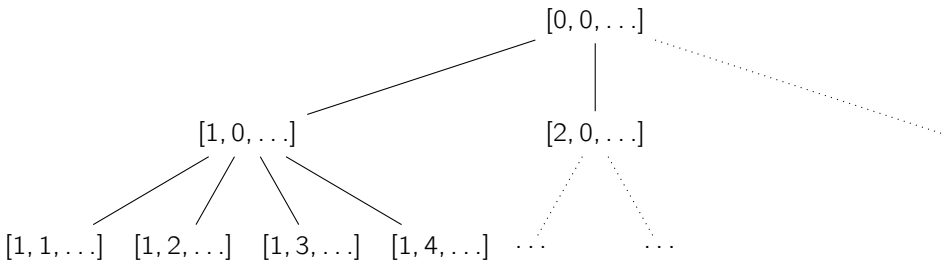
die testet, ob es sich um eine gültige Färbung handelt, d.h. durch Kanten verbundenen Knoten nie die gleiche Farbe besitzen.

b) In dieser Teilaufgabe soll nun ein Methode implementiert werden die, basierend auf Backtracking, einer gültige Färbung bestimmt.

Backtracking arbeitet nach dem Prinzip der Tiefensuche auf dem *Baum aller möglichen Lösungen*.

Ausgehend von einem vollständig ungefärbten Graphen wird ein Knoten nach dem anderen eingefärbt. Die Färbungen, die wir erhalten, indem wir einen weiteren Knoten einfärben, bezeichnen wir als Nachfolger. Für jeden Knoten haben wir vier mögliche Färbungen (1 . . . 4) und somit auch vier Nachfolger.

Diese Nachfolgerbeziehung, zusammen mit den vollständigen wie unvollständigen Färbungen, kann als Graph aufgefasst werden (siehe Abbildung). Es ergibt sich ein Baum, in dem die Wurzel den vollständig ungefärbten Graphen repräsentiert und jedes Blatt eine vollständige Färbung:



Eine gültige Färbung kann nun innerhalb dieses Baumes mit Hilfe der Tiefensuche gefunden werden. Hierbei wird der Baum nicht zuerst vollständig berechnet, sondern die Nachfolger werden während dem Durchlaufen berechnet.

Dies geschieht hier beim Absteigen durch das Hinzufügen einer Farbe für den nächsten (ungefärbten) Knoten, sowie beim Zurücklaufen (Backtracking) durch das Entfernen der Färbung des aktuellen (zuletzt gefärbten) Knoten. Implementieren Sie die Java-Methode

```
boolean loese(boolean[][] matrix, int[] farbe, int knoten),
```

die für den übergebenen Graphen eine gültige Färbung (ab Knoten n) berechnet, die entsprechend Tiefensuche den Baum traversiert und in den Blättern überprüft, ob es sich um eine gültige Färbung handelt. Existiert eine solche Färbung, so wird `true` zurückgegeben und `farbe` enthält eine gültige Lösung. Gibt es keine solche Färbung, so wird `false` zurückgegeben.

c) Während des Abstiegs im Baum werden Knoten besucht, die teilweise eingefärbte Graphen repräsentieren. Auch diese Teilfärbungen können bereits gleichgefärbte benachbarte Knoten besitzen. Ist dies der Fall, so

sind alle vollständige Färbungen, die sich in den darunter liegenden Blättern befinden, ungültig und wir brauchen diesen gesamten Teilbaum nicht zu durchsuchen, sondern können unmittelbar im Baum zurückgehen (Backtracking-Schritt). Passen Sie Ihre Implementierung so an, dass bereits beim Erreichen einer unzulässigen Teilfärbung ein Backtracking-Schritt vollzogen wird, d.h. dass dieser Teilbaum nicht weiter durchsucht wird.

Hinweise:

1. Für die Tiefensuche im Baum ist es nicht nötig Knoten mit einer Farbe zu versehen, da sichergestellt ist, dass jeder Knoten nur einen, eindeutigen Vorgänger besitzt. Somit wird jeder Knoten nur exakt einmal besucht. Vergleichen Sie hierzu auch die Algorithmen zur Baumtraversierung aus der dritten Vorlesung, die der Tiefensuche entsprechen.
2. Im L2P, unter den Materialien zur Globalübung, finden Sie die Implementierung des Backtracking-Algorithmuses zur Lösung von Sudokus, der in der Globalübung am 21. Juni vorgestellt wurde.
3. Die in dieser Aufgabe zu implementierenden Algorithmen sollen in Java implementiert werden. Schicken Sie Ihren kompilierbaren und lauffähigen Quellcode per E-Mail an Ihren Tutor. Die E-Mail-Adresse Ihres Tutors finden Sie im L2P unter Teilnehmer.

Lösung:

a) Die folgende `test`-Methode überprüft ob zwei verbundene Knoten die gleiche Färbung besitzen. Hierbei ist zulässig, dass zwei benachbarte Knoten ungefärbt (mit Farbe 0 versehen) sind. Letzteres benötigen wir für Aufgabenteil **c)**.

```
public static boolean test(boolean[][] matrix, int[] farbe){
    // Durchlaufe alle Paarungen von Knoten (i,j)
    for(int i = 0; i < matrix.length; i++){
        for(int j = 0; j < matrix.length; j++){
            // Teste, ob die beiden Knoten verbunden sind und die gleiche Farbe haben
            if(matrix[i][j]){
                // Es gibt eine Kante zwischen den Knoten i und j
                if(farbe[i] != 0 && farbe[i] == farbe[j]){
                    // Die Knoten i und j sind verbunden und besitzen die
                    // gleiche Farbe. Somit liegt eine Verletzung vor.
                    return false;
                }
            }
        }
    }
    // Wir sind alle Kombinationen durchgegangen und haben keine Verletzung gefunden
    return true;
}
```

b) Die Methode `loese` traversiert den Lösungsbaum entsprechend der Tiefensuche und testet in jedem Blatt, ob es sich um eine gültige Lösung handelt. Wird eine solche Lösung gefunden, so wird diese zurückgegeben.

```
boolean loese(boolean[][] matrix, int[] farbe, int knoten){
    // Wenn wir den letzten Knoten bereits eingefärbt haben
    // Testen wir ob wir eine gültige Färbung gefunden haben.
    if(knoten == farbe.length)
        return test(matrix, farbe);

    // Es gibt noch einen ungefärbten Knoten. Wir testen alle möglichen Farben.
    for(int i = 1; i <= 4; i++){
        // Wir geben den Knoten die Farbe i
        farbe[knoten] = i;
        //... und testen, ob es ausgehend von diesem Nachfolger eine Lösung gibt.
        if(loese(matrix, farbe, knoten+1))
            return true;
        // wenn es keine Lösung gibt versuchen wir den nächsten Nachfolger ...
    }
    // Wir haben in keinem der Nachfolger einen Lösung gefunden!
    // Wir setzen die Färbung zurück ...
    farbe[knoten] = 0;

    // und übergeben, mit einer Erfolglos-Meldung wieder an den Vorgänger
    // um dort weiter zu suchen.
    return false;
}
```

c) Die Methode aus **b)** kann noch optimiert werden, indem man die Suche unterhalb eines inneren Knoten abbricht, wenn bereits der innere Knoten zwei benachbarte Knoten gleicher Farbe besitzt. Die Test-Methode aus **a)** wurde so implementiert, dass sie auch dann `true` zurückgibt, wenn es innerhalb der Teilfärbung keine Verletzungen gibt.

```
boolean loese(boolean[][] matrix, int[] farbe, int knoten){

    // Wir testen, ob es eine Verletzung für diese Teilfärbung gibt
    if(!test(matrix, farbe))
        return false;

    // Es wurde keine Verletzung festgestellt. Haben wir bereits den letzten
    // Knoten erreicht, so handelt es sich um eine gültige Lösung.
    if(farbe.length == knoten)
        return true;

    // Der Rest bleibt wie in Aufgabenteil b)
    for(int i = 1; i <= 4; i++){
        farbe[knoten] = i;
        if(loese(matrix, farbe, knoten+1))
            return true;
    }

    farbe[knoten] = 0;
    return false;
}
```

Prof. Dr. Ir. Joost-Pieter Katoen

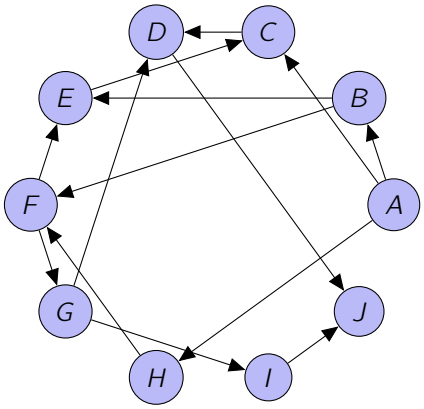
Jonathan Heinen, Sabrina von Styp

Hinweise:

- Die Übungsblätter sind in Gruppen von je 3 Studierenden der gleichen Kleingruppenübung zu bearbeitet.
- Die Lösungen müssen bis Montag, den 5. Juli um 11:00 Uhr in die Übungskasten eingeworfen werden.
- Namen und Matrikelnummern, sowie die Nummer der Übungsgruppe sind auf jedes Blatt zu schreiben.

Aufgabe 1 (topologische Sortierung): (3+4+6 Punkte)

Gegeben seien der folgende Graph G, sowie die folgenden Laufzeiten:



Aktivität	Laufzeit
A	0
B	4
C	16
D	9
E	2
F	7
G	5
H	8
I	3
J	10

a) Finden Sie eine topologische Sortierung für den Graphen G.

b) Bestimmen Sie den Kritischen-Pfad auf der topologischen Sortierung von **a)** mit den obigen Kosten.

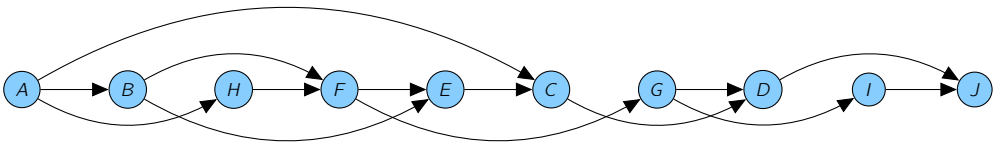
c) In der Vorlesung haben wir eine Erweiterung der Tiefensuche kennengelernt, die den Kritischen-Pfad eines Graphen berechnet. Vervollständigen Sie den folgende Programmcode, sodass der berechnete Kritische-Pfad, sowie seine Kosten ausgeben werden.

Sie dürfen beliebige Hilfsmethoden schreiben, Ihre Lösung muss jedoch eine lineare Laufzeit besitzen.

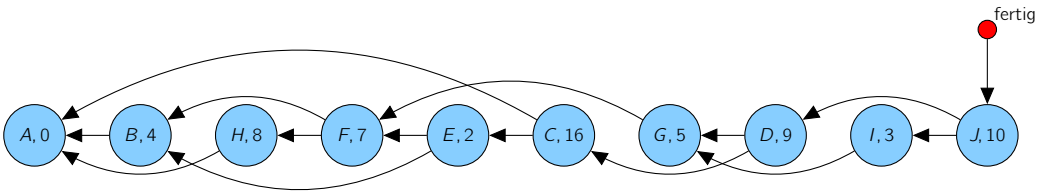
```
void printPath(List adjL[n], int n, int duration[n] ){
    int eft[n]; int critDep[n]; int color[n];
    for (int v = 0; v < n; v++) { color[v] = WHITE; }
    for (int v = 0; v < n; v++) {
        if (color[v] == WHITE) { dfsSearch(adjL, n, v, color, duration, critDep, eft); }
    }

    int maxDuration;
    // Hier muss Code ergänzt werden ...
    ausgabe("Dauer des Kritischen-Pfads: " + maxDuration);

    ausgabe("Kritischer-Pfad:");
    // Hier muss Code ergänzt werden ...
}
```

Lösung:

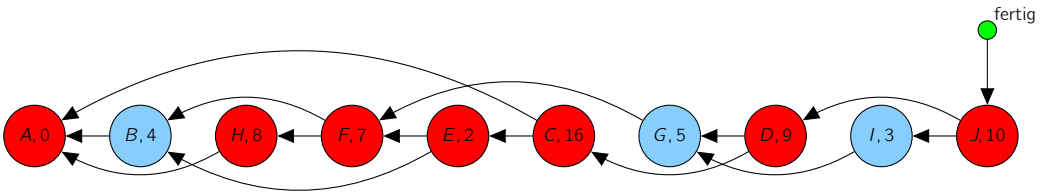
a) eine mögliche topologische Sortierung ist:



b) Aufstellen der Pfade:



nach Durchführung des Algorithmus:



Der kritische Pfad besteht also aus den Knoten A,H,F,E,C,D und J mit Kosten 52.

c) Zuerst müssen wir den Knoten mit dem höchsten earliest finish time (eft) finden. Von diesem aus durchlaufen wir dann die Vorgängerliste um den Kritischen-Pfad auszugeben.

```
void printPath(List adjL[n], int n, int duration[n] ){
    int eft[n]; int critDep[n]; int color[n];
    for (int v = 0; v < n; v++) { color[v] = WHITE; }
    for (int v = 0; v < n; v++) {
        if (color[v] == WHITE) { dfsSearch(adjL, n, v, color, duration, critDep, eft); }
    }
    int maxDuration;
    // Wir suchen uns den Knoten mit der höchsten eft.
    int maxNode = 0;
    for(int i = 1: i < n; i++)
        if(eft[maxNode] < eft[i]) maxNode = i;

    // Die Laufzeit des Kritischen-Pfads ist gleich der höchsten eft
    maxDuration = eft[maxNode];

    ausgabe("Dauer des Kritischen-Pfads: " + maxDuration);
    ausgabe("Kritischer-Pfad:");
    // Der Knoten mit höchster eft ist der letzte Knoten des Kritischen-Pfades
    // von hier aus gehen wir den Pfad zurück bis zum ersten Knoten.
    // Zum zurückgehen nutzen wir die Information über den Vorgänger aus critDep
    printPfadAusDep(critDep, maxNode);
    ausgabe("fertig");
}

// printPfadAusDep gibt den Kritischen Pfad rekursiv aus
void printPfadAusDep(int critDep[n], int node){
    // gibt es weiter Vorgänger, so wird zuerst der Pfad bis zu diesem Knoten ausgegeben
    if(critDep[node] != -1)
        printPfadAusDep(critDep, critDep[node]);
    // Zuletzt wird der Knoten selbst ausgegeben
    ausgabe(node + " -> ");
}
```

Aufgabe 2 (Beweis auf Graphen): (6 Punkte)

Zeigen Sie, dass für jeden gerichteten Graphen G gilt, dass die Transposition des Kondensationsgraphen von G gleich dem Kondensationsgraphen der Transposition von G ist:

$$(G\downarrow)^T \;=\; (G^T)\downarrow$$

Lösung:

Sei $G = (V, E)$ ein gerichteter Graph.

Dann ist der Kondensationsgraph $G\downarrow$ von G

$$G\downarrow \;=\; (\{S_1, \ldots S_p\},\ \{(S,T) \mid \exists (v,w) \in E\colon v \in S, w \in T\}),$$

wobei $\{S_1, \ldots, S_p\}$ die Menge der starken Zusammenhangskomponenten von G ist.
Die Transposition $(G\downarrow)^T$ von $G\downarrow$ ist dann

$$(G\downarrow)^T \;=\; (\{S_1, \ldots S_p\},\ \{(T,S) \mid \exists (v,w) \in E\colon v \in S, w \in T\})$$

Die Transposition G^T von G ist

$$G^T \;=\; (V,\ \{(w,v) \mid \exists (v,w) \in E\})$$

Da für einen beliebigen gerichteten Graphen die starken Zusammenhangskomponenten des Graphen und seiner Transposition gleich sind (siehe Vorlesung), ist der Kondensationsgraph $(G^T)\downarrow$ von G^T

$$(G^T)\downarrow \;=\; (\{S_1, \ldots S_p\},\ \{(T,S) \mid \exists (w,v) \in \{(w,v) \mid \exists (v,w) \in E\}\colon w \in T, v \in S\})$$

Wegen $\exists (w,v) \in \{(w,v) \mid \exists (v,w) \in E\} \;\Leftrightarrow\; \exists (v,w) \in E$ gilt

$$(G^T)\downarrow \;=\; (\{S_1, \ldots S_p\},\ \{(T,S) \mid \exists (v,w) \in E\colon w \in T, v \in S\}) \;=\; (G\downarrow)^T$$

Damit ist die Aussage gezeigt.

Prof. Dr. Ir. Joost-Pieter Katoen

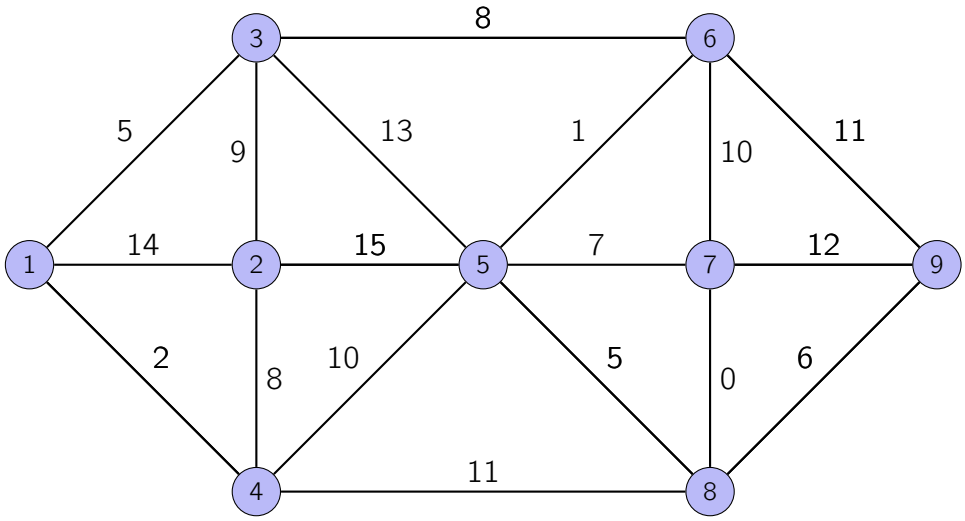
Jonathan Heinen, Sabrina von Styp

Hinweise:

- Die Übungsblätter sind in Gruppen von je 3 Studierenden der gleichen Kleingruppenübung zu bearbeitet.
- Die Lösungen müssen bis Montag, den 12. Juli um 11:00 Uhr in die Übungskasten eingeworfen werden.
- Namen und Matrikelnummern, sowie die Nummer der Übungsgruppe sind auf jedes Blatt zu schreiben.

Aufgabe 1 (Prim-Algorithmus): (5+2+4+6+4 Punkte)

a) Bestimmen Sie mit dem Algorithmus von Prim einen minimalen Spannbaum auf dem nachfolgenden Graphen und geben Sie an, in welcher Reihenfolge Sie die Kanten in den Spannbaum aufnehmen. Starten Sie beim Knoten 1.



b) Geben sei der folgende Algorithmus, der zu einem zusammenhängenden, gewichteten, ungerichteten Graphen $G = (V, E)$ einen MST berechnet.

```
1   mst(Graph G = (V, E) ){
2       // sort gibt eine absteigend sortierte Folge der übergebenen Menge zurück
3       E' = sort(E);
4       foreach (k in E')
5           if ( zusammenhaengend( (V, E'\{k}) ) ) // E'\{k} ist die Menge E' ohne Kante k
6               E':=E'\{k}
7       return (V, E')
8   }
```

Geben Sie ein Verfahren aus der Vorlesung an, mit dem der Test auf Zusammenhang in Zeile 5 realisiert werden kann.

c) Bestimmen Sie die Laufzeit des Algorithmus aus *b)* in Abhängigkeit von $n = |V|$ und $m = |E|$.

d) Zeigen Sie, dass der obige Algorithmus einen Spannbaum berechnet, indem sie zeigen,

(i) dass das Ergebnis zusammenhängend ist und

(ii) dass das Ergebnis zykelfrei ist.

e) Zeigen Sie, dass der obige Algorithmus einen minimalen Spannbaum berechnet.

Lösung:

a) Reihenfolge: $\{1,4\},\ \{1,3\},\ \{4,2\},\ \{3,6\},\ \{6,5\},\ \{5,8\},\ \{8,7\},\ \{8,9\}$

b) z.B. Tiefensuche (DFS)

c) $\mathcal{O}(\underbrace{m \log m}_{\text{Zeile 3}} + \underbrace{m}_{\text{Zeile 4}} + \underbrace{m(m+n))}_{\text{m-mal DFS}} = \mathcal{O}(m + m^2 + mn) = \mathcal{O}(m^2)$, da $m \geq n-1$ für zusammenhängende Graphen.

d) Zu zeigen ist: Am Ende des Algorithmus (Zeile 7) ist T zusammenhängend und kreisfrei.

Zusammenhang:

Annahme: Der Graph T zerfällt in Zeile 7 in zwei Zusammenhangskomponenten. Da initial $T = G$, und G zusammenhängend, muss es einen Schleifenindex k geben, so dass vor dem Entfernen von Kante e_k Graph T noch zusammenhängend war, nach dem Entfernen aber nicht mehr. Kante e_k wird aber in Zeile 6 nur entfernt, wenn $T \setminus \{e_k\}$ zusammenhängend ist - ein Widerspruch zur Annahme.

Kreisfreiheit

Annahme: Der Graph T enthält in Zeile 7 noch einen Kreis. Sei K die Menge der Kanten eines kleinsten Kreises in T. Es kann mindestens eine der Kanten aus K entfernt werden, ohne dass der Zusammenhang zerstört wird. Da wir in Zeile 4 über alle Kanten iterieren, muss eine der Kanten aus K als erste getestet und entfernt worden sein, denn ihr Entfernen zerstört offensichtlich nicht den Zusammenhang. Damit kann aber am Ende in Zeile 7 kein Kreis K mehr existieren. Widerspruch!

e) Nach Teil d.) ist T ein Spannbaum. Wir zeigen: T ist *minimaler* Spannbaum.

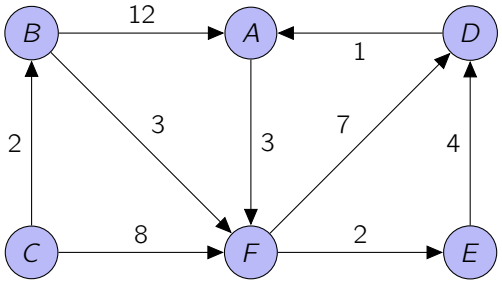
Annahme: Der Graph T ist nicht minimal und es gibt einen minimalen Spannbaum M mit geringerem Kantengewicht. Dann gibt es mindestens eine Kante e in T, die nicht in M vorkommt. Kante e muss auf einem Kreis in G liegen (sonst könnte sie in M nicht fehlen ohne den Zusammenhang in M zu zerstören). Von diesem Kreis lieft nur Kante e in T, in M muss eine andere Kante des Kreises, sagen wir e' liegen, die selber nicht in T vorkommt.

Unser Algorithmus durchläuft die Kanten in Reihenfolge nicht aufsteigender Gewichte. Wenn e' in T fehlt, e aber nicht, muss e' aus T entfernt worden sein (Zeilen 5–6) bevor der Algorithmus die Kante e testet. Also gilt: $gewicht(e') \geq gewicht(e)$. Da sich unsere Spannbäume T und M im Gewicht unterscheiden, muss es auch ein Paar (e, e') geben mit $gewicht(e') > gewicht(e)$.

Damit könnten wir das Gewicht von M verbessern, indem wir e' aus M entfernen und stattdessen e benutzen. Das ist aber ein Widerspruch zur Minimalität von M.

Aufgabe 2 (Bellman-Ford-Algorithmus): (6+6 Punkte)

a) Bestimmen Sie, für den Startknoten C, die Längen der kürzesten Wege zu jedem Knoten im unten gegebenen Graphen G mit Hilfe des Bellman-Ford Algorithmus. Dabei sei die Reihenfolge der Knoten gegeben durch die alphabetische Sortierung $(A, B, C, \dots)$. Geben Sie für jeden Knoten K die Kosten an, die dem Knoten während der Berechnung zugewiesen werden. Z.B. im Format $K : \infty, 10, 7, 6, 3$:



b) Passen Sie die Implementierung von Bellman-Ford, die Sie in der Vorlesung kennengelernt haben (Vorlesung 17, Folie 13), so an, dass der kürzeste Pfad zwischen zwei Knoten i und j ausgegeben wird. Sie dürfen davon ausgehen, dass der übergebene Graph keine negativen Zykel besitzt.

Ihre Funktion soll die folgende Signatur haben:

```
void bellFord(List adjLst[n], int n, int i, int j)
```

Lösung:

a) Es ergeben sich die folgenden Zuweisungen:

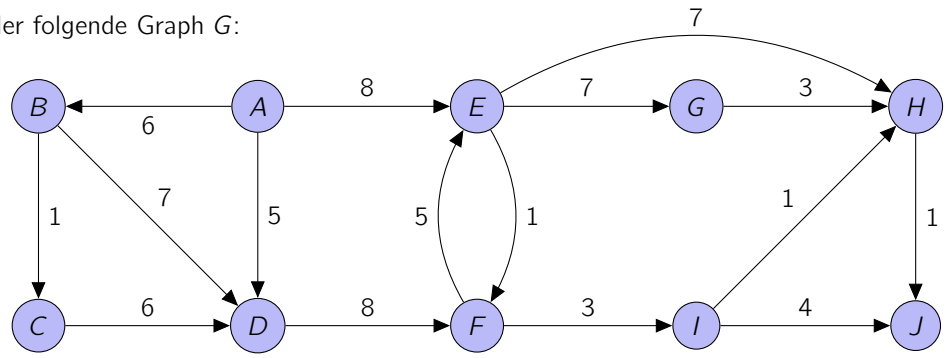
			1/C	1/F	2/B	2/E	2/F	3/D	3/E	4/D	
A:	∞				14			13		12	A: ∞, 14, 13, 12
B:	∞		2								B: ∞, 2
C:	∞	0									C: ∞, 0
D:	∞			15		14	12		11		D: ∞, 15, 14, 12, 11
E:	∞			10			7				E: ∞, 10, 7
F:	∞		8		5						F: ∞, 8, 5

b) Der folgende Algorithmus gibt einen kürzesten Pfad von i nach j aus:

```
void bellFord(List adjLst[n], int n, int i, int j){
    int d[n] = +inf;
    d[i] = 0;

    int parent[n] = -1;

    for (int k = 1; k < n; k++) // n-1 Durchläufe
        for (int v = 0; v < n; v++) // alle Kanten
            foreach (edge in adjLst[v])
                if (d[edge.w] > d[v] + edge.weight) {
                    d[edge.w] = d[v] + edge.weight;
                    parent[edge.w] = v;
                }
    ausgabe("Ein kürzester Weg mit Länge " + d[j] + ":");
    printPfad(parent, j);
}

// printPfad gibt den Pfad rekursiv aus
void printPfad(int parent[n], int node){
    // gibt es weiter Vorgänger, so wird zuerst der Pfad bis zu
        diesem Knoten ausgegeben
    if(parent[node] != -1){
        printPfad(parent, parent[node]);
        ausgabe(" -> ");
    }
    // Zuletzt wird der Knoten selbst ausgegeben
    ausgabe(node);
}
```

Aufgabe 3 (Dijkstra-Algorithmus): (6 Punkte)

Gegeben sei der folgende Graph G:



Ermitteln Sie mit Hilfe des Dijkstra-Algorithmus, unter Verwendung einer Tabelle der folgenden Form, die kürzesten Wege von Startknoten A zu allen anderen Knoten des Graphen. Notieren Sie für jeden Rechenschritt den aktuell gewählten Knoten zur Verbesserung der Wege und die Länge der bis zu diesem Zeitpunkt möglichen kürzesten Wege für jeden noch nicht abgeschlossenen Knoten ($D[\ldots]$). Streichen Sie Felder der Tabelle die nicht mehr benötigt werden durch.

Schritt	0	1	2	3	4	5	6	7	8	9
Knoten										
$D[A]$										
$D[B]$										
$D[C]$										
$D[D]$										
$D[E]$										
$D[F]$										
$D[G]$										
$D[H]$										
$D[I]$										
$D[J]$										

Lösung:

Schritt	0	1	2	3	4	5	6	7	8	9
Knoten	A	D	B	C	E	F	I	H	J	G
$D[A]$	0	—	—	—	—	—	—	—	—	—
$D[B]$	6	6	6	—	—	—	—	—	—	—
$D[C]$	∞	∞	7	7	—	—	—	—	—	—
$D[D]$	5	5	—	—	—	—	—	—	—	—
$D[E]$	8	8	8	8	8	—	—	—	—	—
$D[F]$	∞	13	13	13	9	9	—	—	—	—
$D[G]$	∞	∞	∞	∞	15	15	15	15	15	15
$D[H]$	∞	∞	∞	∞	15	15	13	13	—	—
$D[I]$	∞	∞	∞	∞	∞	12	12	—	—	—
$D[J]$	∞	∞	∞	∞	∞	∞	16	14	14	—

Prof. Dr. Ir. Joost-Pieter Katoen

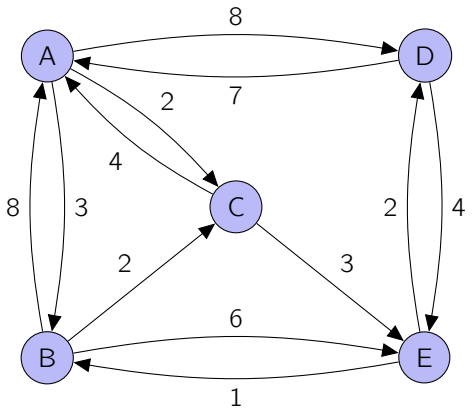
Jonathan Heinen, Sabrina von Styp

Hinweise:

- Die Übungsblätter sind in Gruppen von je 3 Studierenden der gleichen Kleingruppenübung zu bearbeitet.
- Die Lösungen müssen bis Montag, den 19. Juli um 11:00 Uhr in die Übungskasten eingeworfen werden.
- Namen und Matrikelnummern, sowie die Nummer der Übungsgruppe sind auf jedes Blatt zu schreiben.

Aufgabe 1 (Floyd-Algorithmus): (8 Punkte)

Bestimmen Sie, für den folgenden Graphen, die Werte der kürzesten Pfade zwischen allen Paaren von Knoten. Nutzen Sie hierzu den Algorithmus von Floyd. Geben Sie das Distanzarray vor dem Aufruf, sowie nach jeder Iteration an.
Die Knotenreihenfolge sei mit A, B, C, D, E fest vorgegeben.



Lösung:

	A	B	C	D	E
A	0	3	2	8	∞
B	8	0	2	∞	6
C	4	∞	0	∞	3
D	7	∞	∞	0	4
E	∞	1	∞	2	0

	A	B	C	D	E
A	0	3	2	8	∞
B	8	0	2	16	6
C	4	7	0	12	3
D	7	10	9	0	4
E	∞	1	∞	2	0

	A	B	C	D	E
A	0	3	2	8	9
B	8	0	2	16	6
C	4	7	0	12	3
D	7	10	9	0	4
E	9	1	3	2	0

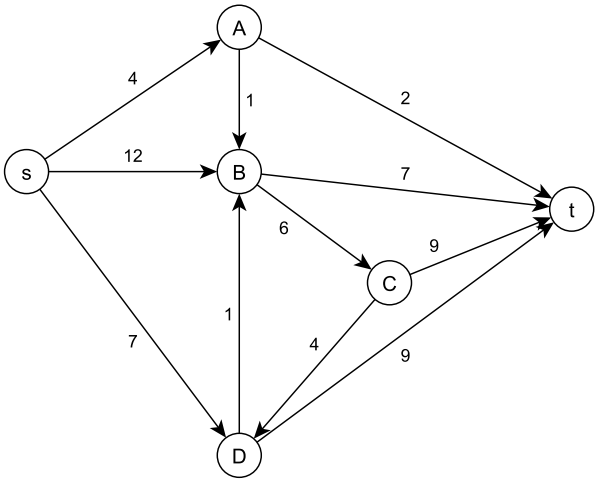
	A	B	C	D	E
A	0	3	2	8	5
B	6	0	2	14	5
C	4	7	0	12	3
D	7	10	9	0	4
E	7	1	3	2	0

	A	B	C	D	E
A	0	3	2	8	5
B	6	0	2	14	5
C	4	7	0	12	3
D	7	10	9	0	4
E	7	1	3	2	0

	A	B	C	D	E
A	0	3	2	7	5
B	6	0	2	7	5
C	4	4	0	5	3
D	7	5	7	0	4
E	7	1	3	2	0

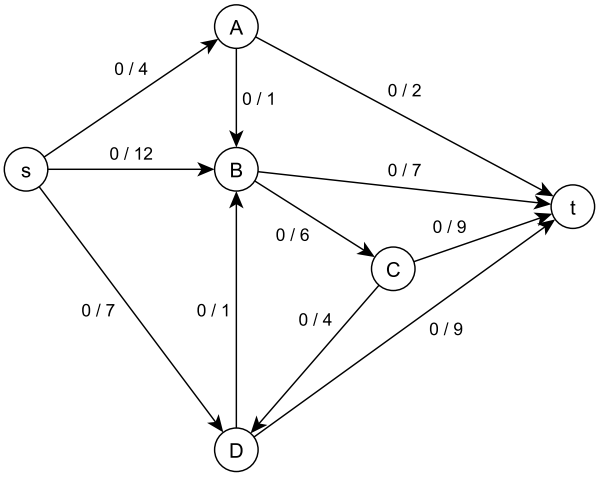
Aufgabe 2 (Maximaler Fluss): (8 Punkte)

Verwenden Sie die Ford-Fulkerson-Methode, um den maximalen Fluss des folgenden Flussnetzwerkes mit den gegebenen Kapazitäten zu berechnen. Geben Sie nach jeder Flusserhöhung das Flussnetzwerk an und kennzeichnen Sie den augmentierten Pfad. Geben Sie ebenfalls das Restnetzwerk nach der ersten, zweiten und letzten Flusserhöhung an.

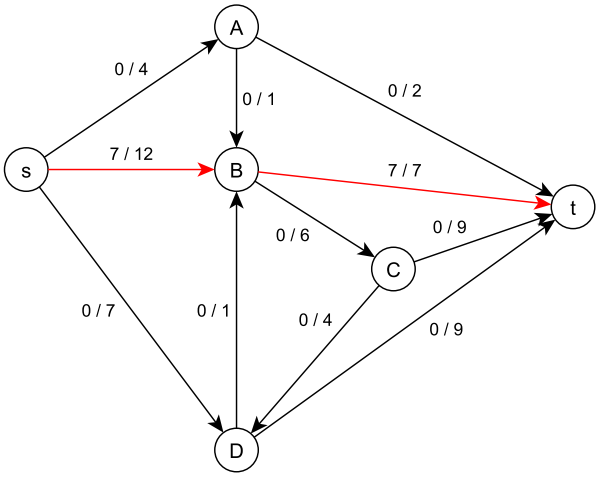


Lösung:

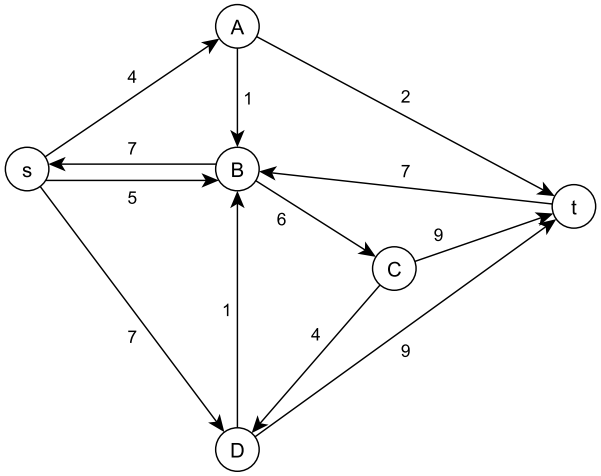
Überführung ins Flussnetzwerk:



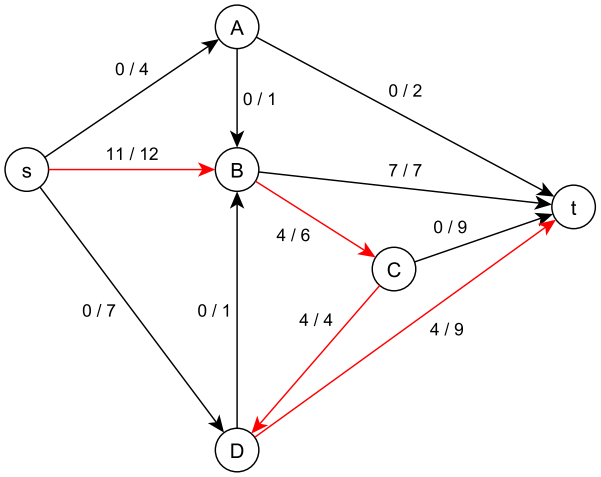
Erste Flusserhöhung:



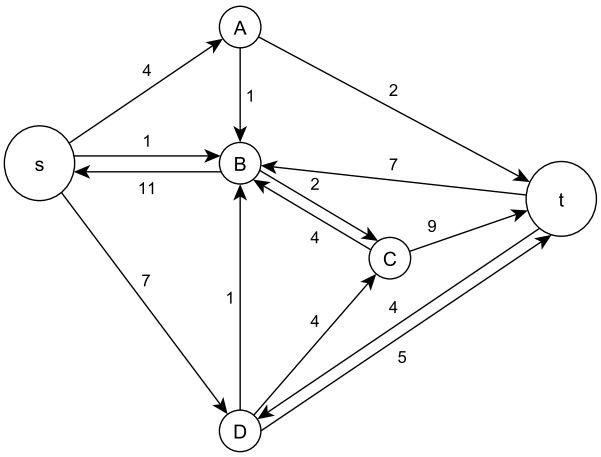
Restnetzwerk:



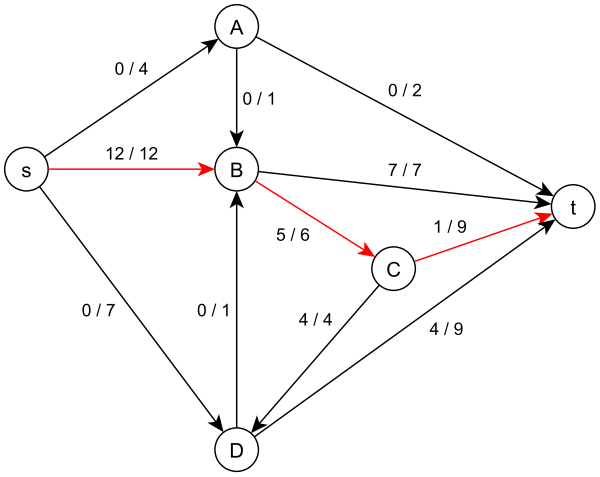
Zweite Flusserhöhung:



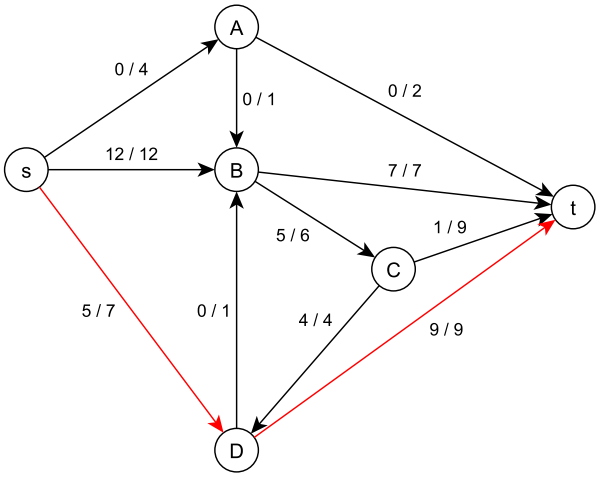
Restnetzwerk:



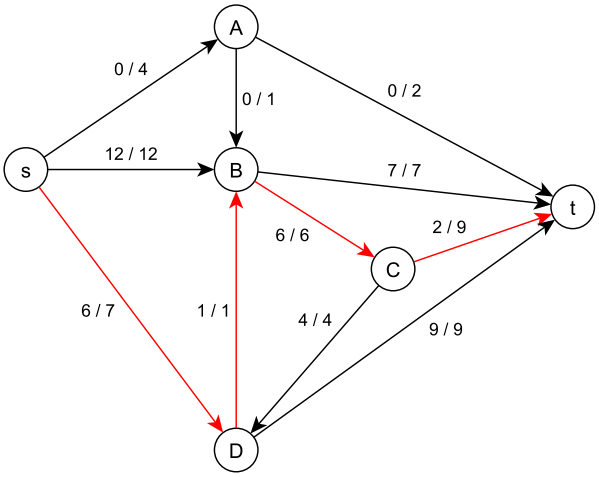
Dritte Flusserhöhung:



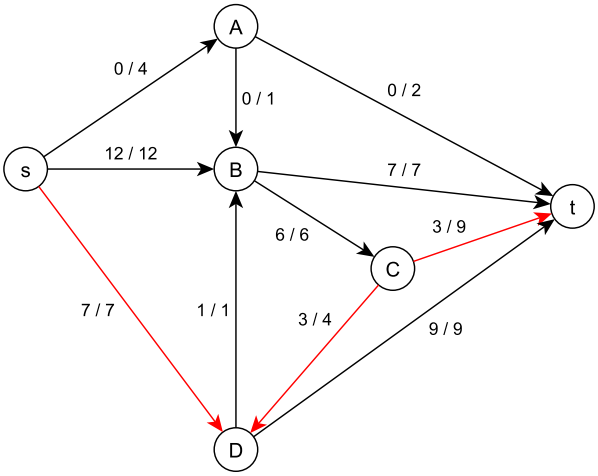
Vierte Flusserhöhung:



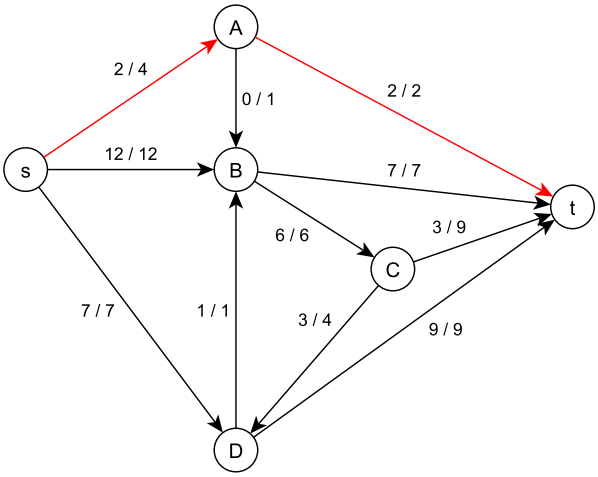
Fünfte Flusserhöhung:



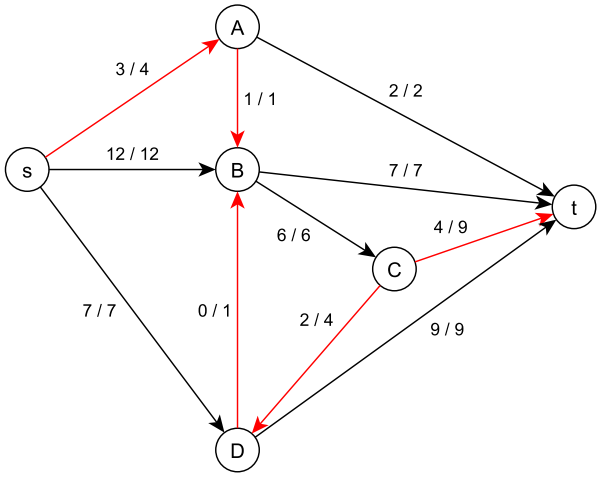
Sechste Flusserhöhung:



Siebte Flusserhöhung:

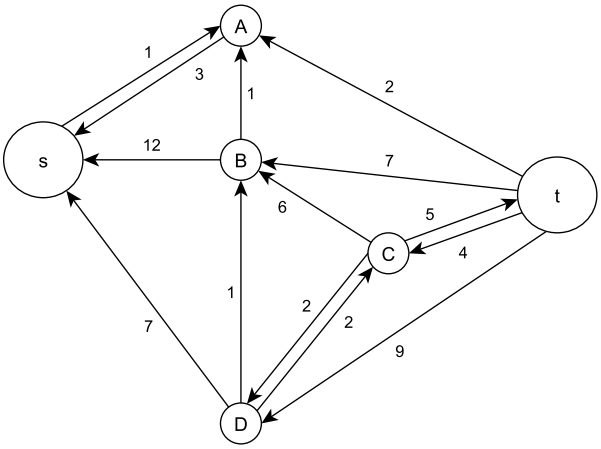


Achte Flusserhöhung:



Es existiert kein augmentierender Pfad mehr (siehe Restnetzwerk).

Residualnetzwerk:



Damit beträgt der maximale Fluss $2 + 7 + 4 + 9 = 22$.

Aufgabe 3 (Eigenschaften von Flüssen): (2+3+4+2 Punkte)

Beweisen Sie die folgenden Aussagen über einen Fluss f und Flussnetzwerk $G(V, E, c)$ mit Quelle s und Senke t:

a) Für alle $X, Y \subseteq V$ gilt:

$$f(X, Y) = -f(Y, X)$$

b) Für alle $X, Y, Z \subseteq V$ mit $X \cap Y = \emptyset$ gilt:

$$f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$$

c) Sei (S,T) ein Schnitt über G, so gilt:

$$f(S,T) = |f|$$

d) Sei (S,T) ein Schnitt über G, so gilt:

$$|f| \leq c(S,T)$$

Lösung:

Die folgenden Aussagen waren zu beweisen:

a) Für alle $X, Y \subseteq V$ gilt: $f(X,Y) = -f(Y,X)$

$$\begin{aligned} f(X,Y) &= \sum_{x\in X}\sum_{y\in Y} f(x,y) \\ &= \sum_{x\in X}\sum_{y\in Y} -f(y,x) \\ &= -\sum_{x\in X}\sum_{y\in Y} f(y,x) \\ &= -\sum_{y\in Y}\sum_{x\in X} f(y,x) \\ &= -f(Y,X) \end{aligned}$$

b) Für alle $X, Y, Z \subseteq V$ mit $X \cap Y = \emptyset$ gilt:

$$f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$$

$$\begin{aligned} f(X \cup Y, Z) &= \sum_{x\in X\cup Y}\sum_{y\in Z} f(x,y) && \mid X \cap Y = \emptyset \\ &= \sum_{x\in X}\sum_{y\in Z} f(x,y) + \sum_{x\in Y}\sum_{y\in Z} f(x,y) \\ &= f(X,Z) + f(Y,Z) \end{aligned}$$

c) Sei (S,T) ein Schnitt über G, so gilt: $f(S,T) = |f|$

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,V-T) && \mid f(S,V) = f(S,T) + f(S,V-T) \text{ mit } S \cup T = V \\ &= f(S,V) - f(S,S) && \mid f(X,X) = 0 \\ &= f(S,V) && \mid f(X \cup Y, Z) = f(X,Z) + f(Y,Z) \\ &= f(s,V) + f(S-\{s\},V) && \mid \text{Flusserhaltung} \\ &= f(s,V) = |f| \end{aligned}$$

d) Sei (S,T) ein Schnitt über G, so gilt: $|f| \leq c(S,T)$

$$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{x \in S} \sum_{y \in T} f(x,y) \\ &\leq \sum_{x \in S} \sum_{y \in T} c(x,y) \\ &= c(S,T) \end{aligned}$$

Aufgabe 4 (Ausgabe des kürzesten Pfades): (5+3 Punkte)

a) Erweitern Sie die in der Vorlesung gegebene Implementierung von Floyd (Vorlesung 18, Folie 21), sodass jeweils die zugehörigen Vorgänger in einem zusätzlichen Array `int &V[][]` abgelegt werden (vgl. Folie 22). Ihre Methode soll die folgende Signatur besitzen:

```
void floydSP(double W[][], int n, double &D[][], int &V[][])
```

b) Geben Sie eine Implementierung der Methode `void pfadAusgeben(int &V[][], int start, int ziel)` an, die auf Basis des von der Methode aus **a)** berechneten Vorgängerarrays den kürzesten Pfad zwischen `start`- und `ziel`-Knoten ausgibt.

Lösung:

a)

```
void floydSP(double W[][], int n, double &D[][], int &V[][]) {
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
      D[i,j] = W[i,j]; // Kopiere W nach D
      if (W[i,j] < inf) {
        V[i,j] = i; // i ist letzter Knoten vor j
      } else {
        V[i,j] = -1; // bzw. null
      }
    }

  for (int i = 0; i < n; i++) {
    D[i,i] = 0;
    V[i,i] = -1; // ... ließe sich auch später erkennen ...
  }

  for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        if (D[i,j] > D[i,k] + D[k,j]) {
          D[i,j] = D[i,k] + D[k,j];
          // Der kürzeste Weg von i nach j muss zu einem
          // letzten 'k' von woaus er direkt nach j geht.
          V[i,j] = V[k,j];
        }
}
```

b)

```
void pfadAusgeben(int &V[][], int start, int ziel) {
  if (start == ziel) {
    ausgabe(start);
  } else if (V[start, ziel] == -1) {
    ausgabe("Es gibt keinen Pfad von "+start+" nach "+ziel+".");
  } else {
    pfadAusgeben(V, start, V[start, ziel]);
    ausgabe(" -> " + ziel);
  }
}
```

Prof. Dr. Ir. Joost-Pieter Katoen

Jonathan Heinen, Sabrina von Styp

Hinweise:

- Die Übungsblätter sind in Gruppen von je 3 Studierenden der gleichen Kleingruppenübung zu bearbeitet.
- Die Lösungen müssen bis Montag, den 26. Juli um 11:00 Uhr in die Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55).
- Namen und Matrikelnummern, sowie die Nummer der Übungsgruppe sind auf jedes Blatt zu schreiben.

Aufgabe 1 (Longest Common Subsequence): (5 Punkte)

Bestimmen Sie, entsprechend dem in der Vorlesung vorgestellten Verfahren, die Longest Common Subsequence der Wörter `WELTMEISTER` und `ELFMETER`. Geben Sie hierzu die berechnete Matrix an und kennzeichnen Sie den Pfad, der zur Rekonstruktion des Ergebnisses genutzt wird.

Lösung:

		W	E	L	T	M	E	I	S	T	E	R
	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	1	1	1	1	1	1	1	1	1	1
L	0	0	1	2	2	2	2	2	2	2	2	2
F	0	0	1	2	2	2	2	2	2	2	2	2
M	0	0	1	2	2	3	3	3	3	3	3	3
E	0	0	1	2	2	3	4	4	4	4	4	4
T	0	0	1	2	3	3	4	4	4	5	5	5
E	0	0	1	2	3	3	4	4	4	5	6	6
R	0	0	1	2	3	3	4	4	4	5	6	7

Die Längste Gemeinsame Teilsequenz `ELMETER` hat die Länge *sieben*.

Aufgabe 2 (Levenshtein-Distanz): (5 + 4 + 5 + 3 Punkte)

Die Levenshtein-Distanz zwischen zwei Zeichenketten ist die minimale Anzahl von Einfüge-, Lösch- und Ersetz-Operationen die nötig sind, um die eine Zeichenkette u in die andere v umzuwandeln.
Als Beispiel betrachten wir die Zeichenketten $u =$ `Tier` und $v =$ `Tor`. In einem ersten Schritt können wir u durch das Ersetzen von `e` durch `o` in `Tior` umwandeln. In einem zweiten Schritt erhalten wir dann durch Löschen des Zeichen `i` das Wort $v =$ `Tor`. Da es keine kürzeren Umwandlungssequenzen gibt beträgt die Levenshtein-Distanz zwischen `Tier` und `Tor` 2.
Wie lässt sich die Levenshtein-Distanz zwischen zwei beliebigen Zeichenketten berechnen?

a) Geben Sie eine Rekursionsgleichung zur Berechnung der Levenshtein-Distanz zwischen den Zeichenketten u und v an.

b) Nutzen Sie die Rekursionsgleichung aus **a)** bottom-up, um eine $|u| \times |v|$ Matrix für die Überführung von `RHABARBER` in `BARBAREI` zu füllen und lesen Sie aus dieser die Levenshtein-Distanz ab.

c) Nutzen Sie die Rekursionsgleichung aus **a)** für die Implementierung der Methode `levenshtein(char u[n], char v[m], int n, int m)`, die – entsprechend dem Verfahren aus **b)** – die Levenshtein-Distanz zwischen den Zeichenketten u und v bestimmt.

d) Bestimmen Sie die Laufzeit der von Ihnen erstellten Methode in Abhängigkeit von n und m, den Längen der beiden Zeichenketten.

Lösung:

a) Es ergibt sich die folgende Rekursionsgleichung:

$$D[i,j] = \begin{cases} j & i = 0 \\ i & j = 0 \\ D[i-1, j-1] & a_i = b_j \\ min\{D[i-1,j-1], D[i-1,j], D[i,j-1]\} + 1 & sonst \end{cases}$$

b) Unter Nutzung der Rekursionsgleichung aus **a)** ergibt sich die folgende Matrix:

		R	H	A	B	A	R	B	E	R
	0	1	2	3	4	5	6	7	8	9
B	1	1	2	3	3	4	5	6	7	8
A	2	2	2	2	3	3	4	5	6	7
R	3	2	3	3	3	4	3	4	5	6
B	4	3	3	4	3	4	4	3	4	5
A	5	4	4	3	4	3	4	4	4	5
R	6	5	5	4	4	4	3	4	5	4
E	7	6	6	5	5	5	4	4	4	5
I	8	7	7	6	6	6	5	5	5	5

Es werden *fünf* Schritte benötigt um das Wort `RHABARBER` in das Wort `BARBAREI` umzuwandeln. Aus dem oben gegebenen Pfad (einer von mehreren möglichen) lassen sich die folgenden Schritte ablesen:

1. Wandel den ersten Buchstaben `R` in ein `B` um: `BHABARBER`
2. Lösche den zweiten Buchstaben: `BABARBER`
3. Füge ein `R` als dritten Buchstaben ein: `BARBARBER`
4. Lösche den siebten Buchstaben `B`: `BARBARER`
5. Wandle den letzten Buchstaben `R` in ein `I` um: `BARBAREI`

c) Die folgende Java Methode implementiert Levenshtein basierend auf dynamischer Programmierung:

```
public static int levenshtein(char[n] A, int n, char[m] B, int m){

    int[][] matrix = new int[n+1][m+1];

    for(int i = 0; i <= n; i++)
        for(int j = 0; j <= m; j++)

            if(i == 0)
                matrix[i, j] = j;

            else if(j == 0)
                matrix[i, j] = i;

            else if(A[i-1]) == B[j-1])
                matrix[i, j] = matrix[i-1, j-1];

            else
                matrix[i, j] = min(matrix[i-1, j-1], matrix[i-1, j],
                      matrix[i, j-1]) + 1;

    return matrix[n][m];
}
```

d) Die Matrix hat die Größe $(n+1) \times (m+1)$. Für jeden Eintrag der Matrix wird konstanter Aufwand benötigt, daher hat der Algorithmus eine Laufzeit von $O(n \cdot m)$.

Aufgabe 3 (Graham-Algorithmus): (2 + 4 Punkte)

Gegeben seien die folgenden Punkte:

$$(12,4),\ (7,7),\ (3,13),\ (6,13),\ (4,2),\ (4,14),\ (9,10),\ (4,13),\ (8,12),\ (2,12)$$

a) Bestimmen Sie den Punkt p_1 mit der minimalsten x-Koordinate und sortieren sie die restlichen Punkte nach aufsteigendem Polarwinkel.

b) Bestimmen Sie, mit Hilfe des Graham-Algorithmus die konvexe Hülle der oben angegebenen Punkte. Geben Sie alle berechneten Determinanten an, sowie den jeweils aktuellen Zustand des Stacks.

Lösung:

a) Der Punkt mit der niedrigsten x-Koordinate ist der Punkt $p_1 = (2,12)$. Wir sortieren die restlichen Punkte nach aufsteigenden Polarwinkel, z.B. mit Hilfe von Insertionsort:

Wir fügen als erstes $(12,4)$ in die noch leere Liste der Punkte. Dann betrachten wir Punkt $(7,7)$ und bestimmen:

$$det((7,7)-(2,12),(12,4)-(2,12)) = det\begin{pmatrix} 5 & 10 \\ -5 & -8 \end{pmatrix} = 10.$$

Da der Wert positiv ist befindet sich $(7,7)$ rechts von $(12,4)$ und wir müssen ihn somit vor $(12,4)$ einfügen.

Fügen wir alle Punkte entsprechend ein erhalten wir die folgende Reihenfolge:

$$(4,2),\ (7,7),\ (12,4),\ (9,10),\ (8,12),\ (6,13),\ (4,13),\ (4,14)$$

b) Zu Beginn legen wir die ersten drei Punkte auf den Stack:

$(2,12),\ (4,2),\ (7,7)$

Wir wissen, dass die Punkte $(2,12)$ und $(4,2)$ auf jeden Fall auf der konvexen Hülle liegen. Somit müssen wir an diesem Punkt eigentlich noch keine Überprüfung durchführen. Wir werden dennoch die Richtung testen:

$$det((4,2)-(2,12),(7,7)-(4,2))det\begin{pmatrix} 2 & 3 \\ -10 & -5 \end{pmatrix} = 2 \cdot 5 \ - \ -10 \cdot 3 = 40$$

Wie zu erwarten war ist der Wert positiv und somit bilden die beiden Strecken einen links-Knick. Als nächstes fügen wir den Punkt $(12,4)$ hinzu:

$(2,12),\ (4,2),\ (7,7),\ (12,4)$

$$det((7,7)-(4,2),(12,4)-(7,7)) = det\begin{pmatrix} 3 & 5 \\ 5 & -3 \end{pmatrix} = -34$$

wir entfernen den vorletzten Punkt aus dem Stack, da dieser nicht zur Hülle gehört:

$(2,12),\ (4,2),\ (12,4)$

Eine Überprüfung der ersten drei Punkte ist nicht nötig, da die ersten beiden Punkte auf jeden Fall auf der Hülle liegen. Dennoch hier die Berechnung:

$$det \begin{pmatrix} 2 & 8 \\ -10 & 2 \end{pmatrix} = 84$$

Wir fügen den Punkt $(9, 10)$ hinzu:

$(2, 12),\ (4, 2),\ (12, 4),\ (9, 10)$

$$det \begin{pmatrix} 8 & -3 \\ 2 & 6 \end{pmatrix} = 54$$

Wir fügen den Punkt $(8, 12)$ hinzu:

$(2, 12),\ (4, 2),\ (12, 4),\ (9, 10),\ (8, 12)$

$$det \begin{pmatrix} -3 & -1 \\ 6 & 2 \end{pmatrix} = 0$$

Die Punkte $(12, 4), (9, 10)$ und $(8, 12)$ bilden eine Linie. Somit könnten wir den Punkt $(9, 10)$ eigentlich entfernen, da er überflüssig ist. Jedoch wird er laut Algorithmus der Vorlesung beibehalten.

Wir fügen den Punkt $(6, 13)$ hinzu:

$(2, 12),\ (4, 2),\ (12, 4),\ (9, 10),\ (8, 12),\ (6, 13)$

$$det \begin{pmatrix} -1 & -2 \\ 2 & 1 \end{pmatrix} = 3$$

Wir fügen den Punkt $(4, 13)$ hinzu:

$(2, 12),\ (4, 2),\ (12, 4),\ (9, 10),\ (8, 12),\ (6, 13),\ (4, 13)$

$$det \begin{pmatrix} -2 & -2 \\ 1 & 0 \end{pmatrix} = 2$$

Wir fügen den Punkt $(4, 14)$ hinzu:

$(2, 12),\ (4, 2),\ (12, 4),\ (9, 10),\ (8, 12),\ (6, 13),\ (4, 13),\ (4, 14)$

$$det \begin{pmatrix} -2 & 0 \\ 0 & 1 \end{pmatrix} = -2$$

Der Punkt $(4, 13)$ liegt nicht auf der konvexen Hülle. Wir entfernen ihn vom Stack:

$(2, 12),\ (4, 2),\ (12, 4),\ (9, 10),\ (8, 12),\ (6, 13),\ (4, 14)$

$$det \begin{pmatrix} -2 & -2 \\ 1 & 1 \end{pmatrix} = 0$$

Es ergibt sich das folgende konvexe Polygon:

$$(2, 12),\ (4, 2),\ (12, 4),\ (9, 10),\ (8, 12),\ (6, 13),\ (4, 14)$$

Aufgabe 4 (Ghostbusters): (5 + 5 Punkte)

Ein Gruppe von n Geisterjägern kämpft gegen n Geister. Jeder Geisterjäger ist mit einem Protonenwerfer bewaffnet. Der Strahl eines Protonenwerfers verläuft auf einer geraden Linie und bricht ab sobald er einen Geist trifft. Die Geisterjäger entscheiden, die Geister unter sich aufzuteilen, um dann gleichzeitig auf den jeweils zugeteilten Geist zu schiessen. Bekannterweise ist es sehr gefährlich, wenn sich Photonenstrahlen kreuzen, weshalb die Geisterjäger die Geister so unter sich aufteilen müssen, dass sich die Strahlen nicht kreuzen.
Gehen Sie davon aus, dass die Position jedes Geisterjägers, sowie jedes Geistes ein fester Punkt einer Ebene ist und keine drei Positionen auf einer gemeinsamen Gerade liegen.

a) Zeigen Sie, dass es eine Gerade gibt, die durch einen Geisterjäger und einen Geist verläuft, sodass auf jeder Seite die Anzahl der Geisterjäger gleich der Anzahl von Geistern ist.

Beschreiben Sie, wie eine solche Gerade in Zeit $O(n \log n)$ gefunden werden kann.

b) Geben Sie einen Algorithmus mit Laufzeit $O(n^2 \log n)$ an, der die Geister so auf die Geisterjäger aufteilt, dass sich die Protonenstrahlen nicht schneiden.

Lösung:

a) Im folgenden Algorithmus wird ein entsprechendes Paar von Geist und Jäger ermittelt. Hierzu wird zuerst das Objekt o_1 mit geringste x-Koordinate bestimmt. Dann werden die restlichen Objekte entsprechend dem Polarwinkel im Bezug auf dieses Objekt sortiert. Gibt es nun einen entsprechenden Partner zu $o1$, so Teilen diese die Menge der Objekte in zwei Partitionen wobei die eine alle Objekte beinhaltet, die sich in der Liste vor dem gefundenen Partner befinden und die andere die, die nach dem gefunden Objekt in der Liste stehen.

Wir durchlaufen nun die sortierte Liste und bestimmen jeweils die Differenz zwischen Geistern und Jägern. Ist die Differenz 0, so haben wir bereits gleich viele Geister wie Jäger gesehen und es handelt sich um einen gültigen Partner für o_1:

```
struct Object {
  int x;
  int y;
  int typ;
}

(Object, Object) findPair(Object[2*n] objects)
    // Finde Index des Punktes mit minimaler x Koordinate
    int refPnt = findRef(objects);
    // Sortiere Punkte nach aufsteigendem Polarwinkel bezüglich
    // refPnt, setze dabei refPnt an Position 0.
    list = polarSort(objects, refPnt);

    // Durchlaufe die sortierte Liste solange, bis
    // gleichviele Geister und Jäger gesehen wurden
    int count = 0;
    for( int i = 0; i < 2*n; i++){
        if(list[i].typ == GHOST)
            count++;
        else
            count--;

        if(count == 0)
            // Das zuletzt gesehene Objekt bildet mit dem
            // Referenzobjekt ein gesuchtes Paar
            return (list[0], list[i])
    }
```

Wir müssen noch sicherstellen, dass ein entsprechender Partner immer gefunden werden kann.

Ohne Beschränkung der Allgemeinheit nehmen wir an, dass es sich bei unser Referenzobjekt o_1 =`list[0]` um einen Jäger handelt. Wenn das erste (`list[1]`) oder letzte Element ein Geist ist, so sind diese offensichtlich gültige Partner, wobei wir zwei Partitionen erhalten bei der die eine leer ist und die andere alle weiteren Objekte enthält.

Ist sowohl das erste als auch das letzte Element kein Geist, so ist der Wert von `count` nach dem wir das erste Element gesehen haben -2 und bevor wir das letzte Element sehen 1, da wenn wir das letzte Element erreichen `count = 0` ist, da wir alle Objekte gesehen haben und somit auch gleich viele Geister wie Jäger.

Da wir in jedem Schritt den Wert von `count` nur um eins erhöhen oder verringern und der Wert von anfänglich -2 zum Schluss auf 1 geändert wird muss er zwischenzeitlich den Wert 0 annehmen.

Die Position mit minimaler x-Koordinate lässt sich in $O(n)$ finden. Die Sortierung der Polarwinkel ist in $O(n \cdot \log n)$ möglich. Für die Suche nach dem Partner zu g_1 muss die sortierte Folge lediglich einmal durchlaufen werden $O(n)$. Somit ergibt sich eine Gesamtlaufzeit von $O(n + n \cdot \log n + n) = O(n \cdot \log n)$

b) Der folgende Algorithmus bestimmt eine entsprechende Aufteilung indem ein Paar gesucht wird und dann rekursiv in beide Partitionen weitere Paare gesucht werden. Da die Anzahl der Geister und Geisterjäger in allen Partitionen jeweils gleich ist, und die Strahlen die Partition nicht verlassen (die Strahlen laufen nur bis zu den entsprechenden Geistern), kann so eine Aufteilung ohne schneidende Strahlen gefunden werden.

```
(Object, Object)[n] findPairs(Object[2*n] objects, int n)

    // Rekursionsabbruch bei nur einem Paar
    if(n == 0)
        return (Object, Object)[0];
    if(n == 1)
        return (objects[0], objects[1])[1];

    // Finde Index des Punktes mit minimaler x Koordinate
    int refPnt = findRef(objects);
    // Sortiere Punkte nach aufsteigendem Polarwinkel bezüglich
    // refPnt, setze dabei refPnt an Position 0.
    list = polarSort(objects, refPnt);

    // Durchlaufe die sortierte Liste solange bis
    // gleichviele Geister und Jäger gesehen wurden
    int count = 0;
    for(int i = 0; i < 2*n; i++){
        if(list[i].typ == GHOST)
            count++;
        else
            count--;

        if(count == 0){
            // Es wurden gleich viele Geister und Jäger gesehen.
            // Teile die enstandenen Partionen weiter in Paare auf.
            (Object, Object)[] result;
            result.add(list[0], list[i])
            result.add(findPairs(objects[1 .. i-1], i-1))
            result.add(findPairs(objects[i+1 .. 2*n], 2*n - i))
            return result;
        }
    }

    // Diese Zeile kann nie erreicht werden
    return null;
```

Der gegebene Algorithmus hat im Worst-Case (jeweils die ersten beiden Objekte der sortierten Liste bilden ein Paar) eine Laufzeit von $T(n) = T(n-1) + n \cdot \log n = n^2 \cdot \log n$.