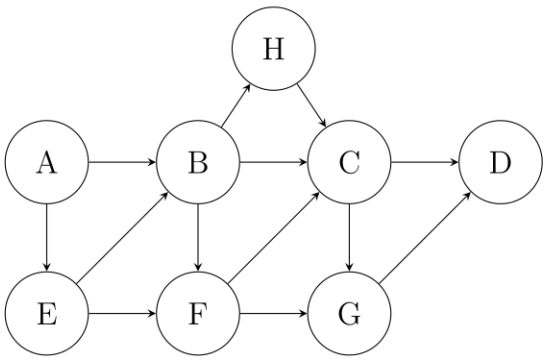
Aufgabe 1: Breitensuche und Tiefensuche (11 Punkte)

(a) (3 Punkte) Der folgende gerichtete Graph ist gegeben. Führen Sie eine Handsimulation der Breitensuche (BFS) und Tiefensuche (DFS) durch und geben Sie die Reihenfolge an, in der die Knoten abgearbeitet werden (**Postorder**-Traversierung für DFS).

Hinweise: Beginnen Sie beide Algorithmen im Knoten A. Gehen Sie in alphabetischer Reihenfolge vor, falls es Wahlmöglichkeiten für benachbarte Knoten gibt.



Breitensuche (BFS): A B E C F H D G

Tiefensuche (DFS): D G C F H B E A

Punkte:

1.5 P für die richtige BFS-Reihenfolge (0.5 P Abzug pro Abweichung. Also ab 3 Abweichungen 0 P.)
1.5 P für die richtige DFS-Reihenfolge (0.5 P Abzug pro Abweichung. Also ab 3 Abweichungen 0 P.)

(b) (2 Punkte) Handelt es sich bei der folgenden Implementierung um BFS oder DFS? Begründen Sie.

```
1  public void traverse(Graph G, int start) {
2      boolean[] visited = new boolean[G.V()];
3      Queue<Integer> dataStructure = new LinkedList<>();
4
5      dataStructure.add(start);
6      visited[start] = true;
7
8      while (!dataStructure.isEmpty()) {
9          int v = dataStructure.poll();
10         System.out.print(v + " ");
11
12         for (int w : G.adj(v)) {
13             if (!visited[w]) {
14                 visited[w] = true;
15                 dataStructure.add(w);
16             }
17         }
18     }
19 }
```

Antwort: Die Implementierung ist **BFS (Breitensuche)**.

Begründung: Die Implementierung verwendet eine Queue (FIFO - First In, First Out), die charakteristisch für BFS ist. Knoten werden in der Reihenfolge ihrer Entdeckung abgearbeitet, was zu einer schichtweisen Traversierung führt.

Punkte:

1 P für korrekte Identifikation als BFS
1 P richtige Begründung (Queue ist typisch für BFS)

(c) (1 Punkt) Geben Sie an, welche Zeilen mindestens geändert werden müssten, um den Algorithmus in den jeweils anderen Suchalgorithmus (BFS ↔ DFS) umzuwandeln.

Zu ändernde Zeilen: 3, 5, 9, 15

Konkrete Änderungen:

- **Zeile 3:** `Queue<Integer>` ändern zu `Stack<Integer>`
- **Zeile 5:** `dataStructure.add(start)` ändern zu `dataStructure.push(start)`
- **Zeile 9:** `dataStructure.poll()` ändern zu `dataStructure.pop()`
- **Zeile 15:** `dataStructure.add(w)` ändern zu `dataStructure.push(w)`

Diese Änderungen wandeln die FIFO-Datenstruktur (Queue) in eine LIFO-Datenstruktur (Stack) um, wodurch aus BFS eine DFS wird.

Punkte:

0.5 P für korrekte Zeilennummer (3), Änderung der Datenstruktur (Queue zu Stack) **0.5 P** für korrekte Zeilennummern (5, 9, 15), Änderung der Methodenaufrufe (diese drei Zeilen müssen exakt stimmen)

(d) (2 Punkte) Entscheiden Sie für jedes Szenario, welcher Algorithmus am besten geeignet ist. Kreuzen Sie das passende Feld an. Pro Aussage ist nur eine Antwort richtig.

Szenario	beide	nur BFS	nur DFS	keiner
Garantiertes Finden eines kürzesten Pfads zwischen zwei Knoten in einem ungewichteten Graphen	☐	☒	☐	☐
Finden aller Knoten in einer Zusammenhangskomponente eines ungerichteten Graphen	☒	☐	☐	☐
Generierung einer topologischen Sortierung in einem azyklischen gerichteten Graphen (DAG)	☐	☐	☒	☐
Navigation in einem Labyrinth mit unendlich tiefen Sackgassen, um den Ausgang zu finden	☐	☒	☐	☐

Punkte:

0.5 P für jedes richtige Kreuz

Erklärungen:

- **Garantiertes Finden kürzesten Pfads:** Nur BFS garantiert den kürzesten Pfad in ungewichteten Graphen, da es Knoten schichtweise abarbeitet. DFS könnte zufällig einen kürzesten Pfad finden, aber es gibt keine Garantie.
- **Finden aller Knoten in einer Zusammenhangskomponente:** Beide Algorithmen können alle Knoten in einer Zusammenhangskomponente finden, indem sie von einem Startknoten aus alle erreichbaren Knoten besuchen.
- **Topologische Sortierung:** Nur DFS eignet sich natürlich für topologische Sortierung durch die Postorder-Reihenfolge.
- **Labyrinth mit unendlichen Sackgassen:** Nur BFS terminiert garantiert, da es nicht in unendlich tiefe Pfade gerät wie DFS.

(e) (3 Punkte) Sie erhalten ein 2D-Grid, das eine Karte darstellt. `1` steht für Land und `0` für Wasser. Eine Insel ist eine Gruppe von horizontal oder vertikal benachbarten Landfeldern, umgeben von Wasser. Ihre Aufgabe ist es, eine Funktion zu implementieren, die die Anzahl der Inseln auf der Karte zählt.

Beispiel mit drei Inseln:

1	1	1	0	0
1	1	0	0	0
0	0	1	0	0
0	0	0	1	1

Implementieren Sie die rekursive Helfermethode `dfs`. Die Methode soll eine gefundene Insel als besucht markieren (indem sie die `1`-Felder auf `2` setzt), um sie nicht doppelt zu zählen. Die Methode `isInBounds` prüft, ob die Koordinaten innerhalb der Grid-Grenzen liegen.

```java
public class Solution {
    public int countIslands(int[][] grid) {
        int numIslands = 0;
        for (int r = 0; r < grid.length; r++) {
            for (int c = 0; c < grid[0].length; c++) {
                if (grid[r][c] == 1) {
                    numIslands++;
                    dfs(grid, r, c);
                }
            }
        }
        return numIslands;
    }

    private boolean isInBounds(int[][] grid, int r, int c) {
        return r >= 0 && r < grid.length && c >= 0 && c < grid[0].length;
    }

    private void dfs(int[][] grid, int r, int c) {









    }
}
```


Lösung:

```
public class Solution {
    public int countIslands(int[][] grid) {
        int numIslands = 0;
        for (int r = 0; r < grid.length; r++) {
            for (int c = 0; c < grid[0].length; c++) {
                if (grid[r][c] == 1) {
                    numIslands++;
                    dfs(grid, r, c);
                }
            }
        }
        return numIslands;
    }

    private boolean isInBounds(int[][] grid, int r, int c) {
        return r >= 0 && r < grid.length && c >= 0 && c < grid[0].length;
    }

    private void dfs(int[][] grid, int r, int c) {
        // 1. Abbruchbedingungen pruefen
        if (!isInBounds(grid, r, c) || grid[r][c] != 1) {
            return;
        }

        // 2. Aktuelles Feld als besucht markieren
        grid[r][c] = 2;

        // 3. Rekursiv fuer alle 4 Nachbarn aufrufen
        dfs(grid, r + 1, c); // unten
        dfs(grid, r - 1, c); // oben
        dfs(grid, r, c + 1); // rechts
        dfs(grid, r, c - 1); // links
    }
}
```

Punkte:

1 P für korrekte Abbruchbedingungen: `!isInBounds(grid, r, c) || grid[r][c] != 1` (0.5 P Abzug wenn isInBounds vergessen wurde)
1 P für korrekte Markierung des aktuellen Feldes: `grid[r][c] = 2`
1 P für korrekte rekursive Aufrufe für alle 4 Nachbarn (oben, unten, links, rechts) (0.5 P wenn nur 1 bis 3 Nachbarn aufgerufen werden)

Aufgabe 2: Greedy-Algorithmen (10 Punkte)

(a) (4 Punkte) Beschreiben Sie für jeden der folgenden Algorithmen kurz und präzise die zugrundeliegende **Greedy-Strategie** (**nicht** den gesamten Algorithmus). Erklären Sie, was die **lokal optimale Entscheidung** ist, die der Algorithmus in jedem Schritt trifft, um eine **globale optimale Lösung** zu finden.

1.) **Algorithmus von Prim**
Die Greedy-Strategie von Prim besteht darin, in jedem Schritt die **Kante mit dem minimalen Gewicht** auszuwählen, die einen bereits im entstehenden minimalen Spannbaum (MST) befindlichen Knoten mit einem Knoten **außerhalb** des MST verbindet. Der Baum wird also von einem Startknoten aus schrittweise um die jeweils günstigste erreichbare Kante erweitert.

2.) **Algorithmus von Kruskal**
Die Greedy-Strategie von Kruskal ist es, in jedem Schritt die **global leichteste Kante** aus der Menge aller Kanten im Graphen auszuwählen, die **keinen Zyklus** mit den bereits zum MST hinzugefügten Kanten bildet.

3.) **Algorithmus von Dijkstra**
Die Greedy-Strategie von Dijkstra besteht darin, in jedem Schritt den noch **unbesuchten Knoten** auszuwählen, der die **geringste bisher bekannte Distanz** zum Startknoten hat. Die Distanz zu diesem Knoten wird dann als final deklariert.

4.) **A*-Algorithmus**
Die Greedy-Strategie von A* ist die Auswahl des Knotens, der den **minimalen Wert der Kostenfunktion** $f(n) = g(n) + h(n)$ aufweist. Dabei ist $g(n)$ die bereits bekannte, exakte Distanz vom Startknoten zum Knoten n, und $h(n)$ ist eine **heuristische Schätzung** der Distanz von n zum Zielknoten. Der "greedy" Anteil ist die Heuristik $h(n)$, die den Algorithmus in die vielversprechendste Richtung lenkt.

Punkte:

Teilaufgabe	**Notwendige Keywords/Konzepte für 1 Punkt**	**Kriterien für 0.5 Punkte**
1.) Prim	Auswahl der **Kante mit minimalem Gewicht**, die einen Knoten **im MST mit einem Knoten außerhalb** verbindet.	Nennt nur die Auswahl der leichtesten Kante, ohne den Kontext (Verbindung zum MST) zu erwähnen.
2.) Kruskal	Auswahl der **global leichtesten Kante**, die beim Hinzufügen **keinen Zykel** erzeugt.	Nennt nur die Auswahl der global leichtesten Kante, aber vergisst die Zykel-Prüfung.
3.) Dijkstra	Auswahl des Knotens mit der **minimalen bekannten Distanz** vom Startknoten. Es ist auch ok, wenn hier die PQ erwähnt wird (als Ersatz für minimale bekannte Distanz).	- Wenn nur Auswahl nach minimaler Distanz genannt wird.
4.) A*	Auswahl des Knotens, der $f(n) = g(n) + h(n)$ minimiert, wobei $g(n)$ **die Kosten vom Start** und $h(n)$ **eine Heuristik** zum Ziel ist.	Erwähnt, dass eine Heuristik verwendet wird, aber beschreibt die Auswahlregel mit $f(n)$ nicht präzise.

(b) (6 Punkte) Sie sind für die Programmplanung eines Fernsehsenders zuständig. Für einen Werbeblock mit einer Gesamtlänge **90 Sekunden** sollen Sie möglichst **viele verschiedene Werbespots** unterbringen. Ihnen liegt eine Liste verfügbarer Spots mit **unterschiedlichen Längen** vor, die alle das **gleiche Gewicht** haben.

Ihre Teamleiterin schlägt folgende Greedy-Strategie vor: *„Um sicherzustellen, dass wir unser Zeitfenster von 90 Sekunden auch wirklich **voll ausnutzen**, sollten wir zuerst die Spots wählen, die den **meisten Platz wegnehmen**. Wähle also immer den **längsten** Werbespot aus, der noch in die verbleibende Zeit des Werbeblocks passt"*

Sie sind skeptisch, ob diese Strategie wirklich die Anzahl der Spots maximiert.

1. (3 Punkte) Widerlegen Sie die Strategie Ihrer Teamleiterin durch ein **Gegenbeispiel**. Geben Sie dazu eine Liste von Werbespots (mit Längen in Sekunden) an, für welche die Strategie **„längster Spot zuerst“** versagt.
2. (3 Punkte) Nennen Sie die **optimale Greedy-Strategie** und demonstrieren Sie an Ihrem Gegenbeispiel aus 1., dass diese zu einer besseren Lösung führt. Geben Sie für beide Strategien die **finale Auswahl an Werbespots** und die **Gesamtzahl der gesendeten Spots** an.

1. Gegenbeispiel *Hinweis: Jedes korrekte Gegenbeispiel ist zulässig.*

Beispiel Gegenbeispiel: Gesamtdauer des Werbeblocks: 90 Sekunden. Folgende Werbespots stehen zur Verfügung:

- Spot A: 80 Sekunden
- Spot B: 45 Sekunden
- Spot C: 45 Sekunden

2. Optimale Strategie und Demonstration

Die optimale Greedy-Strategie zur Maximierung der Anzahl lautet: *„Wähle immer den Werbespot mit der **kürzesten Dauer**, der noch in die verbleibende Zeit passt.“*

Vergleich der Strategien am Gegenbeispiel:

- **Strategie der Teamleiterin („Längster zuerst”):**
  1. Sortierung nach Länge (absteigend): A (80s), B (45s), C (45s).
  2. Wähle den längsten passenden Spot: **Spot A** (80s).
  3. Verbleibende Zeit: 90 - 80 = 10 Sekunden.
  4. Weder Spot B noch C (beide 45s) passen in die verbleibende Zeit.

  Ergebnis: {Spot A}, Anzahl: 1
- **Optimale Strategie („Kürzester zuerst”):**
  1. Sortierung nach Länge (aufsteigend): B (45s), C (45s), A (80s).
  2. Wähle den kürzesten passenden Spot: **Spot B** (45s).
  3. Verbleibende Zeit: 90 - 45 = 45 Sekunden.
  4. Wähle den nächsten kürzesten passenden Spot: **Spot C** (45s).
  5. Verbleibende Zeit: 45 - 45 = 0 Sekunden.

  Ergebnis: {Spot B, Spot C}, Anzahl: 2

Der Vergleich belegt, dass die „Kürzester zuerst“-Strategie (2 Spots) zu einem besseren Ergebnis führt als die „Längster zuerst“-Strategie (1 Spot) und somit in diesem Fall überlegen ist.

Punkte:

Teil 1:
3 P für die Angabe eines validen Gegenbeispiels. Das Beispiel muss mindestens einen “langen” Spot enthalten, dessen Wahl die Auswahl von mindestens zwei “kürzeren” Spots verhindert. Die Zahlen müssen zum Gesamtlimit (90s) passen.

Teil 2:
1 P für die korrekte Benennung der optimalen Strategie („kürzesten Spot zuerst“).
1 P für die korrekte Anwendung der „Längster zuerst“-Strategie auf das eigene Beispiel.
1 P für die korrekte Anwendung der optimalen „Kürzester zuerst“-Strategie und den Nachweis, dass sie zu einem besseren Ergebnis führt.

Aufgabe 3: Branch and Bound (11 Punkte)

Sie haben heute noch **12 Minuten verbleibende Arbeitszeit**. Sie möchten verschiedene kleine Programmieraufgaben bearbeiten, um Ihren **Verdienst zu maximieren**. Für jede Aufgabe schätzen Sie den Zeitaufwand und die Bezahlung. Die folgende Tabelle zeigt die Details:

Aufgabe	Bezahlung (€)	Dauer (Min)
A	3	2
B	4	2
C	6	2
D	7	7

Sie verwenden den **Branch and Bound Algorithmus**, um die optimale Aufgabenauswahl zu finden.

(a) (1 Punkt) Auf welches **algorithmische Problem** aus der Vorlesung lässt sich dieses Problem abbilden?

"Knapsack-Problem" oder "Knapsack" oder "0-1-Knapsack-Problem" oder "Rucksackproblem"

Punkte:

1P Korrekte Nennung

(b) (1 Punkt) Welche Greedy Strategie würden Sie verwenden, um eine möglichst gute **Initiallösung** für dieses Problem zu finden?

Strategie: Der Greedy-Algorithmus sortiert die Aufgaben nach dem Verhältnis von Nutzen zu Aufwand (hier: Minutenlohn) und wählt gierig die besten Aufgaben aus, bis die Kapazität erschöpft ist. Dies liefert schnell eine gute Ausgangslösung für Branch and Bound.

Punkte:

0.5P Sortierstrategie nach Minutenlohn
0.5P Auswahlstrategie der besten Aufgaben bis Kapazität erschöpft ist

(c) (2 Punkte) Berechnen Sie nun die Initiallösung mit Ihrer vorgeschlagenen Strategie und geben sie die **untere Schranke** bei Beginn des Branch and Bound Algorithmus an. Geben sie auch ihren Rechenweg an.

Schritt 1: Berechnung des Minutenlohns

- A: 3€ ÷ 2 Min = 1.5€/Min
- B: 4€ ÷ 2 Min = 2€/Min
- C: 6€ ÷ 2 Min = 3€/Min
- D: 7€ ÷ 7 Min = 1€/Min

Schritt 2: Sortierung nach Minutenlohn (absteigend) C → B → A → D

Schritt 3: Greedy-Auswahl der sortierten Aufgaben C + B + A, da 2 + 2 + 2 = 6 Min ≤ 12 Min (+ D würde 13 Min ergeben und ist nicht erlaubt)

Aufgaben der Initiallösung: C, B, A

Untere Schranke bei Beginn: 13€ (Gesamtverdienst der Greedy-Lösung)

Punkte:

1P Korrekte Sortierung (halber Punkt, wenn alle Minutenlöhne korrekt sind, aber Sortierung falsch). Wenn die Minutenlöhne da sind und die Lösung unten korrekt, aber nicht explizit sortiert wurde, dann kann dieser Punkt auch gegeben werden, da die Sortierung aus dem Kontext implizit ist.
0.5P Korrekte Initiallösung
0.5P Korrekte untere Schranke (13€)
Falls in vorheriger Teilaufgabe nahc Bezahlung und nicht nach Minutenlohn sortiert wurde, dann kann hier immer noch volle Punktzahl erreicht werden.

(d) (2 Punkte) Berechnen Sie die **obere Schranke** bei Beginn des Branch and Bound Algorithmus mittels der Lösung des teilbaren Rucksackproblems. Zeigen Sie ihren Rechenweg.

Wir haben bereits die Initiallösung C+B+A, welche 6 Minuten dauert. Insgesamt sind 12 Minuten erlaubt, also bleiben 6 Minuten übrig.

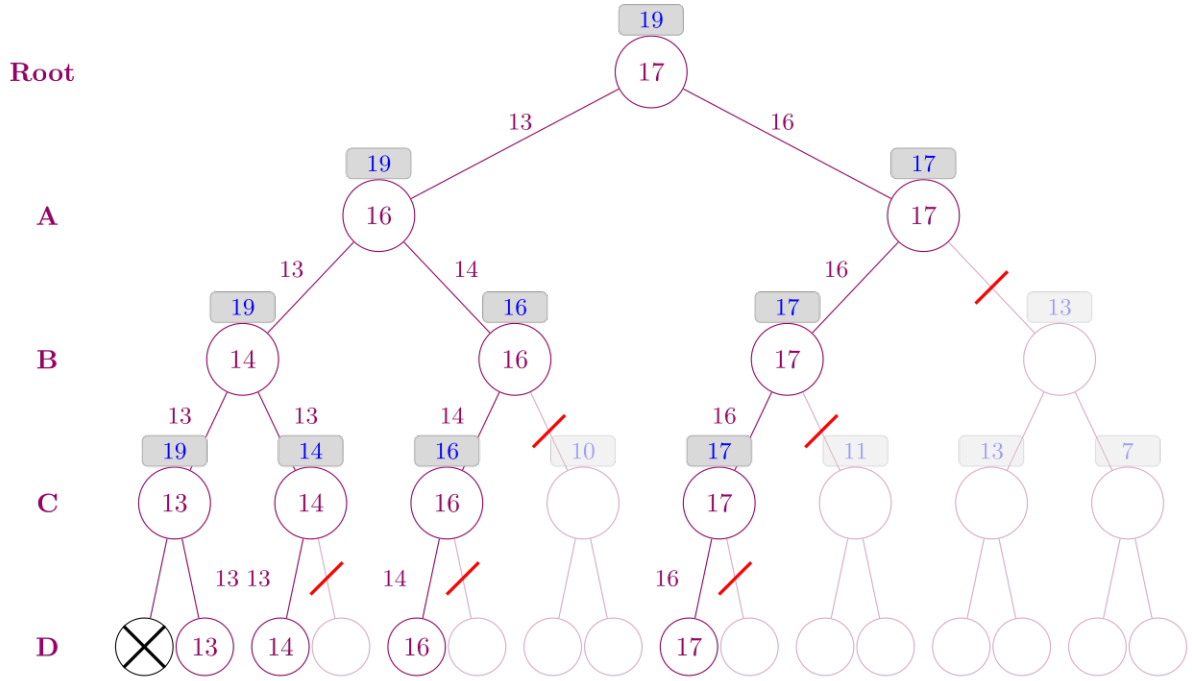
D dauert 7 Minuten, also können wir 6/7 von D hinzufügen. Die obere Schranke ist also 13€ (die untere Schranke) + 6/7 * 7€ = 13€ + 6€ = 19€.

Obere Schranke 19€

Punkte:

1P Korrekter Rechenweg mittels Fractional Scaling
1P Korrekte obere Schranke (19€)

(e) (5 Punkte) Führen Sie den **Branch and Bound Algorithmus** auf dem gegebenen Baum aus.



Optimale Lösung: B, C, D

Erreichter Wert: 17€

Punkte:

1P Korrekte Berechnung der relevanten Blattknoten (0.5 Punkte Abzug für jeden falschen Wert)
1P Korrekte untere Schranken in allen Knoten (0.5 Punkte Abzug für jeden falschen Wert)
1P Korrekte Verdienste (€) in den Elternknoten (0.5 Punkte Abzug für jeden falschen Wert)
1P Korrekte Identifikation aller Pruning-Stellen (0.5 Punkte Abzug für jeden falsche Stelle)
0.5P Korrekte optimale Lösung
0.5P Korrekter Wert

Aufgabe 4: Java Programmierung (11 Punkte)

(a) (6 Punkte) In diesem Code-Snippet soll eine Einkaufswagen-Verwaltung implementiert werden. Der Code enthält jedoch **6 verschiedene Fehler**, die verhindern, dass das Programm korrekt kompiliert oder ausgeführt wird. Geben Sie die Fehler und den korrigierten Java-Code in der Tabelle an.

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  class ShoppingCart {
5      private String customerName;
6      private List<String> items;
7
8      public ShoppingCart(String customerName) {
9          customerName = customerName;
10     }
11
12     public void addItem(String item) {
13         if (item != null & item.length() > 0) {
14             items.add(item);
15         }
16     }
17
18     public String getCartSummary() {
19         String result = "Kunde: " + customerName;
20         for (int i = 0; i <= items.size(); i++) {
21             result += ", Artikel: " + items.get(i);
22         }
23         return result;
24     }
25
26     public boolean containsItem(String item) {
27         for (String cartItem : items) {
28             if (cartItem = item) return true;
29         }
30         return false;
31     }
32 }
33
34 public class ShopSystem {
35     public static void main(String[] args) {
36         ShoppingCart cart = new ShoppingCart("Mueller");
37         cart.addItem("Apfel");
38
39         System.out.println(cart.getCartSummary());
40         System.out.println("Enthaelt Brot: " + cart.containsItem("Brot"));
41         System.out.println("Kunde: " + cart.customerName);
42     }
43 }
```


Zeile	Korrektur
9	Fehler: Fehlendes `this.` im Konstruktor. Korrektur: `this.customerName = customerName;`
9	Fehler: Liste nicht initialisiert. Korrektur: `this.items = new ArrayList<>();`, Es ist auch ok, wenn die liste als `LinkedList` initialisiert wird und der entsprechende Import eingefügt wird.
13	Fehler: Bitweises AND statt logisches AND. Korrektur: `if (item != null && item.length() > 0)`. Bitwise AND (`&`) ist explizit nicht korrekt, da es eine `NullPointerException` wirft, wenn `item` `null` ist. Das logische AND (`&&`) stellt sicher, dass `item.length() > 0` nicht ausgeführt wird, wenn `item` `null` ist (sog. short-circuit).
20	Fehler: IndexOutOfBounds wg. `<=`. Korrektur: `for (int i=0; i < items.size(); i++)`
28	Fehler: Zuweisung statt Vergleich. Korrektur: `if (cartItem.equals(item))` (nicht `cartItem == item`, da dies Objektreferenzen vergleicht)
41	Fehler: Privater Feldzugriff auf `customerName`. Korrektur: Getter-Methode hinzufügen: `public String getCustomerName() { return customerName; }` und `cart.getCustomerName()` verwenden

Punkte:

0.5 P für jedes korrekt markierte Fehler
0.5 P für die korrekte Korrektur

(b) (5 Punkte) Implementieren Sie einen "Shortest Job First"-Scheduler. Verschiedene Prozesse (`Task`) sollen basierend auf ihrer Ausführungszeit (`executionTime`) abgearbeitet werden, wobei Tasks mit der **kürzesten** Zeit zuerst ausgeführt werden sollen.

Hinweis: Nutzen Sie dafür die `java.util.PriorityQueue`. Diese Klasse ist standardmäßig eine **Min-Priority-Queue** und eignet sich daher perfekt für dieses Problem.

Ergänzen Sie den Lückentext auf der nächsten Seite, sodass

- die Klasse `Task` die nötige Schnittstelle implementiert, um von einer `PriorityQueue` anhand der `executionTime` geordnet zu werden.
- eine `PriorityQueue` mit `Task`-Objekten erstellt wird.
- alle Tasks in der korrekten Reihenfolge (kürzeste Ausführungszeit zuerst) aus der Queue entfernt und ihre Namen ausgegeben werden.

```java
import java.util.PriorityQueue;
import java.util.Queue;

// Die Task-Klasse vergleichbar machen
class Task ____________________ ____________________<Task> {
    public String name;
    public int executionTime;

    public Task(String name, int executionTime) {
        this.name = name;
        this.executionTime = executionTime;
    }

    @Override
    public int compareTo(Task other) {
        // Vergleich der Ausfuehrungszeiten
        return ____________________________________________________________;
    }
}

public class Scheduler {
    public static void main(String[] args) {
        // PriorityQueue instanziieren und Tasks hinzufuegen
        Queue<Task> taskQueue = __________________ ____________________;

        taskQueue._______(new Task("Full system scan", 120));
        taskQueue._______(new Task("Compile module", 25));
        taskQueue._______(new Task("Quick backup", 5));

        // Tasks in Prioritaets-Reihenfolge entfernen
        while (_____________________________________) {
            _____________________________________;
            // Die naechste Zeile simuliert die Ausfuehrung des Tasks
            System.out.println("Bearbeite: " + current.name);
        }
    }
}
```

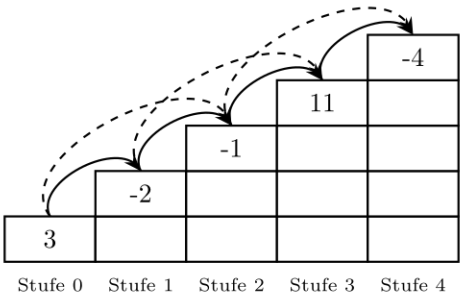

```java
import java.util.PriorityQueue;
import java.util.Queue;

// Die Task-Klasse vergleichbar machen
class Task implements Comparable<Task> {
    public String name;
    public int executionTime; // in Minuten

    public Task(String name, int executionTime) {
        this.name = name;
        this.executionTime = executionTime;
    }

    @Override
    public int compareTo(Task other) {
        // Tasks mit kuerzerer executionTime haben hoehere Prioritaet
        return Integer.compare(this.executionTime, other.executionTime);
        // Alternative: return this.executionTime - other.executionTime;
    }
}

public class Scheduler {
    public static void main(String[] args) {
        // PriorityQueue instanziieren und Tasks hinzufuegen
        Queue<Task> taskQueue = new PriorityQueue<>();

        taskQueue.add(new Task("Full system scan", 120));
        taskQueue.add(new Task("Compile module", 25));
        taskQueue.add(new Task("Quick backup", 5));

        // Tasks in Prioritaets-Reihenfolge entfernen und abarbeiten
        while (!taskQueue.isEmpty()) {
            Task current = taskQueue.poll();
            // Die naechste Zeile simuliert die Ausfuehrung des Tasks
            System.out.println("Bearbeite: " + current.name);
        }
    }
}
```

Punkte:

1 P für `implements Comparable<Task>`. (0.5 Punkte für `extends Comparable<Task>`).
1 P für die korrekte Logik in `compareTo` für "Shortest Job First"
(z.B. `this.executionTime - other.executionTime`
oder `Integer.compare(this.executionTime, other.executionTime)`).
-0.5 wenn Reihenfolge falsch (z.B. `other.executionTime - this.executionTime`
oder `Integer.compare(other.executionTime, this.executionTime)`).
Nach Korrektur: für `this.executionTime.compareTo(other.executionTime)` gibt es 0.5 Punkte.
1 P für `new PriorityQueue<>();`. Nach Korrektur: Auch ok, wenn `new PriorityQueue<Task>()` verwendet wird.
0.5 P für die korrekte Verwendung von `add`. -0.5 Punkte für jedes falsch verwendete `add`. Es ist auch ok, wenn `offer` statt `add` verwendet wird.
0.5 P für `!taskQueue.isEmpty()`.
1 P für `Task current = taskQueue.poll();`.

Aufgabe 5: Dynamische Programmierung (11 Punkte)

Ein Roboter steigt eine Treppe mit n Stufen. Er betritt zuerst Stufe 0 und kann dann **pro Zug eine oder zwei Stufen** nach oben steigen. Jede Stufe i hat einen Punktwert $P[i]$ (dargestellt durch ein Java-Array P). Der Roboter erhält die Punkte einer Stufe beim Betreten der Stufe. Ziel ist es, die oberste Stufe $n-1$ zu erreichen und dabei die **maximale Punktzahl zu sammeln**.



(a) (3 Punkte) Die Abbildung zeigt eine Treppe mit $n = 5$ Stufen und $P = [3, -2, -1, 11, -4]$. Tragen Sie in die Tabelle **alle möglichen Pfade** mit den entsprechenden Punktzahlen ein und bestimmen Sie den optimalen Pfad zur **Maximierung der Punktzahl**.

Pfad (Stufenfolge)	**Punktzahl**
$0 \to 1 \to 3 \to 4$	$3 - 2 + 11 - 4 = 8$
$0 \to 1 \to 2 \to 4$	$3 - 2 - 1 - 4 = -4$
$0 \to 2 \to 4$	$3 - 1 - 4 = -2$
$0 \to 2 \to 3 \to 4$	$3 - 1 + 11 - 4 = 9$
$0 \to 1 \to 2 \to 3 \to 4$	$3 - 2 - 1 + 11 - 4 = 7$

Optimaler Pfad: $0 \to 2 \to 3 \to 4$ mit einer Punktzahl von 9.

Punkte:

0.5 P für jede korrekte Angabe eines Pfades und der Punktzahl (insgesamt 2.5 P). Nach Korrektur: es ist auch ok, wenn die Stufe 0 im Pfad nicht angegeben wird.
0.5 P für korrekte Angabe des optimalen Pfades

(b) (3 Punkte) Geben Sie eine **rekursive Funktion** $\textsc{Opt}(i)$ an, die die **maximale Punktzahl** eines Pfades berechnet, der bei Stufe 0 beginnt und bei Stufe i endet. Verwenden Sie eine mathematische Schreibweise und keinen Java- oder Pseudo-Code.

Hinweis: Geben Sie die allgemeine rekursive Funktion an, unabhängig von den im vorherigen Aufgabenteil gegebenen Werten.

Lösung:

$$\textsc{Opt}(i) = \begin{cases} P[0] & \text{falls } i = 0, \\ P[1] + \textsc{Opt}(0) & \text{falls } i = 1, \\ P[i] + \max\big(\textsc{Opt}(i-1), \textsc{Opt}(i-2)\big) & \text{falls } i \geq 2 \end{cases}$$

Punkte:

1 P für korrekten Basisfall: $\textsc{Opt}(0) = P[0]$
1 P für korrekten Fall für $i = 1$: $\textsc{Opt}(1) = P[1] + \textsc{Opt}(0)$
1 P für korrekten Fall für $i \geq 2$: $\textsc{Opt}(i) = P[i] + \max\big(\textsc{Opt}(i-1), \textsc{Opt}(i-2)\big)$

(c) (2 Punkte) Geben Sie die Laufzeitkomplexität für die Berechnung der **maximal erreichbaren Punktzahl** mit dynamischer Programmierung an. **Vergleichen** Sie diese mit der Laufzeit einer Brute-Force-Lösung und begründen Sie beide Ergebnisse kurz.

Lösung:

Die Größenordnung der Laufzeit für die Berechnung der **maximal erreichbaren Punktzahl** ist $O(n)$, wobei n die Anzahl der Stufen ist. Die Laufzeit ist linear, da jede Stufe nur einmal besucht wird und die Rekursion für jede Stufe konstanten Aufwand hat.

Die Brute-Force Lösung würde alle möglichen Pfade durchlaufen und hätte eine exponentielle Laufzeit von $O(2^n)$, da der Agent für jede Stufe zwei Entscheidungen treffen kann (eine oder zwei Stufen nach oben). Damit reduziert die dynamische Programmierung die Laufzeit erheblich, indem sie bereits berechnete Ergebnisse speichert und wiederverwendet, wodurch die Anzahl der Berechnungen extrem reduziert wird.

Punkte:

0.5 P für korrekte Angabe der Größenordnung der Laufzeit: $O(n)$
0.5 P für korrekte Begründung, dass die Laufzeit linear ist, da jede Stufe nur einmal besucht wird
0.5 P für korrekte Angabe der Brute-Force Laufzeit: $O(2^n)$
0.5 P für korrekte Begründung, dass die dynamische Programmierung die Laufzeit erheblich reduziert, indem sie bereits berechnete Ergebnisse speichert und wiederverwendet

(d) (3 Punkte) Implementieren Sie eine **rekursive** Methode `calculateMaxPoints(int i, int[] P)`, die den Top-Down-Ansatz der dynamischen Programmierung verwendet, um die **maximale Punktzahl** für Stufe i zu berechnen. Die Methode soll bereits berechnete Ergebnisse speichern (Memoization). Sie dürfen die Java-Klasse `Math` verwenden.

```
public class StepClimber {
    private Integer[] memo;

    public StepClimber(int n) {
        // Alle Elemente werden mit `null` initialisiert
        memo = new Integer[n];
    }

    public int calculateMaxPoints(int i, int[] P) {
        // Implementieren Sie hier Ihre Loesung









    }
}
```


Lösung:

```
public class StepClimber {
    private Integer[] memo;

    public StepClimber(int n) {
        // Alle Elemente werden mit 'null' initialisiert
        memo = new Integer[n];
    }

    public int calculateMaxPoints(int i, int[] P) {
        // bereits berechnete Stufe zurueckgeben
        if (memo[i] != null) return memo[i];

        // rekursive Berechnung der maximalen Punktzahl:
        if (i == 0) {
            memo[i] = P[0];
        } else if (i == 1) {
            memo[i] = P[1] + calculateMaxPoints(0, P);
        } else {
            memo[i] = P[i] + Math.max(calculateMaxPoints(i - 1, P),
            calculateMaxPoints(i - 2, P));
        }
        return memo[i];
    }
}
```

Punkte:

1 P für korrekte Überprüfung, ob die Stufe bereits berechnet wurde (`memo[i] != null`)
0.5 P für korrekte Implementierung der Rekursion für
$i = 0$: `memo[i] = P[0]`
0.5 P für korrekte Implementierung der Rekursion für
$i = 1$: `memo[i] = P[1] + calculateMaxPoints(0, P)`
0.5 P für korrekte Implementierung der Rekursion für
$i \geq 2$: `memo[i] =`
`P[i] + Math.max(calculateMaxPoints(i - 1, P), calculateMaxPoints(i - 2, P))`
0.5 P für korrekte Java-Syntax

Aufgabe 6: Minimale Spannbäume (8 Punkte)

(a) (3 Punkte) Lesen Sie die folgenden Aussagen. Entscheiden Sie, ob sie nur für den Prim-Algorithmus, nur für den Kruskal-Algorithmus, für beide oder für keinen der beiden Algorithmen gelten. Kreuzen Sie das passende Feld an. Pro Aussage ist nur eine Antwort richtig.

Aussage	beide	nur Prim	nur Kruskal	keiner
Der Algorithmus kann korrekt auf einem gerichteten Graphen ausgeführt werden, sofern alle Kantengewichte positiv sind.	☐	☐	☐	☒
Der Algorithmus kann so implementiert werden, dass er ohne expliziten Vergleich der Kantengewichte korrekt arbeitet, wenn diese bereits in aufsteigender Reihenfolge gegeben sind.	☐	☐	☒	☐
Der Algorithmus kann bei geschickter Implementierung auch in einem Graphen mit negativen Kantengewichten korrekt einen MST berechnen.	☒	☐	☐	☐
Der Algorithmus kann so implementiert werden, dass sein Speicherbedarf asymptotisch nur von der Knotenzahl V abhängt, nicht von der Kantenzahl E.	☐	☒	☐	☐
Die Reverse-Delete Variante dieses Algorithmus entfernt explizit Kanten aus dem Graphen.	☐	☐	☒	☐
Der Algorithmus eignet sich besser für eine parallele Implementierung.	☐	☐	☒	☐

Punkte:

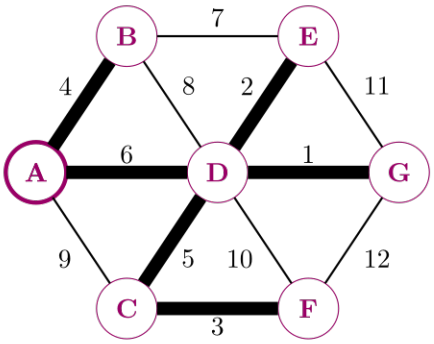
0.5 P für jedes richtige Kreuz

Erklärung der Lösungen:

- **Aussage 1 (keiner):** MST-Algorithmen arbeiten nur auf ungerichteten Graphen. Auf gerichteten Graphen würde man nach einem minimalen Spannwald (Arborescence) suchen, wofür andere Algorithmen nötig sind.
- **Aussage 2 (nur Kruskal):** Kruskal kann ohne explizite Gewichtsvergleiche implementiert werden, wenn die Kanten bereits sortiert vorliegen – man arbeitet sie einfach der Reihe nach ab. Prim muss immer die leichteste ausgehende Kante finden.
- **Aussage 3 (beide):** Beide Algorithmen funktionieren korrekt mit negativen Kantengewichten.
- **Aussage 4 (nur Prim):** Prim kann mit einem einfachen Array für Distanzen implementiert werden (Speicher $O(V)$), während Kruskal alle Kanten speichern muss (Speicher $O(E)$).
- **Aussage 5 (nur Kruskal):** Die Reverse-Delete Variante von Kruskal startet mit allen Kanten und entfernt die schwersten, solange der Graph zusammenhängend bleibt. Diese Variante gibt es nicht für Prim.
- **Aussage 6 (nur Kruskal):** Prim ist inherent sequenziell, da jeder Schritt vom vorherigen abhängt. Kruskal eignet sich besser für Parallelisierung, da die initiale Sortierung der Kanten parallel ausführbar ist (z.B. mithilfe von Merge Sort).

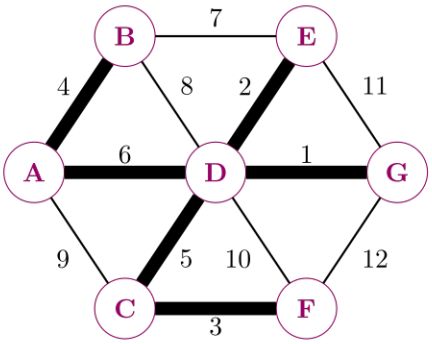
(b) (4 Punkte) Führen Sie den Prim-Algorithmus und den Kruskal-Algorithmus auf den folgenden Graphen aus. Markieren Sie die ausgewählten Kanten direkt in den Graphen und geben Sie in den Tabellen die Reihenfolge an, in der die Kanten zum minimalen Spannbaum hinzugefügt werden.

Prim: Startknoten ist A.



Schritt	Hinzugefügte Kante
1	A-B
2	A-D
3	D-G
4	D-E
5	D-C
6	C-F

Kruskal:



Schritt	Hinzugefügte Kante
1	D-G
2	D-E
3	C-F
4	A-B
5	C-D
6	A-D

Punkte:

1 P für richtig markierten Graph (0.5 P Abzug pro Fehler)
1 P für jede richtig ausgefüllte Tabelle (0.5 P Abzug pro Fehler)

(c) (1 Punkt) Sind die gefundenen minimalen Spannbäume aus Teil (b) identisch oder nicht? Begründen Sie kurz warum sie identisch oder nicht identisch sind.

Da alle Kantengewichte im Graphen eindeutig sind (keine zwei Kanten haben dasselbe Gewicht), gibt es nur einen einzigen minimalen Spannbaum. Beide Algorithmen müssen daher zwangsläufig denselben MST finden, auch wenn sie die Kanten in unterschiedlicher Reihenfolge auswählen.

Punkte:

1 P für korrekte Begründung, dass eindeutige Kantengewichte zu eindeutigem MST führen (es gibt nur einen Punkt für die Begründung, keinen Teilpunkt für die Antwort "gleich", denn das ist ja bereits aus der Lösung von Teil (b) klar)

Aufgabe 7: Dijkstra und Bellman-Ford (8 Punkte)

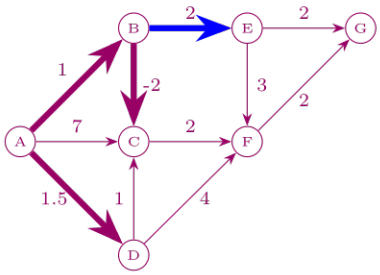
(a) (4 Punkte) Gegeben ist der folgende Graph. Die vier Abbildungen zeigen jeweils einen möglichen Zustand *während der Ausführung* eines Algorithmus zur Bestimmung kürzester Wege mit Startknoten A. Die *dick gezeichneten Kanten* stellen den *bisher konstruierten Kürzeste-Wege-Baum dar.*

Kreuzen Sie an, welcher Algorithmus (nur Dijkstra, nur Bellman-Ford, beide oder keiner) den gezeigten Zustand erzeugt haben könnte.

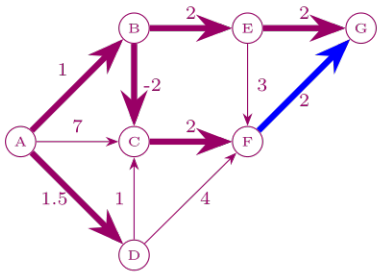
(b) (2 Punkte) Markieren Sie direkt im Graphen durch Umkreisen des Kantengewichts. Die konkrete Markierung hängt von Ihrer Antwort in Teil a) ab:

- Falls Sie **Dijkstra** angekreuzt haben: Umkreisen Sie das Gewicht der Kante, die Dijkstra als **Nächstes** dem Baum hinzufügen würde.
- Falls Sie **Bellman-Ford** angekreuzt haben: Umkreisen Sie das Gewicht der Kante, deren Relaxation als **Nächstes** zu einer Aktualisierung des Kürzeste-Wege-Baums führen würde. (Gehen Sie von einer alphabetischen Abarbeitung der Knoten und Kanten aus).
- Falls Sie **beide** angekreuzt haben: Führen Sie beide oben genannten Markierungen durch. Kennzeichnen Sie eindeutig, welche Markierung zu welchem Algorithmus gehört (z. B. mit „D“ und „BF“).
- Falls Sie **keiner** angekreuzt haben: Umkreisen Sie das Gewicht einer der dick gezeichneten Kanten, die einen Widerspruch zur Funktionsweise beider Algorithmen darstellt und begründen Sie Ihre Wahl stichpunktartig.

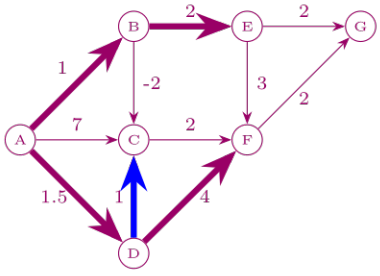
Hinweis: Falls Sie eine Markierung korrigieren, streichen Sie die alte eindeutig durch. Eine lesbare Legende ist bei Abweichungen zulässig.



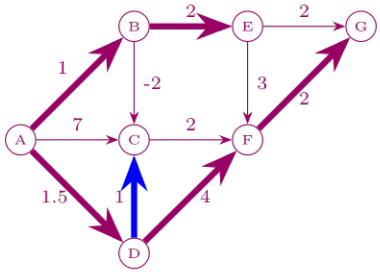
◯ nur Dijkstra ⊗ beide
◯ nur Bellman-Ford ◯ keiner



◯ nur Dijkstra ◯ beide
⊗ nur Bellman-Ford ◯ keiner



◯ nur Dijkstra ◯ beide
◯ nur Bellman-Ford ⊗ keiner



◯ nur Dijkstra ◯ beide
◯ nur Bellman-Ford ⊗ keiner

Punkte:
4 P für richtige Kreuze, (**1 P** pro Kreuz)
2 P für (nächste oder falsche) Kantenmarkierungen (**0.5 P** pro Kante). Für den Graphen oben rechts ist E-G eine weitere mögliche Kante.

(c) (1 Punkt) Nehmen Sie nun an, dem Graphen wird eine **neue** Kante hinzugefügt: eine Kante von Knoten **C** nach Knoten **A** mit dem Gewicht **0**.

Erklären Sie kurz und präzise, welches **fundamentale Problem** für die Suche nach einem kürzesten Pfad durch diese neue Kante entsteht. Nennen Sie den **problematischen Pfad** und sein **Gesamtgewicht**.

Durch die neue Kante entsteht ein **negativer Zyklus**.
Der Zyklus ist $A \to B \to C \to A$. Sein Gesamtgewicht beträgt $w(A,B) + w(B,C) + w(C,A) = 1 + (-2) + 0 = -1$.
In einem Graphen mit negativem Zyklus ist das Problem des kürzesten Weges nicht mehr wohldefiniert, da man den Zyklus unendlich oft durchlaufen und die Pfadkosten damit beliebig senken kann.

Punkte:

1 Punkte insgesamt.

- **0.5 P** für die Nennung des Problems (negativer Zyklus").
- **0.5 P** für die korrekte Angabe des Zyklus ($A \to B \to C \to A$) (und dessen korrektes negatives Gesamtgewicht (-1)).

(d) (1 Punkt) Der gegebene Graph hat sieben Knoten. Beschreiben Sie, wie der Bellman-Ford-Algorithmus das in der vorigen Teilaufgabe genannte Problem in der **siebten Iteration** erkennt.

Der Bellman-Ford-Algorithmus führt nach den regulären $|V| - 1 = 6$ Iterationen eine weitere, siebte ($|V|$-te) Iterationsrunde durch. Wenn sich in dieser zusätzlichen Runde die Distanz zu mindestens einem Knoten immer noch verbessern lässt (d.h. wenn eine Kante (u, v) erfolgreich relaxiert werden kann), ist dies der Beweis für die Existenz eines von der Quelle aus erreichbaren negativen Zyklus.

Punkte:

1 Punkte insgesamt.

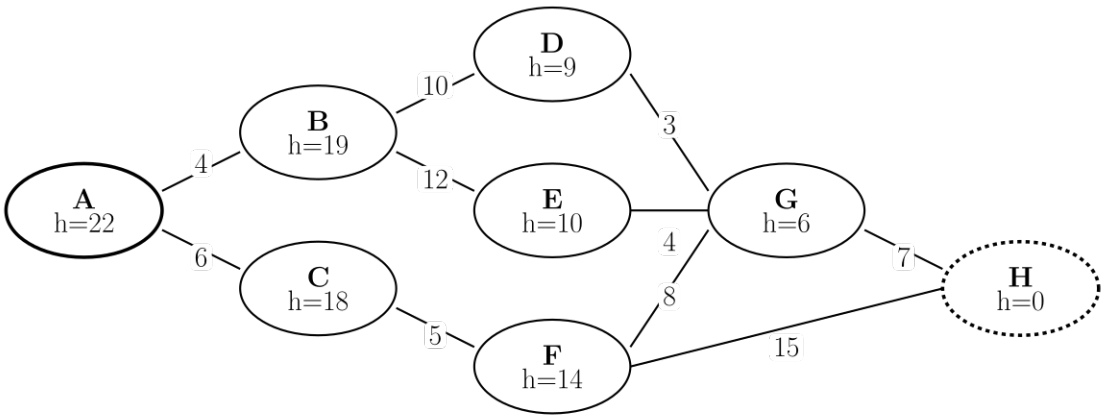
- **0.5 P** für die Erwähnung, dass eine **zusätzliche Iteration** (die $|V|$-te) nach den standardmäßigen $|V| - 1$ Iterationen durchgeführt wird.
- **0.5 P** für die Erklärung, dass eine **weitere Distanzverbesserung** / erfolgreiche Relaxierung in dieser Zusatzrunde den negativen Zyklus nachweist.

Aufgabe 8: Heuristische und randomisierte Algorithmen (10 Punkte)

(a) (5 Punkte) Sie sind für die Logistikplanung eines Online-Versandhändlers verantwortlich. Ihre Aufgabe ist es, die schnellste Lieferroute zwischen zwei Verteilzentren zu finden.

Gegeben ist ein Graph, der die Verteilzentren (Knoten) und die Transportwege (Kanten) darstellt. Die Kantengewichte entsprechen der Lieferzeit in Stunden. Die heuristischen h-Werte schätzen die verbleibende Lieferzeit bis zum Ziel-Verteilzentrum **H**.

Ihre Aufgabe ist es, den A*-Algorithmus auszuführen, um den schnellsten Weg von **Verteilzentrum A** nach **Verteilzentrum H** zu finden.



Hinweis zur Erinnerung:

- $g(n)$-Wert: Die Kosten des bisher kürzesten bekannten Pfades vom Startknoten zum Knoten n.
- $h(n)$-Wert: Eine Schätzung/Heuristik der Kosten vom Knoten n zum Ziel.
- $f(n)$-Wert: Eine Schätzung der Gesamtkosten vom Start zum Ziel über n. Formel: $f(n) = g(n) + h(n)$.

Protokollieren Sie die Ausführung des Algorithmus, indem Sie die folgenden zwei Tabellen vervollständigen.

1. Entwicklung der Priority-Queue (PQ)

Notieren Sie den Zustand der Priority-Queue (`PQ`) **vor** Beginn jeder Iteration. Tragen Sie die Knoten, mit Komma getrennt, ein. Geben Sie den f-Wert eines Knotens in der `PQ` in Klammern an. Zum Beispiel: A(22), C(24).

Regel: Sortieren Sie die `PQ` nach aufsteigenden f-Werten. Bei gleichen f-Werten sortieren Sie alphabetisch.

Iteration	`PQ` vor der Iteration
1	A(22)
2	B(23), C(24)
3	D(23), C(24), E(26)
4	G(23), C(24), E(26)
5	C(24), H(24), E(26), F(39)
6	H(24), F(25), E(26)

2. Untersuchte Knoten und g-Werte

Geben Sie in jeder Iteration den aktuell aus der `PQ` entnommenen und untersuchten Knoten an. Tragen Sie für dessen Nachbarn die **neu berechneten oder verbesserten** g-Werte in die jeweilige Spalte ein. Bereits ermittelte und nicht verbesserte g-Werte müssen nicht erneut eingetragen werden.

Iter.	Knot.	$g(A)$	$g(B)$	$g(C)$	$g(D)$	$g(E)$	$g(F)$	$g(G)$	$g(H)$
0	/	0	∞	∞	∞	∞	∞	∞	∞
1	A		4	6					
2	B				14	16			
3	D							17	
4	G						25		24
5	C						11		
6	H								

3. Ergebnis

Geben Sie die von Ihnen gefundene schnellste Route als Sequenz von Knoten an:

A → B → D → G → H

Punkte:

2 P für die korrekte Entwicklung der `PQ` (0.5 P Abzug pro falscher Zeile).
2.5 P für die korrekte zweite Tabelle (0.5 P Abzug pro falscher Zeile).
0.5 P für die korrekte Angabe der finalen Route.

(b) (3 Punkte) Betrachten Sie ein unendlich großes 2D Gitter, in dem man sich von einer Zelle zu jeder benachbarten Zelle (horizontal oder vertikal, **nicht** diagonal) bewegen kann. Jede Bewegung **kostet 1**. Das Ziel ist es, einen **kürzesten Weg** vom Startknoten S zum Zielknoten G zu finden. Für einen beliebigen Knoten n mit Koordinaten (x_n, y_n) und den Zielknoten G mit Koordinaten (x_G, y_G) werden die folgenden zwei Heuristiken zur Schätzung der verbleibenden Kosten vorgeschlagen:

- **Heuristik 1 (Manhattan-Distanz):** $h_1(n) = |x_n - x_G| + |y_n - y_G|$
- **Heuristik 2 (Euklidische Distanz):** $h_2(n) = \sqrt{(x_n - x_G)^2 + (y_n - y_G)^2}$

Prüfen Sie für **beide** Heuristiken (h_1 und h_2), ob sie für das beschriebene Problem **zulässig** (*admissible*) sind. **Begründen** Sie Ihre jeweilige Antwort kurz. Ein formeller Beweis ist nicht erforderlich.

Eine Heuristik h ist zulässig, wenn sie die tatsächlichen Kosten h^* zum Ziel nie überschätzt, d.h. $h(n) \leq h^*(n)$ für alle Knoten n.

h1 (Manhattan-Distanz): Ist zulässig. Die tatsächlichen Kosten $h^*(n)$ in einem Gitter ohne diagonale Züge sind genau die Summe der horizontalen und vertikalen Schritte, also die Manhattan-Distanz. Somit gilt $h_1(n) = h^*(n)$.

h2 (Euklidische Distanz): Ist zulässig. Die euklidische Distanz ist die direkte Luftlinie. Da man sich nur entlang des Gitters bewegen kann, ist jeder Pfad mindestens so lang wie die Luftlinie. Es gilt immer $|a| + |b| \geq \sqrt{a^2 + b^2}$. Somit gilt $h_2(n) \leq h_1(n) = h^*(n)$.

Punkte:

0.5 P h_1 ist zulässig.
1 P für korrekte Beurteilung und Begründung der Zulässigkeit von h_1.
0.5 P h_2 ist zulässig.
1 P für korrekte Beurteilung und Begründung der Zulässigkeit von h_2.

(c) (2 Punkte) Ordnen Sie die folgenden Algorithmen einer der folgenden Klassen zu:

- **Approximativer Algorithmus (Approx. Alg.)**
- **Deterministischer Algorithmus (Determ. Alg.)**
- **Las-Vegas (LV)**
- **Monte-Carlo (MC)**

Kreuzen Sie für jeden Algorithmus die zutreffende Klasse in der Tabelle an.

Algorithmus	**Approx. Alg.**	**Determ. Alg.**	**LV**	**MC**
Quicksort mit zufälligem Pivot-Element	☐	☐	☒	☐
Zur Approximation eines Funktionsminimums werden exakt 10 Iterationen einer zufälligen lokalen Suche durchgeführt. Das beste bisher gefundene Ergebnis wird zurückgegeben.	☐	☐	☐	☒
Aus einem Sack mit 100 unsortierten Socken wird eine zufällige Teilmenge von 20 Socken entnommen. Aus dieser Teilmenge wird der Socken ausgewählt, der einem gesuchten Muster am besten entspricht.	☐	☐	☐	☒
Aus einem Sack unsortierter Socken werden nacheinander einzelne Socken gezogen und geprüft, bis der exakt passende Socken für einen gegebenen Einzelsocken gefunden wurde.	☐	☐	☒	☐

Algorithmus	**Erklärung**
Quicksort mit zufälligem Pivot-Element	Randomisiertes Quicksort liefert immer eine korrekt sortierte Liste (optimales Ergebnis), aber seine Laufzeit hängt von der zufälligen Wahl der Pivot-Elemente ab. Im Erwartungswert ist sie mit $O(n \log n)$ sehr gut, im Worst-Case (durch Pech bei der Wahl) jedoch $O(n^2)$. Dies entspricht exakt der Definition eines Las-Vegas-Algorithmus
Zur Approximation eines Funktionsminimums werden exakt 10 Iterationen einer zufälligen lokalen Suche durchgeführt. Das beste bisher gefundene Ergebnis wird zurückgegeben.	Der Algorithmus hat eine feste, garantierte Laufzeit (Aufwand für 10 Iterationen). Das Ergebnis ist jedoch nicht garantiert optimal; es ist vom Zufall der lokalen Suche abhängig, ob und wie nah das gefundene Minimum am globalen Minimum liegt. Eine garantierte (effiziente) Laufzeit bei einem nur mit gewisser Wahrscheinlichkeit optimalen Ergebnis ist die Definition eines Monte-Carlo-Algorithmus (MC). Es ist kein approx. Algoirthmus da, da keine Approximationsgüte (ρ) garantiert wird.
Aus einem Sack mit 100 unsortierten Socken wird eine zufällige Teilmenge von 20 Socken entnommen. Aus dieser Teilmenge wird der Socken ausgewählt, der einem gesuchten Muster am besten entspricht.	Der Aufwand ist fest (20 Socken prüfen), die Laufzeit ist also garantiert. Die gefundene Lösung (der "besteSSocken aus der 20er-Teilmenge) ist aber mit hoher Wahrscheinlichkeit nicht die global beste Lösung (der beste Socken aus allen 100). Dies ist das Kennzeichen eines Monte-Carlo-Algorithmus.
Aus einem Sack unsortierter Socken werden nacheinander und zufällig einzelne Socken gezogen und geprüft, bis der exakt passende Socken für einen gegebenen Einzelsocken gefunden wurde.	Das Ziel ist es, den exakt passenden Socken zu finden. Der Algorithmus terminiert erst, wenn dieses korrekte (optimale) Ergebnis gefunden wurde. Die dafür benötigte Zeit (Anzahl der gezogenen Socken) ist jedoch zufällig und nicht im Voraus bekannt. Es könnte der erste gezogene Socken sein oder der letzte. Ein garantiert korrektes Ergebnis bei zufälliger Laufzeit ist die Definition eines Las-Vegas-Algorithmus.

Punkte:

0.5 P pro richtige Antwort (insgesamt **2 Punkte**)

Aufgabe 9: Hashing (10 Punkte)

(a) (4 Punkte) Sie evaluieren verschiedene Hashing-Strategien. Sie untersuchen das Verhalten für eine Hashtabelle der Größe $M = 7$ unter Verwendung der **Divisions-Rest-Methode** als Hashfunktion, also:

$$h(k) = k \mod 7.$$

Die folgende Sequenz von Schlüsseln soll in der angegebenen Reihenfolge in eine anfangs leere Tabelle eingefügt werden:

`[2, 5, 9, 12, 16, 19, 23].`

Simulieren Sie das Einfügen der obigen Schlüssel für zwei verschiedene Kollisionsstrategien. Stellen Sie den **finalen** Zustand der Hashtabelle nach dem Einfügen aller sieben Schlüssel für beide Verfahren dar.

1. **Hashing mit Verkettung (Separate Chaining):** Neue Schlüssel werden stets am **Anfang** der jeweiligen Liste eingefügt.
2. **Lineare Sondierung (Linear Probing):** Nutzen Sie die folgende Sondierungsfolge: $(h(k) + n) \pmod M$ für $n = 0, 1, 2, \ldots$.

Füllen Sie dazu die beiden untenstehenden Tabellen aus.

1. Hashing mit Verkettung

Hinweis:Stellen Sie die Liste dar, indem Sie die Schlüssel mit Pfeilen verbinden (z.B. k1 → k2)

Index	Verkettete Liste (Schlüssel)
0	-
1	-
2	$23 \to 16 \to 9 \to 2$
3	-
4	-
5	$19 \to 12 \to 5$
6	-

2. Lineare Sondierung

Index	Schlüssel
0	19
1	23
2	2
3	9
4	16
5	5
6	12

Trace für Lineare Sondierung:

- `put(2)`: $h(2) = 2$, an Index 2.
- `put(5)`: $h(5) = 5$, an Index 5.
- `put(9)`: $h(9) = 2$. Kollision! Nächster freier Platz: Index 3.
- `put(12)`: $h(12) = 5$. Kollision! Nächster freier Platz: Index 6.

- `put(16)`: $h(16) = 2$. Kollision! Plätze 2, 3 belegt. Nächster freier Platz: Index 4.
- `put(19)`: $h(19) = 5$. Kollision! Plätze 5, 6 belegt. Nächster freier Platz: Index 0 (wrap-around).
- `put(23)`: $h(23) = 2$. Kollision! Plätze 2, 3, 4, 5, 6, 0 belegt. Nächster freier Platz: Index 1.

Punkte:

- **2 P** für die korrekte finale Tabelle beim Hashing mit Verkettung. -0.5 P für falsche Listenreihenfolge oder nicht erkennbare Listenrichtung. -0.5 P für jeden falschen Eintrag.
- **2 P** für die korrekte finale Tabelle bei der Linearen Sondierung. -0.5 P für jeden falschen Eintrag.

(b) (2 Punkte) Betrachten Sie Ihr Ergebnis der **Linearen Sondierung** aus Teil a). Das dort sichtbare Phänomen der primären Häufung entsteht, weil auch Schlüssel mit *unterschiedlichen* Hash-Werten um dieselben freien Plätze konkurrieren.

Erklären Sie diesen Effekt kurz am Beispiel der Einfügung von Schlüssel `16` und `19`.

Schlüssel `16` (hash `2`) muss sondieren und belegt Platz 4. Später muss Schlüssel `19` (hash `5`) ebenfalls sondieren. Sein Sondierungspfad $5 \to 6 \to 0$ führt ihn direkt in den "Wirkungsbereich" des Clusters, der um Hash-Wert `2` entstanden ist. Dadurch verschmelzen die Cluster, die um die Hash-Werte `2` und `5` entstehen, zu einem einzigen großen Cluster.

Punkte:

- **1 P** für die korrekte Beschreibung, dass der Sondierungspfad eines Schlüssels in den Bereich eines anderen Clusters führt.
- **1 P** für die Schlussfolgerung, dass dadurch Cluster verschmelzen und ein größerer, zusammenhängender Block entsteht.

(c) (2 Punkte) Der Belegungsfaktor beider Hashtabellen in a) beträgt nach dem Einfügen aller Schlüssel $\alpha = 1$.

Vergleichen Sie kurz die Konsequenz dieses Zustands für eine **erfolglose Suche** bei beiden Verfahren (**lineare Sondierung** und **Verkettung**). Welches Verfahren ist **robuster** gegenüber diesem hohen Belegungsgrad und warum?

- **Lineare Sondierung:** Eine erfolglose Suche prüft im schlimmsten Fall die gesamte Tabelle ($O(M)$ Komplexität). Das Verfahren ist **nicht robust**, da ein weiteres Einfügen unmöglich ist, ohne die Tabelle zu vergrößern (Rehashing).
- **Hashing mit Verkettung:** Eine erfolglose Suche benötigt im Durchschnitt $\alpha = 1$ Vergleiche. Das Verfahren ist **robust**, da $\alpha > 1$ problemlos möglich ist; weitere Schlüssel werden einfach in die Listen eingefügt.

- **TLDR**: Bei linearer Sondierung aggregieren sich die Kollisionen (Clustering) stärker als bei Verkettung, da die nächsten Schlüssel ebenfalls mit hoher Wahrscheinlichkeit kollidieren.

Punkte:

- **1 P** für den korrekten Vergleich der erfolglosen Suche ($O(M)$ bei Sondierung vs. $O(\alpha)$ bei Verkettung).
- **1 P** für die korrekte Identifikation der Verkettung als robusteres Verfahren mit der Begründung, dass weiteres Einfügen möglich ist (im Gegensatz zur Sondierung).

(d) (2 Punkte) Kreuzen Sie an, ob die folgenden Aussagen wahr oder falsch sind. Eine Begründung ist nicht erforderlich.

Aussage	**Wahr**	**Falsch**
Quadratisches Sondieren vermeidet primäre Häufungen, die bei linearem Sondieren auftreten. Das Problem der sekundären Häufung bleibt jedoch bestehen, da Schlüssel mit demselben initialen Hashwert stets derselben Sondierungssequenz folgen.	☒	☐
Bei der Divisions-Rest-Methode ($h(k) = k \mod M$) ist die Wahl von M als eine Zweierpotenz (z.B. $M = 64$) besonders empfehlenswert, da hierbei alle Bits des Schlüssels für die Berechnung der Hashadresse berücksichtigt werden, was zu einer guten Streuung führt.	☐	☒
Beim quadratischen Sondieren ist, im Gegensatz zur linearen Sondierung, nicht immer garantiert, dass die Sondierungsfolge alle (M) Adressen der Tabelle durchläuft. Daher kann die Suche nach einem freien Platz fehlschlagen, obwohl die Tabelle noch nicht voll ist ($\alpha < 1$).	☒	☐
Gemäß dem Vertrag in Java gilt: Wenn für zwei Objekte `a` und `b` die Bedingung `a.equals(b)` zutrifft, dann *muss* auch `a.hashCode() == b.hashCode()` gelten. Die Umkehrung, also dass aus gleichen Hashcodes die Gleichheit der Objekte folgt, ist hingegen nicht zwingend erforderlich.	☒	☐

Aussage	**Begründung**
Quadratisches Sondieren vermeidet primäre Häufungen, die bei linearem Sondieren auftreten. Das Problem der sekundären Häufung bleibt jedoch bestehen, da Schlüssel mit demselben initialen Hashwert stets derselben Sondierungssequenz folgen.	**Wahr**: Quadratisches Sondieren vermeidet die Bildung langer, zusammenhängender Cluster (primäre Häufung), da die Sondiervorgehensfolge nicht linear ist. Sekundäre Häufungen treten dennoch auf, weil alle Schlüssel, die auf dieselbe Adresse abgebildet werden, exakt dieselbe alternative Sequenz von Speicherplätzen prüfen, was zu Häufungen führen kann.
Bei der Divisions-Rest-Methode ($h(k) = k \bmod M$) ist die Wahl von M als eine Zweierpotenz (z.B. $M = 64$) besonders empfehlenswert, da hierbei alle Bits des Schlüssels für die Berechnung der Hashadresse berücksichtigt werden, was zu einer guten Streuung führt.	**Falsch**: Die Wahl von M als Zweierpotenz ist eine schlechte Praxis. Die Operation $k \pmod{2^i}$ berücksichtigt nur die i niedrigstwertigen Bits des Schlüssels k. Dies führt oft zu einer schlechten Verteilung und vielen Kollisionen, wenn die Schlüssel in ihren niederwertigen Bits Muster aufweisen. Empfohlen werden Primzahlen, die nicht in der Nähe einer Zweierpotenz liegen.
Beim quadratischen Sondieren ist, im Gegensatz zur linearen Sondierung, nicht immer garantiert, dass die Sondierungsfolge alle (M) Adressen der Tabelle durchläuft. Daher kann die Suche nach einem freien Platz fehlschlagen, obwohl die Tabelle noch nicht voll ist ($\alpha < 1$).	**Wahr**: Die Aussage ist korrekt. Während die lineare Sondierung ((h(k)+n)) garantiert alle (M) Plätze durchläuft, bevor sie sich wiederholt, ist dies beim quadratischen Sondieren nicht zwangsläufig der Fall. Wie im Skript erwähnt (S. 280, basierend auf [Radke, 1970]), durchläuft die Sondierungsfolge nur dann garantiert alle (M) Adressen, wenn die Tabellengröße (M) eine Primzahl der Form (4k+3) ist. Ist diese Bedingung nicht erfüllt (z.B. bei (M=10)), kann die Sondierungsfolge in einen Zyklus geraten, der nur einen Teil der Indizes abdeckt. Sind alle Indizes in diesem Zyklus belegt, meldet der Einfügevorgang fälschlicherweise, dass die Tabelle voll ist, obwohl an anderer Stelle noch freie Plätze existieren könnten. Das Einfügen schlägt also fehl, obwohl $\alpha < 1$ gilt.
Gemäß dem Vertrag in Java gilt: Wenn für zwei Objekte `a` und `b` die Bedingung `a.equals(b)` zutrifft, dann *muss* auch `a.hashCode() == b.hashCode()` gelten. Die Umkehrung, also dass aus gleichen Hashcodes die Gleichheit der Objekte folgt, ist hingegen nicht zwingend erforderlich.	**Wahr**: Dies ist die exakte Definition des `hashCode()-equals()`-Vertrags in Java. Wenn zwei Objekte laut `equals()` gleich sind, müssen ihre Hashcodes identisch sein, um sicherzustellen, dass sie in einer Hashtabelle im selben "Bucket"gefunden werden. Umgekehrt müssen Objekte mit demselben Hashcode nicht gleich sein; dies ist eine Kollision, die von der Hashtabelle korrekt behandelt wird.

Punkte:

0.5 P für jede richtige Antwort