Aufgabe 1: Grundlagen in Java ($5 + 6 = 11$ Punkte)

(a) Die Klasse `Pair` soll zwei Elemente in einer festen Reihenfolge speichern. Die Funktion `swap()` innerhalb der Klasse soll die beiden Elemente der Klasse `Pair` vertauschen. Ihr Coding Assistent hat Ihnen den folgenden Vorschlag generiert. Notieren Sie, in welchen Zeilen sich Fehler in der Klasse `Pair` befinden und berichtigen Sie diese. Wenn Sie zur Berichtigung neue Zeilen einfügen, geben Sie an, zwischen welchen Zeilen diese eingefügt werden müssen.

```
1  public class Pair<E> {
2      private E first;
3      private E second;
4
5      public Pair(E first, E second) {
6          first = this.first;
7          second = this.second;
8      }
9
10     // Vertauscht die beiden Elemente
11     public int swap() {
12         this.first = this.second;
13         this.second = this.first;
14     }
15 }
```

Zeile(n) des Fehlers	Berichtigung
6	`this.first = first;`
7	`this.second = second;`
11	`public void swap)() {`
i 11–12	`E tmp = this.first;`
13	`this.second = tmp;`

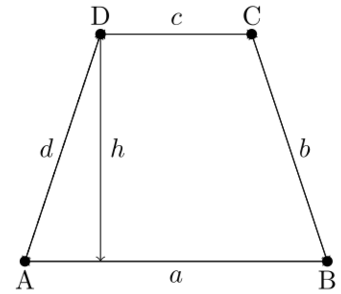
Zeile 6+7 `this.` auf falscher Seite der Zuweisung; `swap()` überschreibt `second` mit sich selbst, da `first` auf `second` zugewiesen wurde; `int swap()` hat keinen `return`

(b) Die Klasse `Trapezoid` repräsentiert ein Trapez, wobei die Kanten des Trapezes als `Edge` gespeichert sind.

Ihr Coding Assistent hat Ihnen die Methoden zur Berechnung des Flächeninhalts `area` und des Umfangs `perimeter` vorgeschlagen. Sie können davon ausgehen, dass alle Funktionsaufrufe innerhalb der Methoden korrekt sind und funktionieren. `length()` gibt hierbei Länge einer Kante zurück und hat eine konstante Laufzeit.

Hinweis: Der Flächeninhalt eines Trapezes kann mit folgender Formel berechnet werden:

$A = \frac{1}{2} \cdot h \cdot (a + c)$ mit der Höhe h und den Längen a, c der parallelen Seiten.




```
1  public class Trapezoid {
2      private Edge parallelEdgeA;
3      private Edge edgeB;
4      private Edge parallelEdgeC;
5      private Edge edgeD;
6      private double height;
7
8      public Trapezoid(Edge parallelEdgeA, Edge edgeB, Edge parallelEdgeC,
           Edge edgeD, double height) {
9          this.parallelEdgeA = parallelEdgeA;
10         this.edgeB = edgeB;
11         this.parallelEdgeC = parallelEdgeC;
12         this.edgeD = edgeD;
13         this.height = height;
14     }
15     //to evaluate:
16     public double area() {
17         double sideA = parallelEdgeA.length();
18         double sideC = parallelEdgeC.length();
19         return 0.5 * (sideA + sideC) * height;
20     }
21     //to evaluate:
22     public static double perimeter(Edge[] edges) {
23         double perimeter = 0.0;
24         for (Edge edge : edges) {
25             for (int i = 0; i < edges.length; i++) {
26                 perimeter += edge.length();
27             }
28         }
29         return perimeter;
30     }
31 }
```

Geben Sie die Größenordnung der Laufzeit von den Methoden `area` und `perimeter` mit den richtigen abhängigen Größen an. Geben Sie außerdem an, ob die jeweilige Methode funktioniert, also die Fläche bzw. den Umfang korrekt bestimmt. Begründen Sie kurz, falls eine Methode fehlerhaft ist.

`area()` ist korrekt, hat Laufzeit $O(1)$

`perimeter()` ist falsch (Zwei Schleifen iterieren über die Kanten: Output ist Vielfaches des Umfangs), hat Laufzeit $O(edges.length^2)$.

Aufgabe 2: Java-Programmierung ($2 + 7 + 2 = 11$ Punkte)

Sie haben hier die Klasse `MatrixImage`, welche das Interface `Image` implementiert. Die Klasse soll eine Bildrepräsentation als Integer-Array implementieren.

```
1  public interface Image {
2
3      int sizeX();
4      int sizeY();
5
6  }
7
8  public class MatrixImage implements Image {
9      public int[][] field;
10
11     /**
12      * A Constructor for the MatrixImage class
13      * @param sx the size on the x-axis
14      * @param sy the size on the y-axis
15      */
16     public MatrixImage(int sx, int sy) {
17         //TODO a)
18     }
19
20     /**
21      * A deep copy constructor for a given MatrixImage that
22      * @param that the to be copied MatrixImage
23      */
24     public MatrixImage(MatrixImage that) {
25         //TODO b)
26     }
27
28     /**
29      * @return the size on the x-axis
30      */
31     @Override
32     public int sizeX() {
33         return field.length;
34     }
35
36     /**
37      * @return the size on the y-axis
38      */
39     @Override
40     public int sizeY() {
41         return field[0].length;
42     }
43 }
```

(a) Ergänzen Sie den Konstruktor, der die `field`-Variable mit einer passenden Zuweisung initialisiert.

```
1 field = new int[sx][sy];
```

(b) Schreiben Sie nun den deep copy Konstruktor, der ein existierendes `MatrixImage` übergeben bekommt und die Werte so in ein neues Objekt kopiert, dass bei weiteren Änderungen an einem der beiden `MatrixImage`-Objekten nur das jeweilige Objekt betroffen bleibt. Zum Erstellen unabhängiger Kopien der Integer-Werte in den Arrays haben 1-dimensionale Arrays die Methode `clone()`. Sie können die Aufgabe aber auch ohne diese lösen.

```
1  public MatrixImage(MatrixImage that) {
2      this(that.sizeX(), that.sizeY());
3      for (int x = 0; x < sizeX(); x++) {
4          field[x] = that.field[x].clone();
5      }
6  }
7
8  Alternative:
9
10 public MatrixImage(MatrixImage that) {
11     field = new int[that.sizeX()][that.sizeY()];
12     for (int x = 0; x < sizeX(); x++) {
13             for (int y=0; y < sizeY(); y++) {
14                 field[x][y] = that.field[x][y];
15             }
16     }
17 }
```

(c) Warum ergibt das Konzept Sinn, Methoden in einem **Interface** zu definieren, wie die beiden `size`-Methoden in **Image**, und diese dann in der `MatrixImage`-Klasse zu implementieren? Antworten Sie in unter 5 Sätzen.

(So lange `MatrixImage` die einzige implementierende Variante bleibt, erhält man keine Vorteile. Die Klasse heißt jedoch relativ spezifisch und legt nahe, dass noch andere Varianten von `Image`-Klassen existieren könnnen.)
Sobald mehrere Klassen das `Interface` implementieren, können diese gemeinsame Standards einhalten und auch alle gemeinsame Schnittstellenbedingungen von `Image` erfüllen.

Aufgabe 3: Greedy Raumplanung ($4+4+4=12$ Punkte)

Zur Bestimmung der Belegung eines Konferenzraums sollen aus einer Liste von Raumbuchungsanfragen **möglichst viele** ausgewählt werden, die sich nicht überlappen. Dabei kann die Endzeit einer Buchung mit der Anfangszeit der folgenden Buchung übereinstimmen. Für die Auswahl wird der optimale Greedy Algorithmus für *Interval Scheduling* aus der Vorlesung genutzt.
Die ersten beiden Klassen sind **fehlerfrei**. Die Klasse `Interval` repräsentiert die Zeitintervalle der Buchungsanfragen. Die Klasse `IntervalCompareFinishTime` vergleicht Objekte der Klasse `Interval` auf Basis der Endzeitpunkte.

```
1  public class Interval {
2    private double s; //start
3    private double f; //finish
4    public Interval(double s, double f)
       {
5      this.s = s;
6      this.f = f;
7    }
8    public double start() { return s; }
9    public double finish() { return f; }
10 }
```

```
1 public class
     IntervalCompareFinishTime
     implements Comparator<Interval>
      {
2   @Override
3   public int compare(Interval i1,
       Interval i2) {
4     return Double.compare(i1.
         finish(), i2.finish());
5   }
6 }
```

Durch Fehler in der Klasse `IntervalScheduling` zeigt der Greedy Ansatz nicht das erwartete Verhalten. Die Importe wurden in allen Klassen dieser Aufgabe weggelassen. Sie können davon ausgehen, dass diese korrekt sind.

```
1  public abstract class IntervalScheduling {
2      public static Iterable<Interval> intervalScheduling(Iterable<Interval>
           intervals) {
3          PriorityQueue<Interval> pq =
4                  new PriorityQueue<>(new IntervalCompareFinishTime());
5          for (Interval i : intervals) {
6              pq.add(i);
7          }
8          Queue<Interval> queue = new LinkedList<>();
9          double finishTime = 0;
10         while (!pq.isEmpty()) {
11             Interval iv = pq.peek();
12             if (iv.start() >= finishTime) {
13                 queue.add(iv);
14             }
15         }
16         return queue;
17     }
18     public static void main(String[] args) {
19         Queue<Interval> intervals = new LinkedList<>();
20         intervals.add(new Interval(1, 2));
21         intervals.add(new Interval(2, 4));
22         intervals.add(new Interval(3, 5));
23         intervals.add(new Interval(4, 7));
24         intervals.add(new Interval(5, 6));
25         Iterable<Interval> jobs = intervalScheduling(intervals));
26     }
27 }
```

(a) Korrigieren Sie die Zeilen 8-16 der Klasse `IntervalScheduling`, sodass der Greedy Ansatz die richtigen Ergebnisse liefert.
Sie brauchen dabei nur die Zeilen aufzuschreiben, die berichtigt oder hinzugefügt werden müssen. Geben Sie dabei die Zeilennummern an. Wenn Sie zur Berichtigung neue Zeilen einfügen, geben Sie an, zwischen welchen Zeilen diese eingefügt werden müssen.
Hinweis: Es sind keine reinen Syntaxfehler im Code.

Zeile(n) des Fehlers	Berichtigung
11	`Interval iv = pq.poll()`
Einfügen 13-14	`finishTime = iv.finish();`

(b) Führen Sie den korrigierten Algorithmus mit dem in der `main` gegebenen Beispiel aus. Listen Sie die gewählten Intervalle auf!
Hinweis: Die PriorityQueue sortiert basierend auf der Klasse `IntervalCompareFinishTime`.

(1,2), (2,4), (5,6)

(c) Nun stehen genügend Räume zur Verfügung und es sollen alle Buchungsanfragen erfüllt werden. Durch den folgenden Pseudocode wird immer dann ein neuer Raum hinzugenommen, wenn es keinen Raum gibt, der aktuell die `if`-Abfrage in Zeile 12 erfüllen kann.

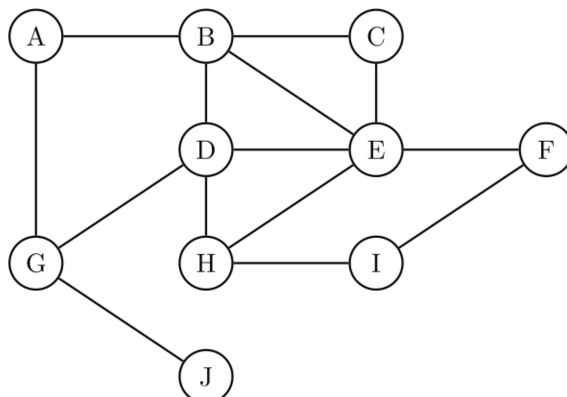
```
1  if Intervall iv ist kompatibel zu einem Raum R[k] für ein 0 ≤ k < K
2      füge iv zu R[k] hinzu, k minimal gewählt
3  else
4      initialisiere neuen Raum R[K]
5      füge iv zu R[K] hinzu
6      K++
```

Wie werden dadurch die Buchungsanfragen in der `main` auf die Räume aufgeteilt werden?

Raum 1: (1,2), (2,4), (5,6)
Raum 2: (3,5)
Raum 3: (4,7)

Aufgabe 4: Tiefensuche ($9 + 2 + 2 + 1 = 14$ Punkte)

Gegeben ist der folgende ungerichtete und ungewichtete Graph G:



(a) Führen Sie die Tiefensuche auf dem Graphen $\boldsymbol{G}$ aus. Beginnen Sie die Tiefensuche im **Knoten A**. Notieren Sie alle Knoten in der Reihenfolge, in der sie von der Tiefensuche gefunden werden (**Nebenreihenfolge/preorder**) und in der sie abgearbeitet werden, d.h. die **Hauptreihenfolge** bzw. **postorder**.

Gehen Sie dabei davon aus, dass bei Wahlmöglichkeit (Tiebreak) die benachbarten Knoten in **alphabetischer Reihenfolge** abgearbeitet werden.

Nebenreihenfolge: **A B C E D G J H I F**
Hauptreihenfolge: **J G F I H D E C B A**
Punkte:
Nebenreihenfolge: **(9 - Edit Distance) / 2**
Hauptreihenfolge: **(9 - Edit Distance) / 2**, beides mit Skript `afg_dfs.py` zu bestimmen

(b) Welche Kante müsste aus dem oben dargestellten Graphen $\boldsymbol{G}$ entfernt werden, damit ein anderer Knoten als **J** als erstes in der Hauptreihenfolge steht? Dabei soll der Graph weiterhin zusammenhängend bleiben.

Kante **D–G**
Weitere Möglichkeiten: **A–B**, **B–C**, **C–E**

(c) Erklären Sie, wie die reine Tiefensuche ergänzt werden muss, um einen **Pfad** vom Knoten s aus zu Knoten v zu bestimmen, wenn Knoten v vom Knoten s aus erreichbar ist. Geben Sie ebenfalls an, wie der Pfad dann bestimmt wird.

Man startet DFS in s. **(0.5 Punkte)** Man speichert, neben den besuchten Knoten `[marked]`, während der Tiefensuche für jeden Knoten, der noch nicht besucht wurde, den Vorgängerknoten, von dem der Knoten aus gefunden wurde `[parent]`. **(0.5 P)** Den Pfad bestimmt man dann, indem man: von dem Endknoten v den Einträgen des parent Array folgt, d.h. $w_{next} = \text{parent}[w]$ **(0.5 P)**, und dann den Pfad umkehrt. **(0.5 P)** Alternative Lösungsansätze, die gleiche Bedingungen erfüllen, werden auch gewertet mit Abzug für unerfüllte Schritte.

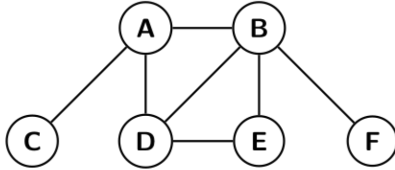
(d) Wenn ein solcher Pfad per Tiefensuche gefunden wurde, hat dieser dann immer minimale Länge? Wenn ja, begründen Sie. Wenn nein, geben Sie ein Gegenbeispiel an.

Nein, Graph aus der Aufgabe ist ein Gegenbeispiel: Zu Knoten D wird von A ein Weg der Länge 4 gefunden, obwohl es einen Weg mit 2 Kanten gibt.

Aufgabe 5: Radius eines Graphen ($3 + 2 + 6 + 3 = 14$ Punkte)

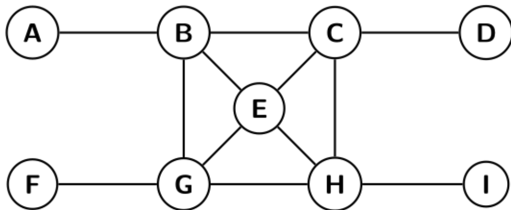
Zu einem zusammenhängenden, ungewichteten Graphen $G = (V, E)$ werden folgende Begriffe definiert: Die Anzahl der Kanten eines kürzestes Weges zwischen zwei Knoten v, w bezeichnen wir als Abstand $dist(v, w)$. Die *Exzentrizität* eines Knotens v ist als das Maximum der Abstände von v zu allen anderen Knoten des Graphens definiert $\epsilon(v) := \max\{dist(v, w) \mid w \in V\}$. Das Minimum der Exzentrizitäten aller Knoten in einem Graphen nennt man den *Radius* des Graphen $radius(G) := \min\{\epsilon(v) \mid v \in V\}$.

(a) Bestimmen Sie im folgenden Graphen die Exzentrizität der Knoten **A** und **C**.



$\epsilon(\mathbf{A}) = 2$ $\epsilon(\mathbf{C}) = 3$

(b) Markieren Sie den Knoten mit der niedrigsten Exzentrizitzät in dem folgenden Graphen.



E

(c) Geben Sie eine exakte Beschreibung (bzw. Pseudocode) eines Verfahrens an, um den **Radius** eines gegebenen Graphen zu bestimmen. Sie dürfen dabei die aus der Vorlesung bekannten Verfahren benutzen (genauen Namen des Algorithmus angeben!). Ihre Lösung sollte eine *worst case* Laufzeit in $\boldsymbol{O}(V(V + E))$ besitzen.

```
1  for all s < V
2    run BFS from s to determine dist(s,v) for all v < V
3    eps(s) = max of all dist values determined by BFS
4  end
5  radius = min of all eps values
6  return radius
```

(d) Begründen Sie die Laufzeit Ihres Algorithmus' aus (c). Verwenden Sie nach Möglichkeit die Voraussetzung, dass der Graph **zusammenhängend** ist, um eine bessere Abschätzung als $\boldsymbol{O}(V(V+E))$ zu erreichen.

Bei zusammenhängenden Graphen ist $E \geq V - 1$, also die Laufzeit von BFS in $\boldsymbol{O}(E)$. BFS wird für jeden der V Knoten aufgerufen, was zu einer Laufzeit in $\boldsymbol{O}(VE)$ führt. Die Berechnung des Maximums in Zeile 3 läuft über V Werte. Dies trägt nichts zur Gesamtlaufzeit bei, weil diese innerhalb der Schleife von BFS dominiert wird. Die Berechnung des Minimums in Zeile 5 läuft über V viele Werte und trägt nichts zur Gesamtlaufzeit bei, da sie nach der Schleife ausgeführt wird.

Bemerkungen:

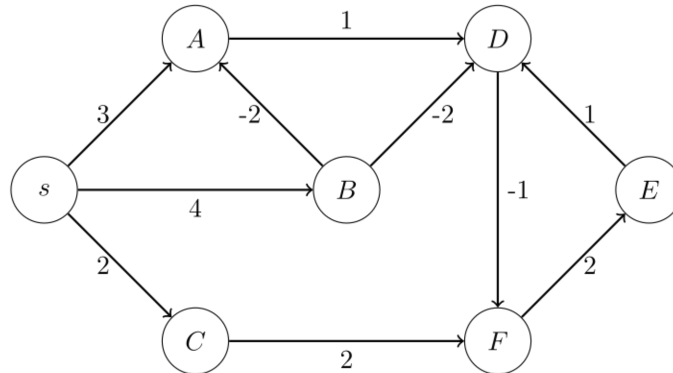
Die Exzentrizität eines Knotens kann auch innerhalb der BFS als der `dist`-Wert des zuletzt besuchten Knotens bestimmt werden.

Der Radius kann auch innerhalb der Schleife über die Startknoten aktualisiert werden.

Topologische Suche als SSSP Algorithmus kann nur bei *azyklischen* Graphen angewendet werden, was hier nicht vorausgesetzt wurde. Die anderen SSSP Algorithmen haben eine schlechtere Laufzeit. Zudem muss dabei angegeben werden, dass die Kantengewichte auf 1 gesetzt werden.

Aufgabe 6: SSSP $(7 + 2 = 9$ **Punkte)**

Gegeben ist der folgende gerichtete und gewichtete Graph G:



(a) Führen Sie den Bellman-Ford-Algorithmus mit Warteschlange aus der Vorlesung auf dem Graphen G aus. Beginnen Sie im Knoten s und notieren Sie die Knoten in der Reihenfolge, in der sie aus der Warteschlange entfernt werden. Geben Sie außerdem für jeden Knoten des Graphen, der vom Bellman-Ford-Algorithmus erfasst wird, die Distanz des kürzesten Pfades vom Startknoten aus zu dem jeweiligen Knoten an.

Gehen Sie dabei davon aus, dass bei Wahlmöglichkeit die Knoten in **alphabetischer Reihenfolge** in die Queue hinzugefügt werden.

Bewertet werden in der unten stehenden Tabelle nur die Spalten **Knoten** und **Distanz zum Startknoten**. Die Spalten *Iteration* und *Notizen* müssen nicht ausgefüllt werden, können aber zur Lösung der Aufgabe verwendet werden.
Hinweis: Streichen Sie keine Zeilen durch, die bewertet werden sollen.

Lösung 1 – Knoten in der Reihenfolge, in der sie aus der Warteschlange entfernt werden, wie in der Aufgabenstellung gefordert:

Iteration	**Knoten**	**Distanz zum Startknoten**	*parent*	*queue*
0	s	0	-	
1	A	3	s	[B(4), C(2), D(4)]
1	B	4	s	[C(2), D(2), A(2)]
1	C	2	s	[D(2), A(2), F(4)]
2	D	4	B	[A(2), F(1)]
2	A	2	B	[F(1)]
2	F	1	D	[E(3)]
3	E	3	F	[]

Lösung 2 – Nicht streng nach Aufgabenstellung, aber auch akzeptiert, da diese Art der Lösung in dem Screencast Video präsentiert wird:

Iteration	**Knoten**	**Distanz zum Startknoten**	*parent*	Notizen
0	s	0	-	
1	A	3	s	später verbessert von B
1	B	4	s	
1	C	2	s	
2	D	4	A	später verbessert von B
2	A	2	B	
2	D	2	B	
2	F	4	C	später verbessert von D
3	F	1	D	
4	E	3	F	

(b) Nehmen Sie an, das Gewicht der Kante F-E wäre -2, statt 2. Können Sie mit dem Bellmann-Ford-Algorithmus weiterhin den kürzesten Weg von s zu D bestimmen? Begründen Sie.

Nein. Negativer Zyklus. Es gibt keinen kürzesten Weg.

Aufgabe 7: Hashing ($8 + 3 = 11$ Punkte)

Es wird eine Hashtabelle der Größe 10 zur Speicherung von Integer-Werten betrachtet, d.h., die Schlüssel sind vom Typ Integer. Die Hashadresse eines Integer k wird durch

$$hash(k) = k \mod 10$$

berechnet. In dieser Aufgabe geht es um die Anzahl von Schlüsselvergleichen bei Kollisionsauflösung durch Lineares Sondieren (Linear Probing) mit Inkrement 1. In der Hashtabelle unter (a) benötigt eine Suche nach dem Schlüssel **17** *einen* Schlüsselvergleich (mit 17) und eine Suche nach dem Schlüssel **7** *zwei* Schlüsselvergleiche (mit 17 und mit 7).

(a) Die folgenden vier Schlüssel sollen in die untenstehenden Hashtabellen eingetragen werden:

4, 12, 22, 23

Tragen Sie die Schlüssel bei der ersten Tabelle in einer Reihenfolge ein, die die Anzahl der Schlüssel (aus **4**, **12**, **22** und **23**) maximiert, bei denen die Suche jeweils mit nur einem Schlüsselvergleich auskommt.

Index	0	1	2	3	4	5	6	7	8	9
Key								17	7	

Maximiere Anzahl der Schlüssel **4**, **12**, **22**, **23**, bei denen die Suche mit *einem* Schlüsselvergleich auskommt.

Bei der zweiten Tabelle soll eine Reihenfolge gewählt werden, die die *maximale* Anzahl der Schlüsselvergleiche bei der Suche nach den Schlüsseln **4**, **12**, **22**, **23** *minimiert*.

Index	0	1	2	3	4	5	6	7	8	9
Key								17	7	

Minimiere *Worst Case* Anzahl von Schlüsselvergleichen bei der Suche nach **4**, **12**, **22**, **23**.

Index	0	1	2	3	4	5	6	7	8	9
Key			12	23	4	22		17	7	

Die Reihenfolge **22**, **23**, **4**, **12** ist auch korrekt.

Index	0	1	2	3	4	5	6	7	8	9
Key			12	22	23	4		17	7	

Die Reihenfolge **22**, **12**, **23**, **4** ist auch korrekt.

(b) Für diese Teilaufgabe ist die Hashtabelle mit anderen Schlüsseln gefüllt worden. Es soll zunächst der Schlüssel **13** und danach der Schlüssel **8** gesucht werden. Schreiben Sie die Schlüssel aus der Hashtabelle auf, mit denen der gegeben Schlüssel bei der Suche verglichen wird.

Hinweis: Eine der beiden Suchen ist erfolglos, da der Schlüssel nicht in der Hashtabelle enthalten ist.

Index	0	1	2	3	4	5	6	7	8	9
Key	8	19		23	3	14		27	7	18

Schlüsselvergleich bei Suche nach 13: 13 == 23, 13 == 3, 13 == 14, dann erfolgloser Abbruch, da $HT[6]$ leer ist

Schlüsselvergleich bei Suche nach 8: 8 == 7, 8 == 18, 8 == 8, erfolgreicher Abbruch, da Schlüssel in $HT[0]$ gefunden

Aufgabe 8: Minimaler Spannbaum ($6 + 6 = 12$ Punkte)

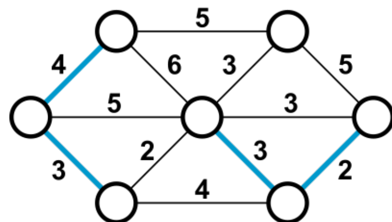
In der folgenden Abbildung sehen Sie sechs mal den gleichen Graphen, wobei jedes mal unterschiedliche Kanten (**blau**) markiert sind. *Hinweis:* Lesen Sie zuerst die ganze Aufgabe.

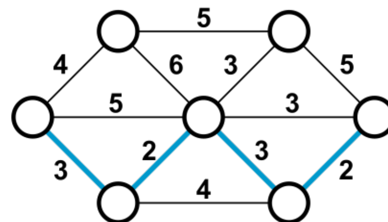
(a) Kreuzen Sie für jeden der sechs Graphen an, ob die markierten Kanten nur durch den **Prim** Algorithmus, nur durch den **Kruskal** Algorithmus, durch **beide** oder durch **keinen** von beiden ausgewählt worden sein können. Dabei muss der jeweilige Algorithmus nicht bis zum Ende durchgelaufen sein. Es können also auch die Markierungen **während** der Berechnung eines MST abgebildet sein.

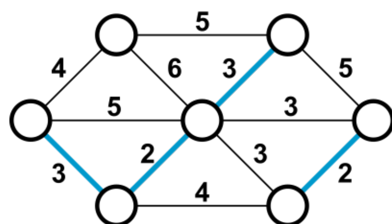
(b) Markieren Sie außerdem in jedem Graphen folgendes:

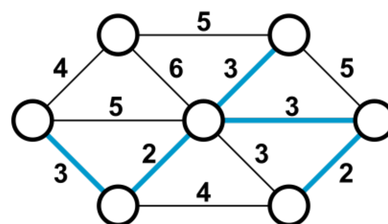
gewählte Algorithmen	Markierung
Prim	einen Startknoten für den Prim Algorithmus, der zu der dargestellten Kantenauswahl führt
Kruskal	eine Kante, die Kruskal als nächstes auswählen könnte
beide	die Markierungen für *Prim* **und** *Kruskal*, siehe oben
keiner	eine der hervorgehobenen Kanten, die in einer partiellen Lösung von Kruskal nicht ausgewählt worden wäre

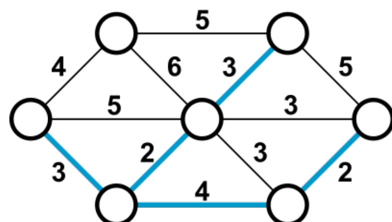
Markieren Sie Kanten, indem Sie das Gewicht umkreisen, und Knoten, indem Sie sie ausmalen. Wenn Sie eine andere Markierung wählen, z. B. weil Sie etwas verändern wollen, schreiben Sie eine lesbare und eindeutige Legende daneben.

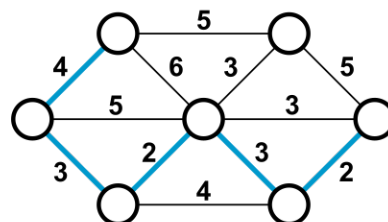












OL: keiner, 4er-Kante oben links oder eine der anderen nicht 2-er Kanten

OR: beide: 3er-Kante oben rechts, beliebiger Knoten an markierter Kante

ML: Kruskal: eine der 3er-Kanten

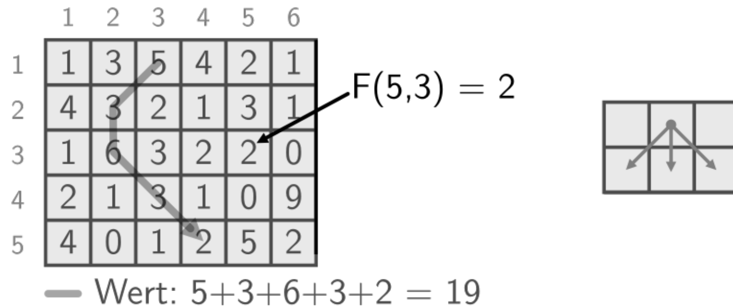
MR: beide: 4er-Kante oben links, Knoten mit markierter Kante

UL: keiner, 4er-Kante unten mitte

UR: Prim: Knoten oben links

Aufgabe 9: Dynamische Programmierung ($4 + 2 = 6$ Punkte)

Es ist eine Matrix der Größe $M \times N$ mit nicht-negativen ganzen Zahlen gegeben. Mit Hilfe dynamischer Programmierung soll ein Weg von der obersten zu der untersten Zeile gefunden werden, der die maximale Summe 'einsammelt', siehe Abbildung. Der Weg kann dabei in einer beliebigen Zelle der ersten Zeile beginnen und in einer beliebigen Zelle der letzten Zeile enden.
Bei jedem Schritt verläuft der Weg von einer Zelle in eine der gerade oder diagonal darunter liegenden Zellen, also von einem Feld (m, n) zu einem der Felder $(m-1, n+1)$, $(m, n+1)$ oder $(m+1, n+1)$, sofern diese Felder innerhalb der Matrix liegen. Es gilt $M \geq 3$ und $N \geq 3$. Der Wert eines Feldes (m, n) wird als $F(m, n)$ bezeichnet.



Die Abbildung links zeigt ein Raster der Größe 6×5 in dem ein Weg von der obersten zur untersten Zeile eingezeichnet ist. Der Wert des Weges ist die Summe der durchlaufenen Felder, also $5 + 3 + 6 + 3 + 2 = 19$. Die Darstellung rechts zeigt, auf welche Art der Weg von einer Zeile in die nächste verlaufen kann.

(a) Definieren Sie eine rekursive Funktion $\text{OPT}(m, n)$, die den größten Wert eines Weges von der obersten Zeile zu dem Feld (m, n) erzielt ($1 \leq m \leq M, 1 \leq n \leq N$). Verwenden Sie eine mathematische Schreibeweise und keinen Java- oder Pseudo-Code.

Hier werden exemplarisch zwei Lösungsmöglichkeiten gezeigt.

Erweitere den Definitionsbereich von OPT auf $0 \leq m \leq M + 1$ und $0 \leq n \leq N$; Anfangsfälle für $n = 0$:

$$\text{OPT}(m, n) = \begin{cases} 0 & \text{falls } m = 0 \text{ oder } m = M + 1 \text{ oder } n = 0 \text{, sonst:} \\ F(m, n) + \max\{\text{OPT}(m-1, n-1),\ \text{OPT}(m, n-1),\ \text{OPT}(m+1, n-1)\} \end{cases}$$

Explizite Unterscheidung der Randfälle; Anfangsfälle für $n = 1$:

$$\text{OPT}(m, n) = \begin{cases} F(m, 1) & \text{falls } n = 1 \\ F(m, n) + \max\{\text{OPT}(m-1, n-1),\ \text{OPT}(m, n-1), \\ \qquad \text{OPT}(m+1, n-1)\} & \text{falls } 1 < m < M \text{ und } 1 < n \leq N \\ F(m, n) + \max\{\text{OPT}(m, n-1),\ \text{OPT}(m+1, n-1)\} & \text{falls } m = 1 \text{ und } 1 < n \leq N \\ F(m, n) + \max\{\text{OPT}(m-1, n-1),\ \text{OPT}(m, n-1)\} & \text{falls } m = M \text{ und } 1 < n \leq N \end{cases}$$

Die Werte können auch *von unten nach oben* bestimmt werden. Dann sind die Anfangsfälle bei $n = N$ (bzw. $n = N + 1$) und das zweite Argument beim rekursiven Aufruf der OPT-Funktion lautet überall $n + 1$ an Stelle von $n - 1$.

(b) Die Werte der OPT-Funktion werden in einer Matrix $D[m][n]$ für alle $1 \leq m \leq M$ und $1 \leq n \leq N$ gespeichert. Geben Sie ein Verfahren an, mit dem die Lösung (maximale Summe) aus dieser Matrix D bestimmt werden kann.

Hinweis: Sie brauchen zur Lösung dieser Teilaufgabe die Lösung von (a) nicht zu kennen. Die Beschreibung der OPT-Funktion aus der Aufgabenstellung reicht aus.

Es wird der maximale Wert in der letzten Zeile des Arrays D gesucht, also $\max\{D[m][N] \mid 1 \leq m \leq M\}$ bestimmt.

Bemerkung:
Falls die Werte von unten nach oben bestimmt werden (Anfangsfälle bei $n = N$), wird die Lösung als maximaler Wert der *ersten* Zeile des Arrays bestimmt.
Der maximale Wert des gesamten Arrays D ist auch richtig, allerdings die Bestimmung weniger effizient.