Prof. Dr. Ir. Joost-Pieter Katoen Christian Dehnert, Jonathan Heinen, Thomas Ströder, Sabrina von Styp

Aufgabe 1 ($\mathcal{O}$-Notation): (6 + 6 + 8 = 20 Punkte)

a) Sortieren Sie für die unten gegebenen Funktionen die $\mathcal{O}$-Klassen $\mathcal{O}(a(n)), \mathcal{O}(b(n)), \mathcal{O}(c(n)), \mathcal{O}(d(n))$ und $\mathcal{O}(e(n))$ bezüglich ihrer Teilmengenbeziehung. Nutzen Sie ausschließlich die echte Teilmenge $\subset$ sowie die Gleichheit $=$ für die Beziehungen zwischen den Mengen. Folgendes Beispiel illustriert diese Schreibweise für einige Funktionen f_1 bis f_5 (diese haben nichts mit den unten angegebenen Funktionen zu tun):

$$\mathcal{O}(f_4(n)) \subset \mathcal{O}(f_3(n)) = \mathcal{O}(f_5(n)) \subset \mathcal{O}(f_1(n)) = \mathcal{O}(f_2(n))$$

Die angebenen Beziehungen müssen weder bewiesen noch begründet werden.

$a(n) = n^2 \cdot log_2 n + 42$ $\quad b(n) = 2^n + n^4$ $\quad c(n) = 2^{2 \cdot n}$ $\quad d(n) = 2^{n+3}$ $\quad e(n) = \sqrt{n^5}$

b) Beweisen oder widerlegen Sie $(\log_2 n)^2 \in O(n)$.

c) Beweisen oder widerlegen Sie $n! \in \Omega(n^{n-2})$.

Lösung:

a) Es gilt:

$$\mathcal{O}(a(n)) \subset \mathcal{O}(e(n)) \subset \mathcal{O}(b(n)) = \mathcal{O}(d(n)) \subset \mathcal{O}(c(n))$$

b) Die Aussage gilt.

Für $n \geq 16$ haben wir

$$\begin{array}{rrcl} & (\log_2 n)^2 & \leq & n \\ \Leftrightarrow & \log_2 n & \leq & \sqrt{n} \end{array}$$

und dies ist eine wahre Aussage für $n \geq 16$. Also gilt die Aussage mit $n_0 = 16$ und $c = 1$.

Alternative Lösung:

Es gilt

$$\begin{array}{rcl} \lim\limits_{n\to\infty} \frac{(\log_2 n)^2}{n} & \overset{L'Hospital}{=} & \lim\limits_{n\to\infty} \frac{2 \cdot \log_2 n \cdot \frac{1}{n \cdot \log(2)}}{1} \\ & = & \lim\limits_{n\to\infty} \frac{2 \cdot \log_2 n}{n \cdot \log(2)} \\ & \overset{L'Hospital}{=} & \lim\limits_{n\to\infty} \frac{2 \cdot \frac{1}{n \cdot \log(2)}}{\log(2)} \\ & = & \lim\limits_{n\to\infty} \frac{2}{n \cdot (\log(2))^2} \\ & = & 0 \end{array}$$

und damit ist die Aussage gezeigt.

c) Die Aussage gilt nicht. Beweis durch Widerspruch:

Angenommen die Aussage gilt. Dann gilt laut Definition:

$$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. n! \geq c \cdot n^{n-2} \forall n > n_0$$

Wir formen die Aussage wie folgt um, um einen Widerspruch zu erhalten (wobei wir die Quantoren weglassen, um die Lesbarkeit zu erhöhen):

$$\begin{array}{lllll} & n! & \geq & c \cdot n^{n-2} & \\ \Leftrightarrow & 1 \cdot 2 \cdot 3 \cdot 4 \ldots \cdot n & \geq & c \cdot n^{n-2} & \\ \Rightarrow & 1 \cdot 2 \cdot 3 \cdot n^{n-3} & > & c \cdot n^{n-2} & \mid : n^{n-3} \\ \Leftrightarrow & 6 & > & c \cdot n & \end{array}$$

Seien nun $c \in \mathbb{R}^+$ und $n_0 \in \mathbb{N}$ beliebig. Dann gilt für alle $n \in \mathbb{N}$ mit $n > \max(n_0, \frac{6}{c})$, dass $6 < c \cdot n$. Widerspruch. Somit muss die Annahme falsch sein und die Aussage gilt nicht.

Aufgabe 2 (Sortieren): (6 + 3 + 3 + 8 = 20 Punkte)

a) Sortieren Sie das folgende Array mittels des Heapsort-Algorithmus aus der Vorlesung.

5	9	13	7	2	3

Geben Sie das vollständige Array nach jeder Versickerung (Heapify-Operation) an, bei der sich der Arrayinhalt ändert.

Hinweis: Es könnte hilfreich sein, die jeweils noch unsortierten Arraybereiche zusätzlich als Heap darzustellen. Dies ist jedoch optional.

b) Beweisen oder widerlegen Sie die folgende Aussage:

Heapsort ist stabil.

c) Geben Sie die asymptotische Best-Case Laufzeit (Θ) von Mergesort an. Begründen Sie Ihre Antwort (ein Verweis darauf, dass dies in der Vorlesung gezeigt wurde, reicht nicht aus).

d) Bestimmen Sie die Best-Case Laufzeit (Θ) des untenstehenden `natMergeSort` Sortieralgorithmus in Abhängigkeit der Arraylänge n und geben Sie an, ob dieser Algorithmus ein stabiler Sortieralgorithmus ist. Nehmen Sie dazu an, dass die Initialisierung des `R` Arrays in konstanter Zeit zu bewerkstelligen ist. Begründen Sie Ihre Antwort.

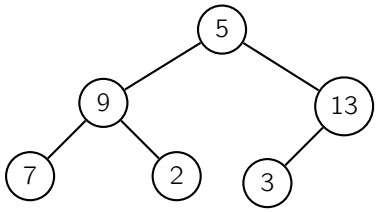
```
void natMergeSort(int E[], int n) { // Eingabearray und seine Laenge
    int R[] = 0; // initialisiere ein neues Array mit 0 Werten
    int r = 1;   // Index fuer R; R[0] bleibt 0, da r mit 1 beginnt
    for (int i = 0; i < n - 1; i++) {
        if (E[i] > E[i + 1]) { // falsche Reihenfolge?
            R[r] = i + 1;      // neues sortiertes Teilstueck gefunden
            r++;
        }
    }
    R[r] = n;
    sort(E, R, 0, r - 1);
}

// nahezu Mergesort
void sort(int E[], int R[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2; // finde Mitte
        sort(E, R, left, mid);        // sortiere linke Haelfte
        sort(E, R, mid + 1, right);   // sortiere rechte Haelfte
        // Verschmelzen der sortierten Haelften
        // merge Operation ist aus Vorlesung bekannt
        // erwartet ein Array und drei Positionen
        // fuegt die beiden Arrayabschnitte zwischen den Positionen
        // sortiert zusammen und setzt sie in das Originalarray ein
        merge(E, R[left], R[mid + 1] - 1, R[right + 1] - 1);
    }
}
```

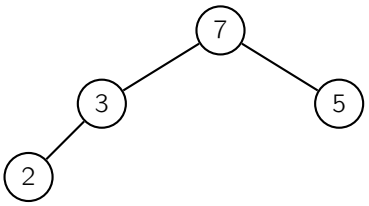
Hinweis: Sie dürfen Eigenschaften von aus der Vorlesung bekannten Sortierverfahren in Ihrer Begründung benutzen.

Lösung:

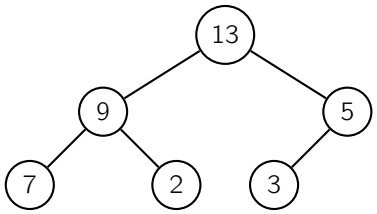
a)



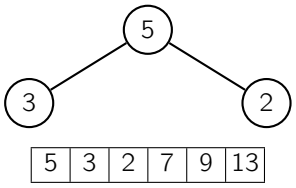
5	9	13	7	2	3



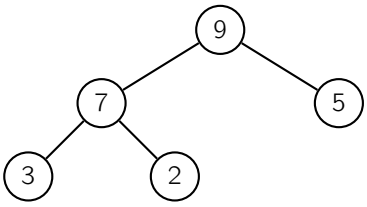
7	3	5	2	9	13



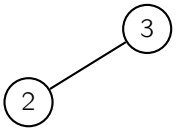
13	9	5	7	2	3



5	3	2	7	9	13



9	7	5	3	2	13



3	2	5	7	9	13

2	3	5	7	9	13

b) Heapsort ist nicht stabil. Betrachten wir folgendes Eingabearray, das nach seinen numerischen Einträgen sortiert werden soll:

1_A	1_B

Dieses Array entspricht bereits einem Heap, sodass der Heapaufbau keine Änderungen bewirkt. Dann wird jedoch das vorderste Element mit dem letzten getauscht und wir erhalten folgendes Array:

1_B	1_A

Nun ist nur noch ein Element im „unsortierten“ vorderen Arraybereich, welcher nicht mehr verändert wird. Das Ergebnis ist also letzteres Array und wir sehen, dass die beiden gleichwertigen Elemente ihre Position vertauscht haben. Dieses Gegenbeispiel zeigt, dass Heapsort nicht stabil ist.

c) Die Best-Case Laufzeit von Mergesort ist $\Theta(n \cdot \log n)$. Mergesort halbiert das Array unabhängig von der Reihenfolge der Arrayelemente, bis nur noch Arrays mit jeweils einem einzelnen Element übrig bleiben. Diese Halbierungen ergeben $\Theta(\log n)$ viele Ebenen mit disjunkten Arrayabschnitten. Für jede dieser Ebenen wird die `merge`-Operation ausgeführt, die lineare Laufzeit hat (auch unabhängig von der Reihenfolge der Arrayelemente). Insgesamt ergibt sich also für Best-, Average- und Worst-Case die gleiche asymptotische Laufzeit von $\Theta(n \cdot \log n)$.

d) Die Best-Case Laufzeit des `natMergeSort` Algorithmus ist $\Theta(n)$. Die Initialisierung des `R` Arrays hat nach Aufgabenstellung konstante Laufzeit. Die anschließende Schleife wird $(n-1)$-mal durchlaufen (also linear oft) und ihr Rumpf hat offensichtlich eine konstante Laufzeit. Diese Betrachtungen treffen unabhängig vom Arrayinhalt zu, womit wir im Best-Case bereits mindestens eine lineare Laufzeit haben. Im Best-Case ist das Array bereits aufsteigend sortiert und der Vergleich in der Schleife schlägt immer fehl. Demnach wird die Variable `r` nicht erhöht und `sort` wird so aufgerufen, dass die letzten beiden Argumente 0 sind. Damit schlägt der Vergleich in der `sort`-Operation fehl und der Algorithmus terminiert. Somit kommt im Best-Case nach der Initialisierung nur noch eine konstante Laufzeit hinzu, was zum oben angegebenen Ergebnis führt.

Der Algorithmus ist stabil, da im Vergleich zum normalen Mergesort lediglich andere Arrayabschnitte betrachtet werden. Die `merge`-Operation, welche die eigentliche Sortierung vornimmt, ist jedoch identisch zu der in Mergesort verwendeten. Damit folgt die Stabilität des `natMergeSort` Algorithmus aus der Stabilität von Mergesort, welche aus der Vorlesung bekannt ist.

Name: **Matrikelnummer:**

Aufgabe 3 (Bäume): (3 + 2 + 8 + 7 = 20 Punkte)

Gegeben sei der folgende Algorithmus für nicht leere, binäre Bäume:

```
class Node{
    Node left, right;
    int value;
}

void do(Node node){
    Node m = max(node);
    if(m.value > node.value){
    // swap der Werte von m und node
        int tmp = m.value;
        m.value = node.value;
        node.value = tmp
    }
    if(node.left != null)
        do(node.left);
    if(node.right != null)
        do(node.right);
}
```

```
Node max(Node node){
  Node max = node;
    if(node.left != null){
        node tmp = max(node.left);
        if(tmp.value > max.value)
            max = tmp;
    }
    if(node.right != null){
        node tmp = max(node.right);
        if(tmp.value > max.value)
            max = tmp;
    }
    return max;
}
```

a) Beschreiben Sie in möglichst wenigen Worten die Auswirkung der Methode `do(tree)`.

b) Die Laufzeit der Methode `max(tree)` ist für sämtliche Eingaben linear in n, der Anzahl von Knoten im übergebenen Baum `tree`. Begründen Sie kurz, warum `max(tree)` eine lineare Laufzeit hat.

c) Geben Sie in Abhängigkeit von n, der Anzahl von Knoten im übergebenen Baum `tree`, jeweils eine Rekursionsgleichung für die asymptotische Best- ($B(n)$) und Worst-Case Laufzeit ($W(n)$) des Aufrufs `do(tree)` sowie die entsprechende Komplexitätsklasse (Θ) an. Begründen Sie Ihre Antwort.

Hinweis: Überlegen Sie, ob die Struktur des übergebenen Baumes Einfluss auf die Laufzeit hat.
Die lineare Laufzeit von `max(tree)` darf vorausgesetzt werden.

d) Beweisen sie per Induktion die folgende Aussage: Ein Rot-Schwarz-Baum der Schwarzhöhe h hat maximal $rot(h) = \frac{2}{3} \cdot 4^h - \frac{2}{3}$ rote Knoten.

Lösung:

a) Sie überführt den gegebenen Baum in einen Baum mit gleichen Werten, der jedoch die Heap-Eigenschaft erfüllt, d. h. der Wert eines Knotens ist größer oder gleich allen Werten innerhalb des Teilbaumes, der an diesem Knoten beginnt.

b) Die Methode `max(n)` wird für jeden Knoten des Baumes exakt einmal aufgerufen. Die Kosten pro Aufruf sind konstant. Entsprechend ergeben sich lineare Kosten.

c) Die Komplexität des Aufrufs `do(Node n)` ergibt sich aus dem konstanten Aufwand der Methode, den linearen Kosten des Aufrufs `max(n)` sowie den zwei rekursiven Aufrufen von `do` für den rechten sowie linken Nachfolger. Die rekursiven Aufrufe sind abhängig von dem Aufbau des Baumes:

Im Best-Case ist der übergebene Baum balanciert. In diesem Fall ergeben sich zwei Aufrufe mit jeweils $\frac{n-1}{2}$ Knoten. Im Worst-Case ist der Baum lediglich eine Liste. In diesem Fall ergibt sich ein Aufruf mit $n-1$ Knoten.

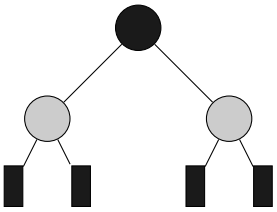
$B(n) = 2 \cdot B(n/2) + n \Rightarrow B(n) \in \Theta(n \cdot log_2 n)$ (Zweiter Fall des Master-Theorems)

$W(n) = B(n-1) + n \Rightarrow W(n) \in \Theta(n^2)$

d) Beweis durch vollständige Induktion über die Schwarzhöhe h.

Induktionsverankerung für $h = 1$:

Der RBT mit Schwarzhöhe $h = 1$ mit maximaler Anzahl an roten Knoten ist der folgende:



Dieser Baum besitzt zwei rote Knoten. Ebenfalls ergibt sich $rot(1) = \frac{2}{3} \cdot 4^1 - \frac{2}{3} = 2$.

Induktionsverankerung für $h = 0$ (alternativ):

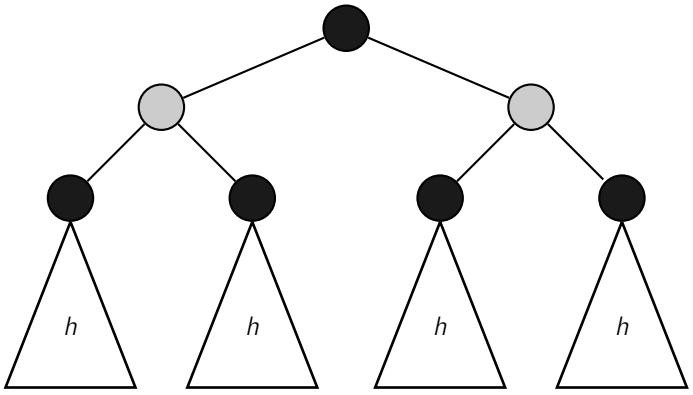
Es gibt nur einen RBT mit Schwarzhöhe $h = 0$. Dieser besteht nur aus der Wurzel. Da diese Schwarz ist enthält er keine roten Knoten. Aus der zu beweisenden Formel ergibt sich ebenfalls $rot(0) = \frac{2}{3} \cdot 4^0 - \frac{2}{3} = 0$.

Induktionsvoraussetzung:

In einem RBT der Schwarzhöhe h gibt es $rot(h) = \frac{2}{3} \cdot 4^h - \frac{2}{3}$ viele rote Knoten.

Induktionsschritt von h nach $h + 1$:

Um einen Baum mit maximaler Anzahl an roten Knoten zu erhalten, müssen die beiden Kinder des Wurzelknotens rot sein. Wäre dies nicht der Fall, so könnten wir einen roten Knoten dazwischen packen, ohne die Schwarzhöhe zu verändern. Um einen Baum der Schwarzhöhe $h + 1$ mit maximaler Anzahl an Rotknoten zu erhalten, bilden wir den folgenden Baum:



Die Anzahl der roten Knoten in diesem Baum berechnet sich nun wie folgt:

$$
\begin{aligned}
rot(h+1) &= \overbrace{4 \cdot rot(h)}^{\text{rote Knoten in den Teilbäumen}} + \overbrace{2}^{\text{Knoten auf der 1. Ebene}} \\
&\overset{I.V.}{=} 4 \cdot (\tfrac{2}{3} \cdot 4^h - \tfrac{2}{3}) + 2 \\
&= \tfrac{2}{3} \cdot 4^{h+1} - \tfrac{8}{3} + \tfrac{6}{3} \\
&= \tfrac{2}{3} \cdot 4^{h+1} - \tfrac{2}{3}
\end{aligned}
$$

Somit ist die Aussage bewiesen.

Name: **Matrikelnummer:**

Aufgabe 4 (Hashing): (4 + 4 + 5 + 5 + 2 = 20 Punkte)

Gegeben seien die Zahlen 43, 13, 61, 41, 53, 95 und 31 und die Hashfunktion $h(x) = x \mod 10$ sowie eine Hashtabelle mit 10 Plätzen. Füge Sie die Elemente mit Hilfe folgender Hashverfahren ein.

a) offene Adressierung mit linearem Sondieren,

b) offene Adressierung mit doppeltem Hashing und folgender zweiter Hashfunktion: $h'(x) = 7 - (x \mod 7)$.

c) Ein weiteres Hashverfahren ist Brent-Hashing. Im Gegensatz zum Hashing aus der Vorlesung wird bei einer Kollision im Brent-Hashing die nächste Sondierungsposition für beide Elemente berechnet und das alte Element genau dann verschoben, wenn dies einen Sondierungsschritt und das neue Element mehr als einen (weiteren) braucht.

Seien h_1 und h_2 Hashfunktionen, so ist $h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m$ die Hashfunktion, die man beim Brent-Hashing benutzt.

Brent-Hashing für ein Element k_1 im i-ten Sondierungsschritt funktioniert nun wie folgt.

- Ist $h(k_1, i)$ frei, so füge k_1 an dieser Position ein.
- Ist $h(k_1, i)$ belegt mit einem Element k_2, $h(k_1, i+1)$ ebenfalls belegt, k_2 wurde mit j Sondierungsschritten eingefügt (d. h. $h(k_1, i) = h(k_2, j)$) und $h(k_2, j+1)$ ist frei, so füge k_2 an der Position $h(k_2, j+1)$ ein und k_1 an der bisherigen Position von k_2.
- Ansonsten fahre mit der Sondierung für k_1 und $i+1$ fort.

Gegeben sei nun eine Hashtabelle mit 11 Plätzen und die Hashfunktion

$$h(k, i) = ((k \mod 11) + i \cdot (k \mod 7)) \mod 11.$$

Fügen Sie die folgenden Werte mittels Brent-Hashing ein: 24, 22, 48, 68 und 59.

d) Vervollständigen Sie den nachfolgenden Algorithmus `brentInsert`, der ein Element `k` nach Brent-Hashing in eine Hashtabelle `table` der Länge `m` einfügt. Die dabei verwendete Datenstruktur `Hash` hat zwei Felder: `key` vom Typ `int` und `state`, welches die Werte `free`, `used` und `deleted` annehmen kann. Gehen Sie davon aus, dass auf die Hashfunktionen h_1 und h_2 mit entsprechenden Funktionen `int h1(int value)` und `int h2(int value)` zugegriffen werden kann. Die zu vervollständigenden Teile sind durch Unterstreichungen markiert.

e) Welche Änderungen müssen an der normalen Suche bei offener Adressierung (Algorithmus `hashSearch` aus der Vorlesung) vorgenommen werden, um für Brent-Hashing korrekt zu funktionieren?

Lösung:

a)

0	1	2	3	4	5	6	7	8	9
	61	41	43	13	53	95	31		

b)

0	1	2	3	4	5	6	7	8	9
	61	41	43	13	95	53			31

c)

0	1	2	3	4	5	6	7	8	9	10
22		~~24~~		~~48~~	24					48
		68		59						

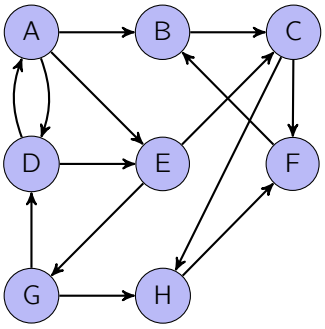
d)
```
void brentInsert(Hash table[], int m, int k) {
    int insertPos = h1(k);
    while (table[insertPos].state == used) {
        int newNext = (insertPos + h2(k)) mod m;
        int oldNext = (insertPos + h2(table[insertPos].key)) mod m;
        if (table[newNext].state == free ||
            table[oldNext].state == used) {
            insertPos = newNext;
        } else {
            table[oldNext].key = table[insertPos].key;
            table[oldNext].state = used;
            table[insertPos].state = deleted;
        }
    }
    table[insertPos].key = k;
    table[insertPos].state = used;
}
```

e) Suchen ist der gleiche Algorithmus wie aus der Vorlesung

```
int hashSearch(int T[], int key) {
  for (int i = 0; i < T.length; i++) {
    int pos = h(key, i); // Berechne i-te Sondierung
    if (T[pos] == key) { // Schlüssel k gefunden
      return T[pos];
    } else if (!T[pos]) { // freier Platz, nicht gefunden
      break;
    }
  }
  return -1; // "nicht gefunden"
}
```

Aufgabe 5 (Graphen): (2 + 6 + 7 + 5 = 20 Punkte)

a) Betrachten Sie den folgenden gerichteten Graphen G_1:

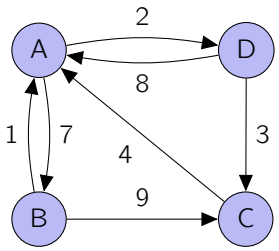


Geben Sie den Kondensationsgraphen $G_1 \downarrow$ an. Beschriften Sie die Knoten im Kondensationsgraphen mit den Namen aller Knoten, die zur jeweiligen starken Zusammenhangskomponente gehören. Bilden beispielsweise die Knoten 1 und 3 eine starke Zusammenhangskomponente, so sieht der zugehörige Knoten im Kondensationsgraphen wie folgt aus:

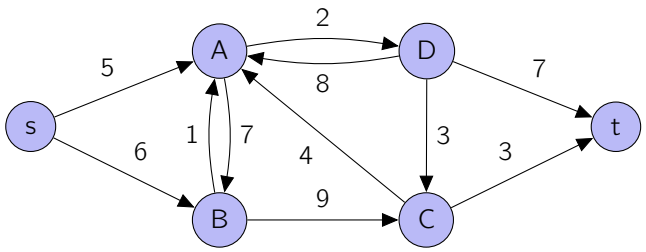
b) Beweisen oder widerlegen Sie die folgende Aussage:

Für jeden zusammenhängenden, ungerichteten, gewichteten Graphen $G = (V, E)$ mit nicht-negativen Kantengewichten gibt es einen Knoten $v \in V$, sodass der SSSP-Baum von v in G ein minimaler Spannbaum von G ist.

c) Bestimmen Sie für den folgenden Graphen G_2 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 fest vorgegeben.

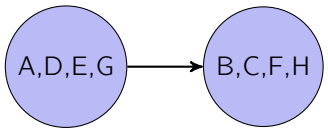


d) Geben Sie einen minimalen Schnitt und den Wert eines maximalen Flusses für folgendes Flussnetzwerk G_3 an.

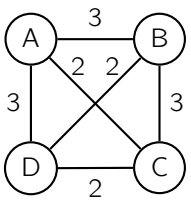


Lösung:

a)



b) Die Aussage ist falsch. Betrachten wir den folgenden Graphen G:



Der minimale Spannbaum enthält die Kanten $\{A, C\}$, $\{C, D\}$ und $\{B, D\}$ mit einem Gesamtgewicht von 6. Jeder SSSP-Baum in G benutzt jedoch mindestens eine Kante mit dem Gewicht 3, sodass sein Gesamtgewicht mindestens 7 beträgt. Dieses Gegenbeispiel widerlegt die Aussage.

c)

	A	B	C	D
A	0	7	∞	2
B	1	0	9	∞
C	4	∞	0	∞
D	8	∞	3	0

	A	B	C	D
A	0	7	16	2
B	1	0	9	3
C	4	11	0	6
D	8	15	3	0

	A	B	C	D
A	0	7	5	2
B	1	0	6	3
C	4	11	0	6
D	7	14	3	0

	A	B	C	D
A	0	7	∞	2
B	1	0	9	3
C	4	11	0	6
D	8	15	3	0

	A	B	C	D
A	0	7	16	2
B	1	0	9	3
C	4	11	0	6
D	7	14	3	0

d) Der minimale Schnitt ist $(\{s, A, B, C\}, \{D, t\})$ und der Wert eines maximalen Flusses ist 5.

Aufgabe 6 (Dynamische Programmierung): (1 + 3 + 8 + 8 = 20 Punkte)

Gesucht ist ein Algorithmus zur Bestimmung der kleinsten Anzahl an Münzen, die nötig sind, um einen bestimmten Geldbetrag zu zahlen. Hierzu werden verschiedene Münzwerte $m_1, m_2, \ldots, m_k$ mit $k > 0$ und $m_i \geq 1$ sowie der zu zahlende Betrag B gegeben.

Beispiel:
Seien die Münzwerte 1, 2 und 5 sowie der Betrag 9 gegeben, so ist die kleinste Anzahl an Münzen 3, nämlich zwei Münzen des Werts 2 sowie eine mit dem Wert 5.

Der folgende Algorithmus soll die Aufgabe übernehmen:

```
int minimum(int m[], int k, int b) {
    sort(m); // sortiert die Werte absteigend
    int n = 0, c = 0;
    while (b > 0) {
        if (b < m[n]) {
            n++;
            if (n >= k) {
                return -1;
            }
        } else {
            b = b - m[n];
            c++;
        }
    }
    return c;
}
```

a) Auf welchem Prinzip beruht der gegebene Algorithmus?

b) Geben Sie ein Beispiel an, bei dem der obige Algorithmus nicht die optimale Lösung findet.

c) Geben Sie eine rekursive Gleichung für $C(i, b)$ an, wobei $C(i, b)$ die minimale Anzahl an Münzen ist, die benötigt wird, um den Betrag b mit den Münzwerten m_1 bis m_i zu bezahlen.

Beachten Sie, dass nicht unbedingt alle Beträge zahlbar sind (z. B. wenn es keinen Münzwert 1 gibt). Diese Fälle sollen durch den Wert ∞ repräsentiert werden.

d) Für zwei Buchstabensequenzen $a = a_1 a_2 \ldots a_n$ und $b = b_1 b_2 \ldots b_m$ lässt sich anhand der folgenden Rekursionsgleichung die Levenshtein-Distanz zwischen den Teilsequenzen $a_1 a_2 \ldots a_i$ und $b_1 b_2 \ldots b_j$ bestimmen:

$$D(i,j) = \begin{cases} j & \text{falls } i = 0 \\ i & \text{falls } j = 0 \\ D(i-1, j-1) & \text{falls } a_i = b_j \\ \min(\ D(i-1,j-1),\ D(i-1,j),\ D(i,j-1)\ ) + 1 & \text{sonst} \end{cases}$$

Nutzen Sie die oben gegebene Rekursionsgleichung, um gemäß dem Prinzip der dynamischen Programmierung die folgende Tabelle zu füllen, und geben Sie die resultierende Levenshtein-Distanz der Buchstabensequenzen AACHEN und ATHEN an.

Lösung:

a) Es handelt sich um einen Greedy-Algorithmus.

b) Gegeben seien die Münzwerte $m_1 = 2$, $m_2 = 3$ sowie der Betrag $B = 4$. Der Algorithmus wählt im ersten Durchlauf den Münzwert $m_2 = 3$. Nun kann keine weitere Münze hinzugenommen werden, ohne den Betrag zu überschreiten. Somit scheitert der Algorithmus und gibt -1 zurück, obwohl der Betrag durch zweifaches Wählen von m_1 hätte erreicht werden können.

c) **Basisfälle:** $i = 0 \vee b = 0$

Im ersten Basisfall $b = 0$ ist der Betrag trivialerweise mit 0 Münzen zu bezahlen.

$$C(i, 0) = 0$$

Im zweiten Basisfall $i = 0, b > 0$ stehen keine Münzwerte zur Verfügung, jedoch ist ein Betrag größer 0 zu bezahlen, d. h. wir können den Betrag nicht bezahlen.

$$C(0, b) = \infty, \textit{ falls } b > 0$$

Standardfälle: $0 < i \leq k \wedge 0 < b \leq B$

Wir müssen zwei Fälle unterscheiden. Ist m_i größer als der Betrag b, so können wir diese Münze nicht nutzen und wir betrachten daher die minimale Anzahl an Münzen, die wir ohne den Münzwert m_i erreichen können:

$$C(i, b) = C(i - 1, b), \textit{ falls } i, b > 0, m_i > b$$

Ist $m_i \leq b$, können wir entscheiden, ob wir m_i nutzen wollen oder nicht. Nutzen wir diesen Wert nicht, so erhalten wir wiederum $C(i - 1, b)$. Nutzen wir eine Münze, so müssen wir lediglich den Restbetrag durch weitere Münzen (möglicherweise mit demselben Wert wie bei der gerade verwendeten Münze) abdecken und erhalten $1 + C(i, b - m_i)$. Um die minimale Anzahl von Münzen zu erhalten, wählen wir das Minimum aus den beiden Alternativen.

$$C(i, b) = min(C(i - 1, b), 1 + C(i, b - m_i)), \textit{ falls } i, b > 0, m_i \leq b$$

Somit ergibt sich die folgende Rekursionsgleichung:

$$C(i, b) = \begin{cases} 0 & \textit{falls } b = 0 \\ \infty & \textit{falls } i = 0, b > 0 \\ C(i - 1, b) & \textit{falls } m_i > b \\ min(\ C(i - 1, b), C(i, b - m_i) + 1) & \textit{sonst} \end{cases}$$

d)

		A	**A**	**C**	**H**	**E**	**N**
	0	1	2	3	4	5	6
A	1	0	1	2	3	4	5
T	2	1	1	2	3	4	5
H	3	2	2	2	2	3	4
E	4	3	3	3	3	2	3
N	5	4	4	4	4	3	2

Levenshtein-Distanz: 2