Aufgabe 1: Bibliothekssystem (8 Punkte)

(a) **2 Punkte**

In dieser Aufgabe entwerfen Sie ein Bibliothekssystem. Definieren Sie dafür eine abstrakte Basisklasse `Book` mit den Attributen `title` (String) und `year` (int), sowie einer abstrakten Methode `getInfo()`, die einen String zurückgibt.

Hinweis: Da keine Getter-Methoden implementiert sind, müssen die Attribute der Basisklasse mit dem korrekten Zugriffsmodifizierer deklariert werden, um den Zugriff für erbende Klassen zu ermöglichen, aber für andere Klassen zu verhindern.

```
1  public ____________________ class Book {
2      ____________________ ____________________ ____________________;
3      ____________________ ____________________ ____________________;
4      public Book(String title, int year) {
5          ____________________.____________________ = ____________________;
6          ____________________.____________________ = ____________________;
7      }
8      ____________________ ____________________ ____________________ getInfo();
9  }
```

```
1  public abstract class Book {
2      protected String title;
3      protected int year;
4
5      public Book(String title, int year) {
6          this.title = title;
7          this.year = year;
8      }
9
10     public abstract String getInfo();
11 }
```

Punkte:

0.5 P für `abstract` Schlüsselwort bei Klasse
0.5 P für `protected` Attribute
0.5 P für korrekte Konstruktor-Zuweisungen (this.title = title, this.year = year)
0.5 P für korrekte abstrakte Methode `getInfo()`

(b) **2 Punkte**

Erstellen Sie eine Klasse `Magazine`, die von `Book` erbt und zusätzlich das Attribut `issue` (int) besitzt. Überschreiben Sie die Methode `getInfo()`, sodass sie einen String im Format `Magazin: [title], Issue #[issue] ([year])` zurückgibt. Die eckigen Klammern sollen nicht gedruckt werden.

```
1  public class Magazine ____________________ ____________________ {
2      private int issue;
3      public Magazine(String title, int year, int issue) {
4          ____________________(____________________, ____________________);
5          this.issue = issue;
6      }
7      ____________________
8      public String getInfo() {
9          ____________________________________________________
10         ____________________________________________________;
11     }
12 }
```

```
1  public class Magazine extends Book {
2      private int issue;
3
4      public Magazine(String title, int year, int issue) {
5          super(title, year);
6          this.issue = issue;
7      }
8
9      ____________________
10     public String getInfo() {
11         return "Magazin: " + title + ", Issue #" + issue + " (" + year +
               ")";
12     }
13 }
```

Punkte:

0.5 P für `extends Book`
0.5 P für `super(title, year)` Aufruf
0.5 P für `@Override`
0.5 P für korrekte `getInfo()` Implementierung (Abweichungen bei Leerzeichen sind erlaubt)

(c) **2 Punkte**

Implementieren Sie eine Klasse `Bookshelf`, die beliebige `Book`-Objekte in einer `ArrayList<Book>` verwaltet. Die Klasse soll eine Methode `addBook(Book b)` zum Hinzufügen von Büchern und eine Methode `printAllBooks()` besitzen, die die Informationen aller Bücher mittels ihrer `getInfo()`-Methode ausgibt.

```
1  import java.util.ArrayList;
2  public class Bookshelf {
3      private ArrayList<Book> books;
4      public Bookshelf() {
5          books = ____________________________________________________;
6      }
7      public void addBook(Book b) {
8          books.______________________________________________________;
9      }
10     public void printAllBooks() {
11         for ____________________________________________ {
12             ____________________________________________________;
13         }
14     }
15 }
```

```
1  import java.util.ArrayList;
2
3  public class Bookshelf {
4      private ArrayList<Book> books;
5
6      public Bookshelf() {
7          books = new ArrayList<>(); // new ArrayList<Book>() ist auch
               korrekt
8      }
9
10     public void addBook(Book b) {
11         books.add(b);
12     }
13
14     public void printAllBooks() {
15         for (Book b : books) {
16             System.out.println(b.getInfo());
17             }
18         // Eine klassische for-Schleife ist hier ebenfalls korrekt, z.B.
19         // for (int i = 0; i < books.size(); i++) {
20         //     System.out.println(books.get(i).getInfo());
```


```
21          // }
22      }
23  }
```

Punkte:

0.5 P für `new ArrayList<>()` oder `new ArrayList<Book>()`
0.5 P für `books.add(b)`
0.5 P für korrekte for Schleife
0.5 P für `System.out.println(b.getInfo())` bzw. `System.out.println(books.get(i).getInfo())`
Notiz nach Korrektur: Falls in `public String getInfo()` fälschlicherweise bereits `System.out.println` verwendet wurde und hier nur getInfo aufgerufen wurde, gibt es einen halben Punkt als Folgefehler.

(d) **2 Punkte**

Schreiben Sie eine `main`-Methode, in der ein `Bookshelf` erstellt wird. Fügen Sie anschließend das `Magazin` *Science Weekly* aus dem Jahr 2025 mit der Ausgabennummer 42 hinzu. Geben Sie zum Schluss alle Bücher im Regal über die Konsole aus.

```
1  public class Main {
2      public static void main(String[] args) {
3          Bookshelf shelf = ____________________________________________;
4          Book mag = ____________________________________________;
5          shelf.____________________________________________;
6          shelf.____________________________________________;
7      }
8  }
```

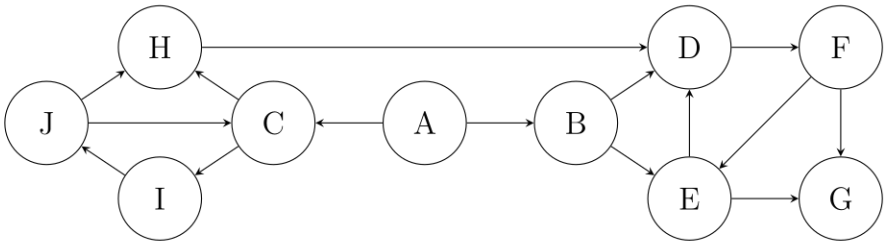
```
1  public class Main {
2      public static void main(String[] args) {
3          Bookshelf shelf = new Bookshelf();
4
5          Book mag = new Magazine("Science Weekly", 2025, 42);
6
7          shelf.addBook(mag);
8
9          shelf.printAllBooks();
10     }
11 }
```

Punkte:

0.5 P für `new Bookshelf()`
0.5 P für korrekte `Magazine`-Erstellung mit allen Parametern
0.5 P für `addBook()`-Aufruf
0.5 P für `printAllBooks()`-Aufruf

Aufgabe 2: Tiefensuche und Laufzeit (11 Punkte)

Der folgende gerichtete Graph ist gegeben.



Führen Sie für die folgenden Aufgaben eine Handsimulation der rekursiven Tiefensuche aus. Beginnen Sie die Tiefensuche im Knoten A. Gehen Sie dabei davon aus, dass bei Wahlmöglichkeiten (Tiebreak) die benachbarten Knoten in alphabetischer Reihenfolge abgearbeitet werden.

(a) (2 Punkte) Geben Sie die **Nebenreihenfolge** (preorder) an.

A B D F E G C H I J

Punkte:

2 P für die richtige Antwort
0.5 P Abzug pro Abweichung. Also ab 4 Abweichungen 0 P.

(b) (2 Punkte) Geben Sie die **Hauptreihenfolge** (postorder) an.

G E F D B H J I C A

Punkte:

2 P für die richtige Antwort
0.5 P Abzug pro Abweichung. Also ab 4 Abweichungen 0 P.

(c) (1 Punkt) Wie kann mit Hilfe der Tiefensuche erkannt werden, ob in einem gerichteten Graphen ein Zyklus vorhanden ist?

Ein Zyklus wird erkannt, wenn der betrachtete Nachbarknoten bereits entdeckt wurde (also in der Preorder steht), aber noch nicht abgearbeitet wurde (also noch nicht in der Postorder steht).

Punkte:

1 P für die richtige Antwort (Preorder und Postorder müssen nicht explizit erwähnt werden).
Nur 0.5 P wenn man die Vorgehensweise für ungerichtete Graphen nennt (Ein Zyklus wird erkannt, wenn der betrachtete Nachbarknoten bereits besucht wurde und nicht der Elternknoten ist).

(d) (2 Punkte) Geben Sie zwei Kanten an, die entfernt werden müssen, damit eine topologische Sortierung möglich ist. Begründen Sie Ihre Auswahl.

Begründung: Topologische Sortierung ist nur bei zyklenfreien Graphen möglich.

Der Graph hat zwei Zyklen. Um diese aufzulösen, muss aus jedem Zyklus eine Kante entfernt werden.

- Zyklus 1 ($C \to I \to J \to C$): Eine von (C, I), (I, J) oder (J, C)
- Zyklus 2 ($D \to F \to E \to D$): Eine von (D, F), (F, E) oder (E, D)

Punkte:

1 P für die richtige Begründung
0.5 P Eine beliebige Kante aus Zyklus 1
0.5 P Eine beliebige Kante aus Zyklus 2

(e) (4 Punkte) Geben Sie für die folgenden Funktionen die asymptotische Laufzeit als engstmögliche obere Schranke in der $\boldsymbol{O}$-Notation in Abhängigkeit von N an.

```java
public static int f1(int N) {
    int sum = 0;
    for (int i = 0; i < N; i++) {
        for (int j = N; j > 0; j = j / 2) {
            sum++;
        }
    }
    return sum;
}
```

Laufzeit:

```java
public static void f2(int N) {
    if (N <= 1) {
        return;
    }
    for (int i = 0; i < N / 2; i++) {
        System.out.println("Hello");
    }
    f2(N / 2);
    f2(N / 2);
}
```

Laufzeit:

```java
public static int f3(int N) {
    if (N <= 0) {
        return 1;
    }
    int result = 0;
    for (int i = 1; i <= N * N; i++) {
        result += i;
    }
    return result + f3(N - 1);
}
```

Laufzeit:

```java
public static int f4(int[] arr) {
    int N = arr.length;
    int sum = 0;
    for (int i = 0; i < N; i++) {
        for (int j = i; j < N; j++) {
            for (int k = 0; k < 5; k++) {
                sum += arr[i] + arr[j];
            }
        }
    }
    return sum;
}
```

Laufzeit:

f1: $\boldsymbol{O}(N \log N)$, da die äußere Schleife N-mal durchläuft und die innere Schleife durch Division von j jeweils halbiert wird, also $\log N$-mal pro Durchlauf der äußeren Schleife. Insgesamt ergibt sich somit $N \cdot \log N$ Iterationen.

f2: $\boldsymbol{O}(N \log N)$. Die for-Schleife hat eine Laufzeit von $\boldsymbol{O}(N)$, denn auch $\boldsymbol{O}(N/2)$ zählt asymptotisch als $\boldsymbol{O}(N)$. Es gibt nun zwei rekursive Aufrufe von f2 mit der halben Eingabegröße. Da N in jedem Rekursionsschritt halbiert wird, erreichen wir nach $\log N$ Schritten $N \leq 1$. In jedem dieser Schritte wird insgesamt Arbeit proportional zu N geleistet. Also: N Arbeit * $\log N$ Schritte = $\boldsymbol{O}(N \log N)$.

Kniff: In jeder Rekursionsebene wird die Funktion zwei Mal aufgerufen – jeweils mit der halben Eingabegröße. Die Anzahl der Aufrufe verdoppelt sich, während die Arbeit pro Aufruf halbiert wird. Dadurch bleibt die Gesamtarbeit pro Ebene bei $\boldsymbol{O}(N)$. Das wird im Rekursionsbaum deutlich.

Tiefe (d)	**Anzahl Aufrufe**	**Arbeit pro Ebene**
0	1 Aufruf mit Größe N	$\boldsymbol{O}(N)$ für for-Schleife
1	2 Aufrufe mit Größe $N/2$	$2 \times \boldsymbol{O}(N/2) = \boldsymbol{O}(N)$
2	4 Aufrufe mit Größe $N/4$	$4 \times \boldsymbol{O}(N/4) = \boldsymbol{O}(N)$
...	...	...
$\log N$	2^d Aufrufe mit Größe 1	$2^d \times \boldsymbol{O}(N/2^d) = \boldsymbol{O}(N)$

f3: $\boldsymbol{O}(N^3)$, da in jedem Rekursionsschritt eine Schleife mit N^2 Durchläufen ausgeführt wird und die Rekursion N-mal aufgerufen wird (von N bis 0). Damit ergibt sich die Gesamtlaufzeit in $\boldsymbol{O}(N^3)$.

f4: $\boldsymbol{O}(N^2)$, wobei N die Länge des Arrays ist. Die beiden äußeren Schleifen durchlaufen zusammen $\boldsymbol{O}(N^2)$ Kombinationen von (i, j). Die innerste Schleife führt eine konstante Anzahl an Operationen aus und erhöht die asymptotische Laufzeit daher nicht.

Punkte:

1 P für jede korrekte Laufzeit (ohne Begründung)
Notizen nach Korrektur: Falls die Laufzeit korrekt ist, aber nicht die engstmögliche obere Schranke angegeben wurde (z.B. $\boldsymbol{O}(5N^2)$ statt $\boldsymbol{O}(N^2)$), dann gibt es 0.5 Punkte statt 1 Punkt.

Aufgabe 3: Greedy-Algorithmen und Backtracking (12 Punkte)

(a) (3 Punkte) Ordnen Sie die unten stehenden Algorithmus-Typen den richtigen Aussagen zu.

Aussage	beide	Greedy	Back-tracking	keine
Die Lösungsstrategie basiert auf der schrittweisen Konstruktion eines Pfades in einem impliziten Lösungsbaum.	☒	☐	☐	☐
Die zu treffenden Entscheidungen werden durch ein lokales Optimum bestimmt und später nicht mehr revidiert.	☐	☒	☐	☐
Dieser Algorithmus-Typ kann als allgemeine Methode verwendet werden, um eine optimale Lösung zu garantieren, indem der Lösungsraum systematisch durchsucht wird.	☐	☐	☒	☐
Der Algorithmus kann genutzt werden, um alle möglichen Lösungen eines Problems zu finden.	☐	☐	☒	☐
Algorithmen dieser Klasse sind sehr effizient, da ihre Laufzeit typischerweise polynomial ist (z.B. $\boldsymbol{O}(n \log n)$ oder $\boldsymbol{O}(n^2)$)	☐	☒	☐	☐
Der Algorithmus-Typ kann durch Pruning-Techniken effizienter gestaltet werden, ohne die Korrektheit der Lösung zu beeinträchtigen.	☐	☐	☒	☐

Punkte:

0.5 P pro richtige Antwort

(b) (9 Punkte) Bei einer Kanu-Tour auf einem Fluss sollen **möglichst viele** Seitenarme durchfahren werden, da dort gebadet werden kann. Jeder Seitenarm ist durch einen Start- und einen Endpunkt auf dem Hauptfluss definiert (siehe Abbildung 2 auf der nächsten Seite).

Es gelten folgende Regeln:

- Aufgrund der starken Strömung können Sie den Hauptfluss nur in eine Richtung befahren. Der Hauptfluss kann als Zeitachse verstanden werden.
- Ein Seitenarm kann nicht befahren werden, wenn er sich zeitlich mit einem bereits ausgewählten Seitenarm überschneidet. Das heißt, der Startpunkt eines neuen Seitenarms muss nach dem Endpunkt des zuletzt befahrenen liegen.

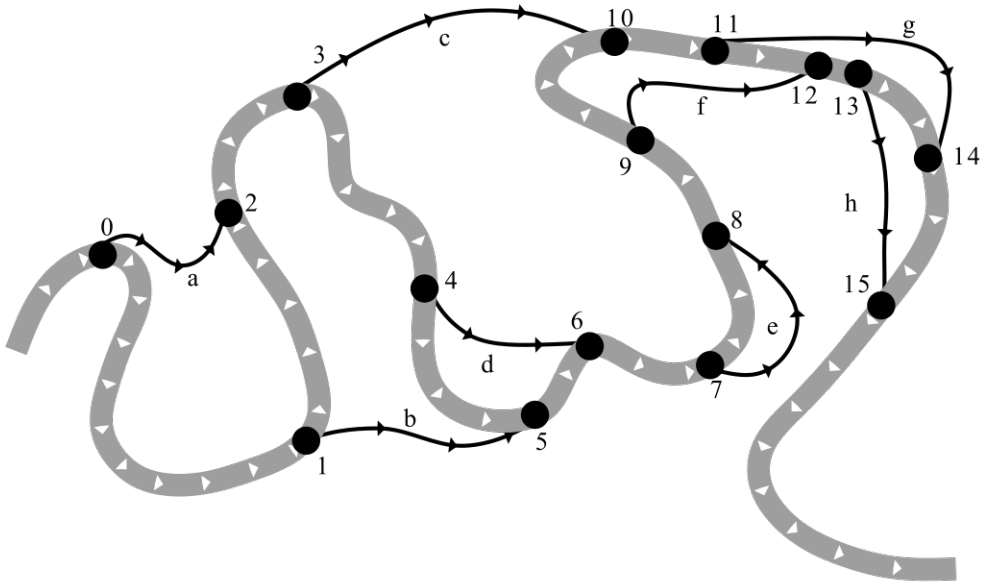


Abbildung 2: Ein Beispiel für den Flussverlauf. Der Hauptfluss ist in hellgrau gekennzeichnet. Seitenarme (a bis h) sind durch schwarze Linien, mit chronologischen Start- und Endzeiten gekennzeichnet.

1. (1 Punkt) Benennen Sie das aus der Vorlesung bekannte Greedy-Problem, auf das sich diese Aufgabenstellung zurückführen lässt.
2. (3 Punkte) Beschreiben Sie schrittweise den Greedy-Algorithmus zur Lösung des Problems. Nennen Sie dabei explizit die Sortier- und die Auswahlstrategie.
3. (5 Punkte) Wenden Sie den beschriebenen Greedy-Algorithmus auf die Seitenarme in der Abbildung an. Geben Sie die finale Reihenfolge der ausgewählten Seitenarme an und bestimmen Sie die maximale Anzahl der durchfahrbaren Seitenarme.

1. Überführung in ein bekanntes Greedy-Problem und Algorithmusbeschreibung

Das Problem der Kanu-Tourplanung zur Maximierung der Seitenarm-Nutzung ist eine klassische Instanz des Intervallauswahlproblems.

2. Algorithmusbeschreibung

Vorgehensweise des Greedy-Algorithmus:

Sortierung: Sortieren Sie alle Seitenarme aufsteigend nach ihren Endpunkten auf dem Hauptfluss.

Die Greedy Auswahlstrategie ist “früheste mögliche Endzeit”.

- Wählen Sie den Seitenarm aus, der als Erster endet (gemäß der Sortierung).
- Fügen Sie diesen Seitenarm zu Ihrer Tour hinzu.
- Entfernen Sie alle Seitenarme aus der Betrachtung, die sich mit dem gerade ausgewählten Seitenarm überlappen (d.h., deren Startpunkt vor dem Endpunkt des ausgewählten Seitenarms liegt).
- Wiederholen Sie die Schritte, bis keine Seitenarme mehr ausgewählt werden können.

Dieser Greedy-Algorithmus liefert eine optimale Lösung, da die Wahl des am frühesten endenden Seitenarms den maximalen Platz für nachfolgende, nicht überlappende Seitenarme lässt.

3. Handsimulation

Sortiert nach Endpunkt:

a (2), b (5), d (6), e (8), c (10), f (12), g (14), h (15)

Auswahlprozess:

- *1. Wähle a [0, 2]*
  - Ende: 2.
  - Entferne: b (Start $1 \leq 2$).
- *2. Wähle d [4, 6]* (nächster verfügbarer nach a mit Start > 2)
  - Ende: 6.
  - Entferne: c (Start $3 \leq 6$).
- *3. Wähle e [7, 8]* (nächster verfügbarer nach d mit Start > 6)
  - Ende: 8.
  - Entferne: Nichts (f,g,h starten alle > 8).
- *4. Wähle f [9, 12]* (nächster verfügbarer nach e mit Start > 8)
  - Ende: 12.
  - Entferne: g (Start $11 \leq 12$).
- *5. Wähle h [13, 15]* (nächster verfügbarer nach f mit Start > 12)
  - Ende: 15.
  - Entferne: Nichts.

Optimale Reihenfolge der ausgewählten Seitenarme: (a, d, e, f, h)

Punkte:

Teil 1:
1 P für die Nennung des Intervallauswahlproblems/Interval Scheduling Problems (oder auch nur "Scheduling")

Teil 2:
1 P für korrekte Sortierstrategie (aufsteigend nach Endpunkt)
1 P für korrekte Greedy-Auswahlstrategie (früheste Endzeit)
1 P für Entfernung von überlappenden Seitenarmen

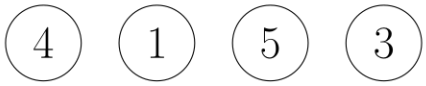
Teil 3:
1 P für Sortierung nach Endzeit (a, b, d, e, c, f, g, h) (Sortierung muss vollständig korrekt sein)
3 P für korrekte Reihenfolge (a, d, e, f, h) (1.5 Punkte Abzug pro Fehler)
1 P für korrekte Anzahl der nutzbaren Seitenarme (5)

Notizen nach Korrektur:

- Es ist auch ok, wenn die Sortierung nach spätester Startzeit erfolgt. Kommt auf gleiches Endergebnis.
- Wenn die ausgewählten Seitenarme stimmen, aber die Reihenfolge falsch ist, gibt es 1 Punkt Abzug.
- Wenn das Endergebnis richtig ist, aber die Sortierung nach Endzeit nicht explizit notiert wurde, ist das in Ordnung. Dann wird der Punkt für die Sortierung trotzdem gegeben, da sie eine Voraussetzung ist, um den Algorithmus korrekt durchzuführen.

Aufgabe 4: Das Münzspiel (13 Punkte)

Alex und Adrian spielen folgendes Münzspiel. Auf dem Tisch liegt eine Reihe von Münzen. Die folgende Abbildung zeigt die aktuelle Spielstellung mit vier Münzen mit den Werten 4, 1, 5 und 3.



Die Spieler ziehen abwechselnd Münzen aus der Reihe. In jedem Zug darf der aktive Spieler entweder die Münze am linken oder am rechten Ende der Reihe entnehmen. Das Spiel endet, wenn alle Münzen genommen wurden. Das Ziel jedes Spielers ist es, am Ende des Spiels einen möglichst großen Vorsprung an Münzwerten gegenüber dem Gegner zu erzielen.

Alex beginnt das Spiel und muss sich entscheiden, ob er die linke Münze (4) oder die rechte Münze (3) nimmt. Er verwendet den **Minimax-Algorithmus**, um seinen optimalen Zug zu finden.

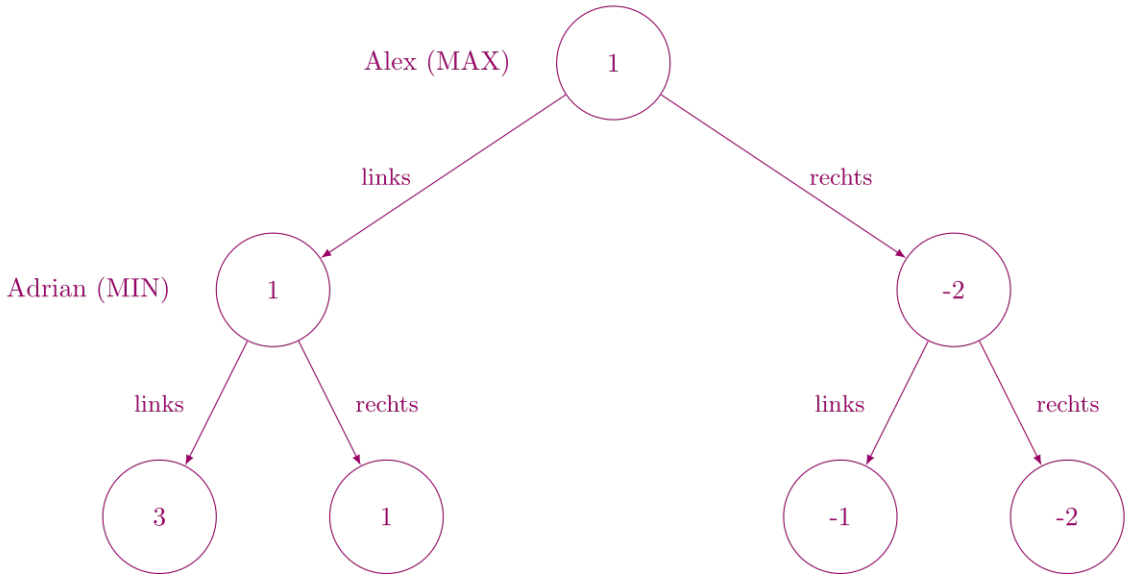
(a) (1 Punkt) Geben Sie die **Bewertungsfunktion** an, die den Vorsprung von Alex gegenüber Adrian berechnet. Alex maximiert diesen Wert, während Adrian ihn minimiert. Geben Sie die Bewertungsfunktion in Abhängigkeit der von Alex und Adrian gesammelten Münzwerte S_{Alex} und S_{Adrian} an.

$$\text{Wert} = S_{\text{Alex}} - S_{\text{Adrian}}$$

Punkte:

1P Korrekte Bewertungsfunktion (Differenz $S_{\text{Alex}} - S_{\text{Adrian}}$). Die Angabe von nur S_{Alex} ist nicht zulässig, da der Vorsprung quantifiziert werden muss.

(b) (3 Punkte) Alex plant seinen Zug nur auf Basis der nächsten beiden Züge (ein Zug von Alex, gefolgt von einem Zug von Adrian). Vervollständigen Sie den folgenden Spielbaum und finden Sie Alex optimalen Zug mit dem Minimax-Algorithmus.



Alex optimaler Zug: Alex optimaler erster Zug ist, die linke Münze (4) zu nehmen.

Erreichter Wert: 1

Punkte:

1P korrekte Werte in den Blattknoten (alle Werte müssen korrekt sein)
1P korrekte Werte in den Elternknoten (alle Werte müssen korrekt sein)
0.5P korrekte optimale Lösung ("links" oder "linke Münze", oder "4" ist ok)
0.5P korrekter Wert (1) (Klarstellung: Es ist auch ok, wenn hier 4 angegeben wird, da 4 der Wert der linken Münze ist. Es ist aus der Angabe nicht eindeutig, ob der Wert nach Bewertungsfunktion oder nach Münzwert angegeben werden soll.)
Notizen nach Korrektur: falls Blattknoten falsch waren, können die letzten 2 Punkte (1 + 0.5 + 0.5) immer noch erreicht werden, wenn MinMax richtig angewendet wurde.

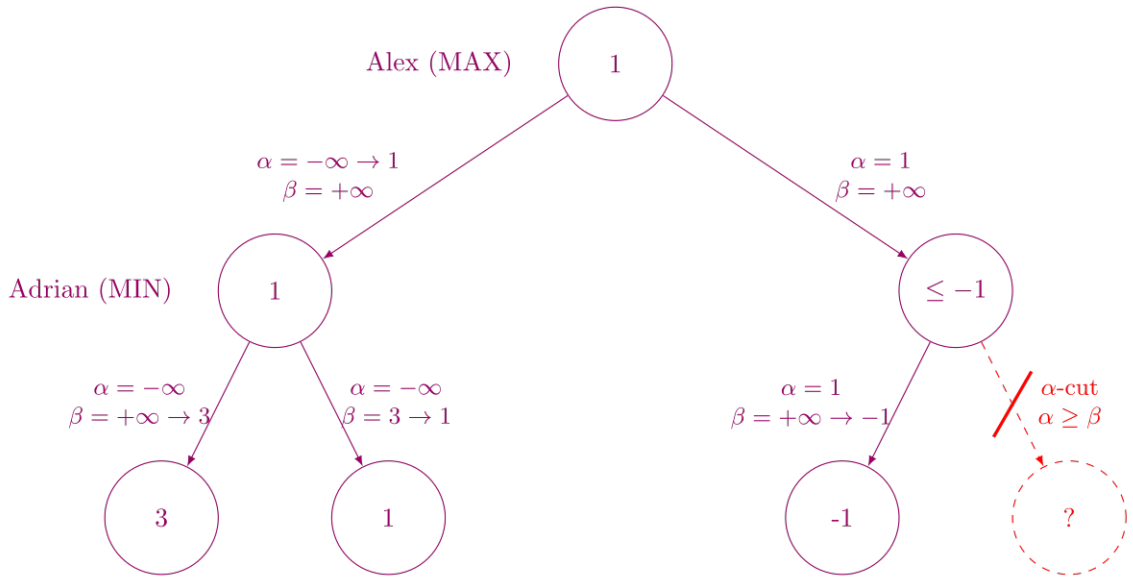
(c) (1 Punkt) Nennen Sie den in der Vorlesung besprochenen Algorithmus, der die Minimax-Suche durch das Beschneiden von Teilbäumen effizienter macht.

Alpha-Beta-Suche

Punkte:

1P Richtiges Ergebnis (zulässig: Alpha-Beta-Suche, Alpha-Beta-Schnitt, Alpha-Beta-Pruning, α-β-Schnitt, α-β-Pruning, α-β-Suche).

(d) (6 Punkte) Übertragen Sie die Werte der Blattknoten aus Aufgabe (b) in den folgenden Baum. Führen Sie anschließend den in Aufgabe (c) genannten Algorithmus aus.



Punkte:

1P Es ist klar ersichtlich, dass initial $\alpha = -\infty$ und $\beta = +\infty$.
2P Korrektes Propagieren und Update von α und β (halber Punkt Abzug für jeden falschen Wert). Es muss klar sein, welcher Wert α und welcher β ist. Dies kann durch eine Beschriftung an den Kanten erreicht werden oder durch das Definieren der Reihenfolge in einer Legende. Falls nicht klar ist, welche Werte α und β sind, gibt es hier 0 Punkte.
1P Korrekte Identifizierung der Stelle, an der Pruning stattfindet.
1P Das Abbruchkriterium $\alpha \geq \beta$ wird genannt.
1P Die Beschriftung des Schnitts ist korrekt (α-cut, nicht β-cut).
Notizen nach Korrektur: falls Blattknoten falsch waren, kann hier immer noch volle Punktzahl erreicht werden, wenn Alpha-Beta richtig angewendet wurde. Falls durch falsche Blattknoten kein Schnitt stattfindet, ist das kein Problem, aber es muss angegeben werden, dass kein Schnitt passiert. Wenn kein Schnitt passiert, kann der letzte Punkt (Beschriftung des Schnitts) nicht erreicht werden.

(e) (2 Punkte) Alex und Adrian spielen nun mit einer längeren Münzreihe und planen ihre Züge bis zum Ende des Spiels. Bisher haben sie die möglichen Züge (d.h. die Kindknoten) immer der Reihe nach von links nach rechts ausgewertet. Sie möchten den Spielbaum jedoch so effizient wie möglich durchsuchen.

Nennen Sie eine spezifische Strategie zur Auswahl des nächsten zu untersuchenden Zuges, die anstelle der festen Links-nach-Rechts-Reihenfolge verwendet werden sollte. Begründen Sie kurz, warum diese Strategie die Gesamtanzahl der tatsächlich untersuchten Knoten im Spielbaum reduziert.

Die ideale Strategie ist, die Züge so zu sortieren, dass die besten Züge zuerst ausgewertet werden (z.B. mittels Heuristik, Gierig). Eine gute Heuristik ist bspw. den Zug mit dem höchsten Münzwert zu wählen.

Begründung: Durch das frühe Finden eines guten Zuges werden die Schranken (α und β) schnell sehr eng gesetzt. Dies maximiert die Wahrscheinlichkeit für Schnitte ($\alpha \geq \beta$) in anderen Teilbäumen, was den Suchaufwand reduziert.

Punkte:

1P Nennung der Kernidee: vielversprechende Züge zuerst auswerten (Nennung von Heuristik/Gierig/Beispiel ist nicht nötig).
1P Korrekte Begründung: schnelle Etablierung von engen Schranken (α, β) maximiert das Pruning.

Aufgabe 5: Dynamische Programmierung (10 Punkte)

Es ist eine Matrix der Größe $M \times N$ gegeben. Die Einträge der Matrix sind $\in \mathbb{N}_{>0}$. Mit Hilfe dynamischer Programmierung soll ein Weg vom oberen linken zum unteren rechten Feld gefunden werden, der die Summe aller Zahlen auf seinem Weg **minimiert**, wie in der Abbildung unten dargestellt. Bei jedem Schritt darf nur entweder nach rechts oder nach unten gegangen werden, also von Feld (m, n) zu einem der Felder $(m+1, n)$, $(m, n+1)$, sofern diese Felder innerhalb der Matrix liegen.

1	3	1	2
1	5	1	1
4	2	3	1

Eine Beispielmatrix der Größe 3×4, in der ein Weg vom oberen linken Feld zum unteren rechten Feld eingezeichnet ist. Der Wert des Weges ist die Summe der durchlaufenen Felder, also $1+3+1+1+1+1=8$.

Ziel dieser Aufgabe ist es, eine rekursive Funktion Opt zu bestimmen, welche es erlaubt, das Problem mithilfe dynamischer Programmierung zu lösen.

(a) (3 Punkte) Tragen Sie in die folgende Matrix die Werte einer Opt-Funktion für jedes Feld (m, n) der Matrix aus der Abbildung oben ein. Dies ist der Wert des günstigsten Pfades vom Startfeld $(1, 1)$ zum jeweiligen Feld.

Hinweis: Sie können diese Aufgabe auch lösen, ohne eine korrekte rekursive Funktion zu kennen.

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

Notizen nach Korrektur: Durch die vorgegebene 1 oben links ist eigentlich klar, dass hier ein Top-Down Ansatz gefordert war (i.e. von Start zu Ziel). Wir lassen nun aber auch einen Bottom-Up-Ansatz zu (i.e. von Ziel zu Start). Es gibt nur volle Punktzahl, wenn die 1 oben links durchgestrichen wurde. Falls die 1 oben links nicht durchgestrichen wurde, gibt es 1 Punkt Abzug (also insgesamt noch maximal 2 Punkte).

8	7	4	4
9	8	3	2
10	6	4	1

Punkte:

2 P für größtenteils richtig (2 oder weniger Felder falsch)
1 P für vollständig richtig

(b) (4 Punkte) Definieren Sie eine rekursive Funktion $\textsc{Opt}(m, n)$, die den Wert des günstigsten Weges vom Startfeld $(1, 1)$ zum Feld (m, n) berechnet ($1 \leq m \leq M, 1 \leq n \leq N$). Der Wert eines Feldes (m, n) ist $F(m, n)$. Verwenden Sie eine mathematische Schreibweise und keinen Java- oder Pseudo-Code.

Hier werden exemplarisch zwei Lösungsmöglichkeiten gezeigt.

Ausführliche Lösung:

$$\textsc{Opt}(m,n) = \begin{cases} F(1,1) & \text{falls } m = 1 \text{ und } n = 1, \\ F(m,n) + \textsc{Opt}(m-1,n) & \text{falls } m > 1 \text{ und } n = 1, \\ F(m,n) + \textsc{Opt}(m,n-1) & \text{falls } m = 1 \text{ und } n > 1, \\ F(m,n) + \min\left(\textsc{Opt}(m-1,n), \textsc{Opt}(m,n-1)\right) & \text{falls } m > 1 \text{ und } n > 1. \end{cases}$$

Alternative Lösung (mit unendlich für Felder außerhalb der Matrix):

$$\textsc{Opt}(m,n) = \begin{cases} \infty & \text{falls } m \leq 0 \text{ oder } n \leq 0, \\ F(1,1) & \text{falls } m = 1 \text{ und } n = 1, \\ F(m,n) + \min\left(\textsc{Opt}(m-1,n), \textsc{Opt}(m,n-1)\right) & \text{sonst.} \end{cases}$$

Notizen nach Korrektur: Es ist auch ok, wenn die Indizes bei 0, 0 starten. Außerdem lassen wir auch einen Bottom-Up-Ansatz zu.

Bottom-Up:

$$\textsc{Opt}(m,n) = \begin{cases} F(m,n) & \text{falls } m = M \text{ und } n = N, \\ F(m,n) + \textsc{Opt}(m+1,n) & \text{falls } m < M \text{ und } n = N, \\ F(m,n) + \textsc{Opt}(m,n+1) & \text{falls } m = M \text{ und } n < N, \\ F(m,n) + \min\left(\textsc{Opt}(m+1,n), \textsc{Opt}(m,n+1)\right) & \text{sonst.} \end{cases}$$

Alternative Lösung (Bottom-Up, mit unendlich für Felder außerhalb der Matrix):

$$\textsc{Opt}(m,n) = \begin{cases} \infty & \text{falls } m > M \text{ oder } n > N, \\ F(m,n) & \text{falls } m = M \text{ und } n = N, \\ F(m,n) + \min\left(\textsc{Opt}(m+1,n), \textsc{Opt}(m,n+1)\right) & \text{sonst.} \end{cases}$$

Punkte:

1 P für korrekten Basisfall ($m = 1$ und $n = 1$)

1 P für korrekte Randfälle ($m > 1$ und $n = 1$, $m = 1$ und $n > 1$)

2 P für korrekten Rekursionsfall $m > 1$ und $n > 1$

(c) (2 Punkte) Geben Sie die Größenordnung der Laufzeit des Algorithmus an, wenn die OPT-Werte mit Hilfe einer dynamischen Programmierungstabelle (DP-Tabelle) berechnet werden. Bestimmen Sie zusätzlich die Größenordnung der Laufzeit eines Brute-Force-Algorithmus, der alle möglichen Wege einzeln betrachtet und deren Summen vergleicht, um den minimalen Weg zu finden. Begründen Sie auch hier Ihre Antworten kurz (jeweils 1-2 Sätze).

Laufzeit des Dynamic-Programming-Algorithmus: Es müssen alle $M \times N$ Einträge der Tabelle berechnet werden. Für jeden Eintrag ist eine konstante Anzahl von Rechenoperationen nötig (ein Vergleich und eine Addition). Die Laufzeit beträgt daher insgesamt $\mathcal{O}(M \cdot N)$.

Laufzeit eines Brute-Force-Algorithmus: Um vom oberen linken Feld $(1, 1)$ zum unteren rechten Feld (M, N) zu gelangen, muss man insgesamt $(M-1)$ Schritte nach unten und $(N-1)$ Schritte nach rechts machen. Die Anzahl der möglichen Wege, die diese Bedingungen erfüllen, wächst **exponentiell** mit M und N. Für jeden dieser Wege muss die Summe der Felder berechnet werden, was proportional zur Länge des Weges $(M + N - 2)$ ist. Die Laufzeit beträgt daher insgesamt $\mathcal{O}(2^{M+N})$.

Punkte:

0,5 P für korrekte Laufzeitangabe DP: $\mathcal{O}(M \cdot N)$

0,5 P für eine verständliche und richtige Begründung DP (konstante Arbeit pro Feld)

0,5 P für korrekte Laufzeitangabe Brute-Force: exponentiell oder $\mathcal{O}(2^{M+N})$

0,5 P für eine verständliche und richtige Begründung Brute-Force (exponentielle Anzahl der Wege)

(d) (1 Punkt) Wie verändert sich die Laufzeit des Algorithmus mit DP-Tabelle, wenn nun auch Doppelsprünge erlaubt sind? Das bedeutet, man darf vom Feld (m, n) nicht nur zu den Feldern $(m + 1, n)$ und $(m, n + 1)$, sondern zusätzlich auch zu $(m + 2, n)$ und $(m, n + 2)$ springen. Dabei darf weiterhin nur nach rechts und nach unten gegangen werden. Begründen Sie Ihre Antwort kurz (1-2 Sätze).

Der Algorithmus muss insofern angepasst werden, dass für jedes Feld (m, n) zusätzlich zu den bisherigen zwei möglichen Vorgängerfeldern (von oben, $(m-1, n)$, oder von links, $(m, n-1)$) auch die beiden neuen möglichen Vorgängerfelder (von zwei Feldern oben, $(m-2, n)$, oder von zwei Feldern links, $(m, n-2)$) betrachtet werden. Das heißt, für jedes Feld wird das Minimum aus bis zu vier möglichen Vorgängerfeldern berechnet. Da dies weiterhin eine konstante Anzahl von Möglichkeiten pro Feld ist, ändert sich die asymptotische Laufzeit nicht. **Die Laufzeit beträgt daher insgesamt immer noch $\mathcal{O}(M \cdot N)$.**

Punkte:

0,5 P für korrekte Laufzeitangabe: $\mathcal{O}(M \cdot N)$

0,5 P für eine verständliche und richtige Begründung (immer noch konstante Arbeit pro Feld)

Aufgabe 6: Minimale Spannbäume (7 Punkte)

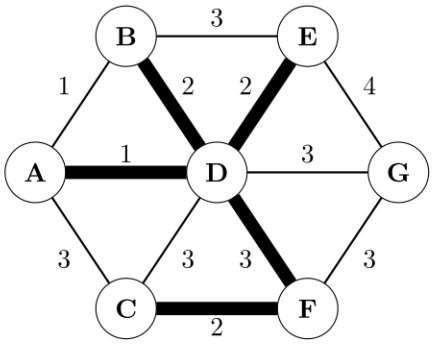
(a) (3 Punkte) Lesen Sie die folgenden Aussagen. Entscheiden Sie, ob sie nur für den Prim-Algorithmus, nur für den Kruskal-Algorithmus, für beide oder für keinen der beiden Algorithmen gelten. Kreuzen Sie das passende Feld an. Pro Aussage ist nur eine Antwort richtig.

Aussage	beide	nur Prim	nur Kruskal	keiner
Der Algorithmus dient der Berechnung eines minimalen Spannbaumes in einem zusammenhängenden, ungerichteten, kantengewichteten Graphen.	☒	☐	☐	☐
Der Algorithmus baut schrittweise einen Spannbaum auf, indem er immer die Kante mit dem minimalen Gewicht auswählt, die den aktuellen Baum mit einem noch nicht besuchten Knoten verbindet und dabei keinen Zyklus erzeugt.	☐	☒	☐	☐
Der resultierende minimale Spannbaum ist immer eindeutig.	☐	☐	☐	☒
Der resultierende Teilgraph nach Beendigung des Algorithmus ist maximal azyklisch.	☒	☐	☐	☐
Für die effiziente Implementierung des Algorithmus ist die Benutzung der Union-Find zur dynamischen Verwaltung von Zusammenhangskomponenten essentiell.	☐	☐	☒	☐
Die Folge der zum Spannbaum hinzugefügten Kantengewichte ist garantiert nicht-absteigend.	☐	☐	☒	☐

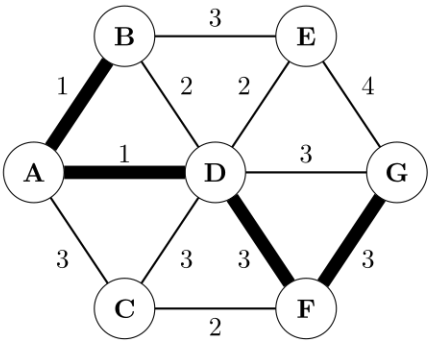
Punkte:

0.5 **P** für jedes richtige Kreuz

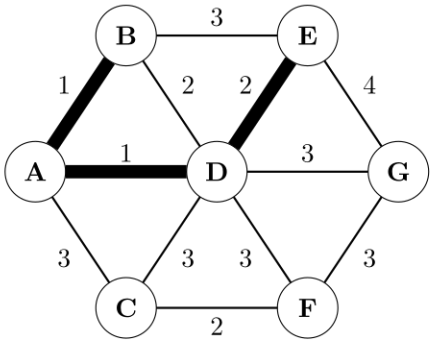
(b) (4 Punkte) In der folgenden Abbildung sehen Sie vier mal den **gleichen** Graphen, wobei jedes mal unterschiedliche Kanten schwarz hervorgehoben sind. Kreuzen Sie an, ob die hervorgehobenen Kanten bei der Suche nach einem minimalen Spannbaum nur durch den Prim Algorithmus, nur durch den Kruskal Algorithmus, 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 partielle Lösungen dargestellt sein.



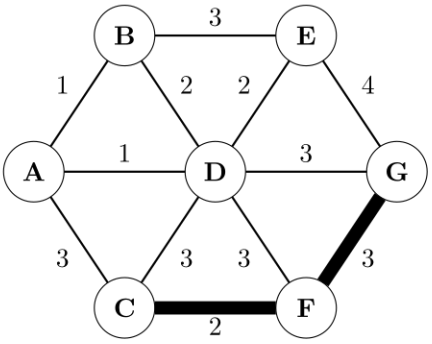
☐ beide ☐ nur Kruskal
☐ nur Prim ☒ keiner



☐ beide ☐ nur Kruskal
☐ nur Prim ☒ keiner



☒ beide ☐ nur Kruskal
☐ nur Prim ☐ keiner



☐ beide ☐ nur Kruskal
☒ nur Prim ☐ keiner

Punkte:

1 P für jedes richtige Kreuz

Aufgabe 7: Kürzeste Pfade (11 Punkte)

Wir betrachten eine Klaviertastatur mit 3 Tasten, die unterschiedliche Töne erzeugen. Die unterschiedlichen Töne nennen wir c, d und e. Wir definieren eine *Melodie* als Abfolge von Tönen.
In dieser Aufgabe suchen wir die *schönste Melodie*, die aus genau 3 Tönen besteht. Für verschiedene Tonübergänge gibt es die in Tabelle 2a angegebenen Kosten. Die Melodie mit der geringsten Summe an Kosten gilt als die *schönste Melodie*. Zusätzlich zu den Übergängen fallen Kosten für den ersten und letzten Ton der Melodie an. Diese sind in Tabelle 2b angegeben.

	zu c	**zu** d	**zu** e
von c	3	5	kein Übergang
von d	1	3	2
von e	kein Übergang	2	3

(a) Kosten für die Tonübergänge.

Ton	c	d	e
Kosten	4	3	1

(b) Kosten für den Start- und Endton

Tabelle 2: Kosten für das Spielen von Tönen.

Beispiel: Die Melodie $c \to d \to e$ hat die Kosten $4 + 5 + 2 + 1 = 12$. Sie startet mit c (Kosten 4), dann folgen zwei Übergänge mit den Kosten 5 und 2. Schließlich endet die Melodie auf e, was zu weiteren Kosten von 1 führt.

(a) (2 Punkte) Modellieren Sie den oben dargestellten Sachverhalt im Graphen auf der nächste Seite. Tragen Sie dazu die korrekten Gewichte an den entsprechenden Kanten ein.

(b) (6 Punkte) Führen Sie eine Handsimulation des Dijkstra-Algorithmus durch, um einen kürzesten Weg von Knoten s nach t zu finden. Füllen Sie dazu die Tabelle auf der nächsten Seite aus und geben Sie für jeden Schritt den ausgewählten Knoten, seine Pfadlänge von s und seinen Vorgänger an.

Hinweis: Haben mehrere Knoten in der Priority Queue die gleiche minimale Distanz, ist die Auswahl in der folgenden Reihenfolge vorzunehmen:

1. Wählen Sie zuerst den alphabetisch kleinsten Knoten (z.B. c_2 vor d_1).
2. Sind Knoten alphabetisch gleich, wählen Sie den Knoten mit dem kleineren numerischen Index (z.B. c_1 vor c_2).

(c) (1.5 Punkte) Wir fügen nun eine weitere Kante $e_3 \to e_1$ in den gezeigten Graphen ein. Geben Sie das kleinstmögliche Gewicht (in $\mathbb{Z}$) für diese Kante an, sodass weiterhin ein kürzester Pfad im Graphen gefunden werden kann. Begründen Sie **kurz** ihre Wahl!

-4. Kleinere Zahlen würden einen negativen Zyklus erzeugen.

Punkte:

0.5 P. für die Zahl
1 P. für die Begründung

(d) (1.5 Punkte) Beschreiben Sie zuerst **kurz**, wie die *schlechteste Melodie* (das Gegenteil der *schönsten Melodie*) gefunden werden kann. Wie muss dazu ein Ihnen bekannter Algorithmus verändert werden?

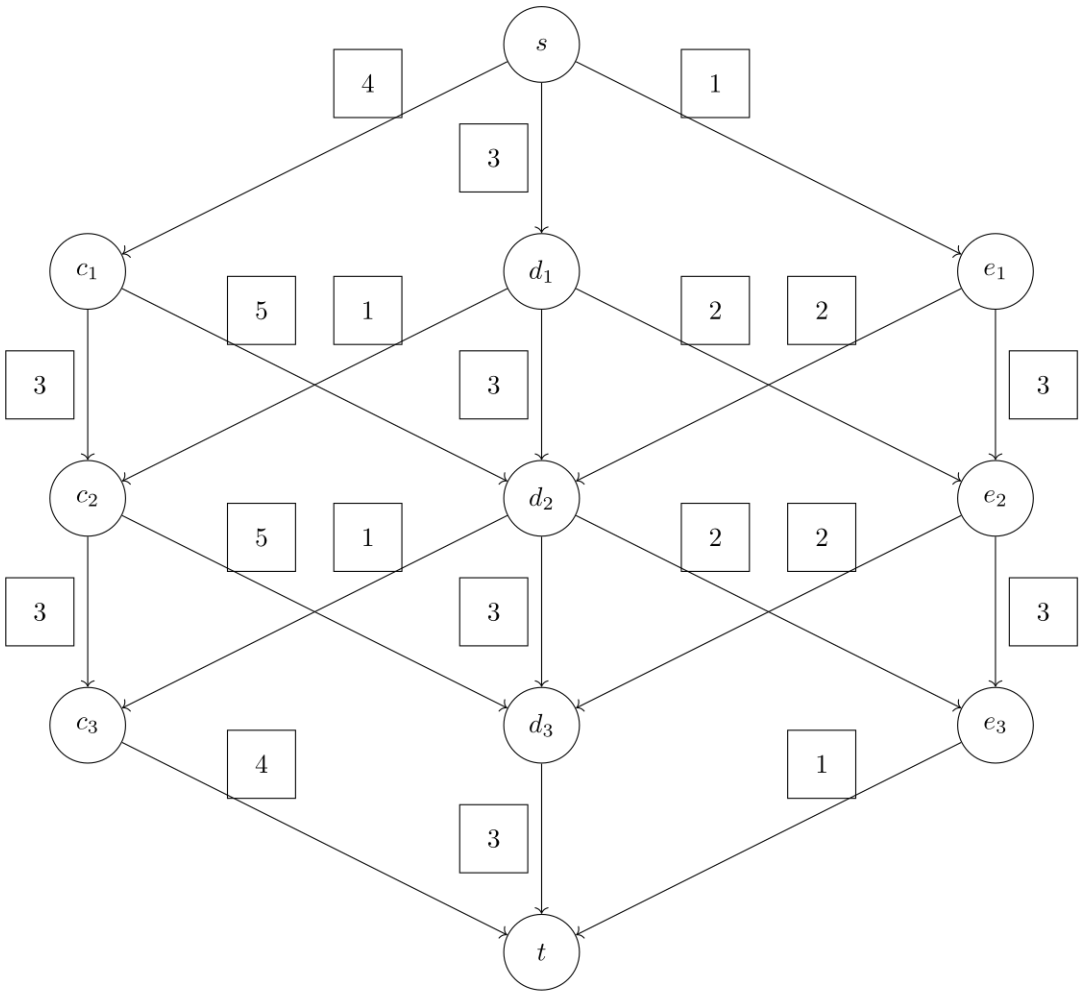
Idee: Statt den kürzesten Pfad einfach den längsten suchen.

Verschiedene mögliche Modifikationen:

- Kantengewichte invertieren, dann Dijkstra oder Bellman-Ford anwenden.
- Einfach die Relaxierung umkehren.

Punkte:

0.5 P. für die korrekte Idee (längsten Pfad suchen)
1.0 P. für eine korrekte Modifikation



Punkte:

zu (a)

Felder korrekt ausgefüllt	Punkte
≥ 5	0.5 P.
≥ 10	1 P.
≥ 15	1.5 P.
alle 20	2 P.

zu (b)

Schritt	1	2	3	4	5	6	7	8	9	10	11
Knoten	s	e_1	d_1	d_2	c_1	c_2	c_3	e_2	e_3	d_3	t
Pfadlänge	0	1	3	3	4	4	4	4	5	6	6
Vorgänger	-	s	s	e_1	s	d_1	d_2	e_1	d_2	d_2	e_3

Punkte:

Felder korrekt ausgefüllt	Punkte
≥ 2	0.5 P.
≥ 5	1 P.
≥ 7	1.5 P.
≥ 10	2 P.
≥ 12	2.5 P.
≥ 15	3 P.
≥ 17	3.5 P.
≥ 20	4 P.
≥ 22	4.5 P.
≥ 25	5 P.
≥ 27	5.5 P.
alle 30	6 P.

Notizen nach Korrektur: Falls der Graph falsch ausgefüllt wurde, kann hier immer noch volle Punktzahl erreicht werden, wenn der Dijkstra-Algorithmus richtig angewendet wurde. Falls Felder in zwei Spalten richtig, aber die Spalten vertauscht wurden, dann zählt das als 3 korrekte Felder (nicht 6). Falls anstelle der Pfadlänge (also Summe der Kantengewichte) die Anzahl der Kanten gezählt wurde, dann ist nur die Hälfte der Felder falsch.

Aufgabe 8: Hashing (9 Punkte)

(a) (4 Punkte) Ein*e Kommiliton*in hat die sechs Schlüssel $\{1, 2, 7, 8, 9, 15\}$ in eine anfangs leere Hashtabelle der Größe $M = 7$ eingefügt. Als Hashfunktion wurde die Divisions-Rest-Methode $h(k) = k \pmod 7$ verwendet. Bei Kollisionen kam ein Sondierungsverfahren zum Einsatz. Die finale Belegung der Hashtabelle sieht wie folgt aus:

Index	0	1	2	3	4	5	6
Schlüssel	7	8	2	1		15	9

1. Welche Sondierungsstrategie wurde verwendet: **Lineares Sondieren** mit der Sondierungsfunktion $s(n) = n$ oder eine Variante von **Quadratisches Sondieren** mit der Sondierungsfunktion $s(n) = n^2$? Begründen Sie Ihre Entscheidung nachvollziehbar in 1-2 Sätzen.
   Hinweis: Sie können diese Aufgabe beantworten ohne die Reihenfolge zu kennen, in der die Schlüssel eingefügt wurden.

2. Geben Sie eine **mögliche Reihenfolge** an, in der die sechs Schlüssel eingefügt worden sein könnten, um exakt die oben gezeigte Tabelle zu erzeugen.

1. **Quadratisches Sondieren.**
   Begründung: Lineares Sondieren kann ausgeschlossen werden. Der Schlüssel 15 hat die Heimatadresse $h(15) = 1$. Um bei linearem Sondieren auf Index 5 platziert zu werden, müssten die Indizes 1, 2, 3 und 4 zum Zeitpunkt des Einfügens belegt sein. Da Index 4 in der finalen Tabelle aber leer ist, ist dies ein Widerspruch.
2. **Mögliche Reihenfolgen:**
   - (2, 7, 8, 15, 1, 9)
   - (2, 8, 7, 15, 1, 9)
   - (2, 8, 15, 1, 7, 9)
   - (2, 8, 15, 1, 9, 7)
   - (2, 8, 15, 7, 1, 9)
   - (7, 2, 8, 15, 1, 9)
   - (7, 8, 2, 15, 1, 9)
   - (8, 2, 7, 15, 1, 9)
   - (8, 2, 15, 1, 7, 9)
   - (8, 2, 15, 1, 9, 7)
   - (8, 2, 15, 7, 1, 9)
   - (8, 7, 2, 15, 1, 9)

Punkte:

1 P für richtige Sondierungsstrategie
2 P für die richtige Begründung
1 P für die richtige Reihenfolge

(b) (3 Punkte) Beantworten Sie die folgenden Fragen kurz basierend auf dem Szenario in Teil (a). Begründen Sie Ihre Antworten nachvollziehbar in 1-2 Sätzen.

1. Geben Sie an, **welcher** eingefügte Schlüssel die höchste Anzahl an Sondierungen für eine erfolgreiche Suche benötigt und **wie viele** dies sind?

2. Unabhängig von Ihrer Antwort in Teil (a), erklären Sie, warum **lineares Sondieren** bei den gegebenen Schlüsseln zu primären Häufungen (primary clustering) geführt hätte.

1. Schlüssel **1**. Die Suche folgt derselben Sondierungssequenz wie beim Einfügen: $h(1) = 1$. Sonden bei Index 1 (falsch), 2 (falsch), 5 (falsch) und schließlich 3 (Treffer). Dies erfordert **4 Vergleiche**.
2. Die Schlüssel 8, 15, und 1 haben alle dieselbe Heimatadresse 1. Beim linearen Sondieren würden sie einen zusammenhängenden Block (Cluster) auf den Indizes 1, 2, 3 bilden, was die Suchzeit für nachfolgende Einfügungen und Suchen in diesem Bereich erhöht. Alternative Begründung: Da alle Eingaben auf 0, 1, und 2 mappen, würde lineares Sondieren zu Häufungen führen, da viele Schlüssel auf nur wenige, nebeneinanderliegende Adressen erteilt werden.

Punkte:

1.) 0.5 P für den richtigen Schlüssel
1.) 0.5 P für die richtige Anzahl an Sondierungen
1.) 0.5 P für die richtige Begründung
2.) 1.5 P für die richtige Begründung für primary clustering

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

Aussage	**Wahr**	**Falsch**
Das Konzept des universellen Hashings *garantiert*, dass die Verwendung einer zufällig gewählten Hashfunktion aus einer universellen Familie stets eine minimale Anzahl von Kollisionen erzielt, was es zu einer robusten Lösung gegen gezielte Denial-of-Service (DoS)-Angriffe macht.	☐	☒
Bei einer Hashtabelle mit Verkettung (Separate Chaining) kann die Suchzeit im *schlimmsten* Fall $\mathcal{O}(n)$ betragen.	☒	☐
Eine gute Hashfunktion sollte für ähnliche Schlüssel (z.B. "Wort1", "Wort2") möglichst *unterschiedliche* Hashwerte erzeugen (Avalanche-Effekt).	☒	☐
Beim quadratischen Sondieren kann es, anders als beim linearen Sondieren, vorkommen, dass *nicht* alle Plätze der Tabelle erreicht werden, auch wenn sie noch nicht voll ist.	☒	☐

Aussage	**Begründung**
Bei einer Hashtabelle mit Verkettung (Separate Chaining) kann die Suchzeit im schlimmsten Fall $\mathcal{O}(n)$ betragen, wobei n die Anzahl der Elemente in der Hashtabelle ist.	**Wahr**: Im schlimmsten Fall erzeugt die Hashfunktion für alle n Schlüssel denselben Hashwert. Dadurch landen alle Elemente im selben Bucket und bilden eine verkettete Liste der Länge n. Die Suche erfordert dann einen linearen Scan dieser Liste.
Eine gute Hashfunktion sollte für ähnliche Schlüssel (z.B. "Wort1", "Wort2") möglichst unterschiedliche Hashwerte erzeugen (Avalanche-Effekt).	**Wahr**: Dies ist die Definition des Lawineneffekts. Er sorgt für eine gute Streuung der Schlüssel, indem kleine Änderungen im Input zu großen, unkorrelierten Änderungen im Output führen. Dies minimiert Kollisionen von Mustern ähnelnden Schlüsseln.
Beim quadratischen Sondieren kann es, anders als beim linearen Sondieren, vorkommen, dass nicht alle Plätze der Tabelle erreicht werden, auch wenn sie noch nicht voll ist.	**Wahr**: Die Sondenfolge beim quadratischen Sondieren besucht nur eine Teilmenge der Tabellenindizes. Es ist möglich, dass diese Folge keine freien Plätze findet, obwohl die Tabelle insgesamt noch leer ist. Lineares Sondieren prüft hingegen garantiert jeden Platz.
Das Konzept des universellen Hashings garantiert, dass die Verwendung einer zufällig gewählten Hashfunktion aus einer universellen Familie stets eine minimale Anzahl von Kollisionen erzielt, was es zu einer robusten Lösung gegen gezielte Denial-of-Service (DoS)-Angriffe macht.	Falsch: Universelles Hashing minimiert die Kollisionen im Erwartungswert, aber gibt keine Garantie für eine minimale Kollisionsanzahl für eine einzelne, feste Hashfunktion. Der Schutz vor DoS-Angriffen ist zwar allg. korrekt, ändert aber nichts an der falschen Aussage bezüglich der Garantie.

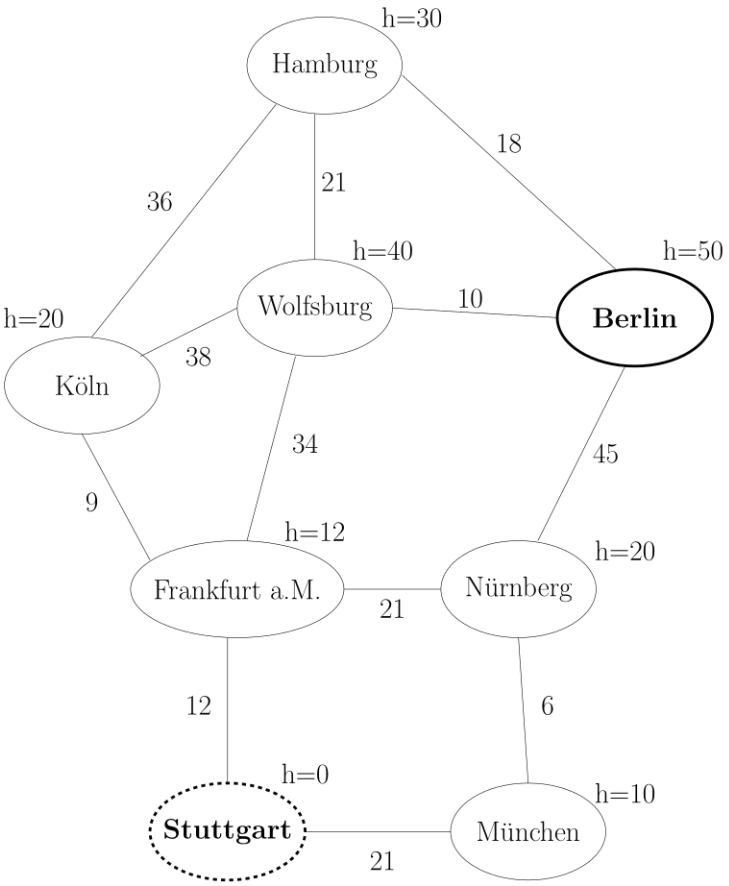
Punkte:
0.5 P für jede richtige Antwort

Aufgabe 9: A*-Algorithmus und Heuristiken (9 Punkte)

(a) (6 Punkte) Gegeben ist der folgende Graph, der deutsche Städte (Knoten) und Bahnverbindungen (Kanten) darstellt. Die Kantengewichte entsprechen der Reisezeit. Die im Graphen angegebenen h-Werte stellen die Schätzung der verbleibenden Reisezeit nach Stuttgart dar.

Ihre Aufgabe ist es, den A*-Algorithmus auszuführen, um den kürzesten Weg von **Berlin** nach **Stuttgart** zu finden. Die Knoten werden durch ihre Anfangsbuchstaben repräsentiert (z.B. B für Berlin, F für Frankfurt a.M.).

Protokollieren Sie die Ausführung des Algorithmus, indem Sie für jede Iteration die zwei Tabellen auf der nächsten Seite vervollständigen.



Hinweis zur Erinnerung:

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

Notieren Sie die Entwicklung der Priority-Queue (PQ) entlang der Iterationen des Algorithmus. Tragen Sie die Knoten in aufsteigender Reihenfolge, mit einem Komma getrennt, ein. Geben Sie außerdem den aktuellen f-Wert eines Knotens in der PQ in Klammern hinter dem Knoten an.

Iteration	PQ vor der Iteration
1	B(50)
2	H(48), W(50), N(65)
3	W(50), N(65), K(74)
4	F(56), N(65), K(68)
5	S(56), N(65), K(68)

Geben Sie in jeder Iteration den aktuell untersuchten Knoten an und tragen Sie zusätzlich für jeden Knoten jeweils den aktuellen g-Wert ein.

Iteration	Knoten	B	H	W	N	K	F	M	S
0	/	0	∞	∞	∞	∞	∞	∞	∞
1	B		18	10	45				
2	H					54			
3	W					48	44		
4	F								56
5	S								

Geben Sie abschließend die kürzeste Route an:

(B)erlin, (W)olfsburg, (F)rankfurt am Main, (S)tuttgart

Punkte:

0.5 P pro richtiger Zeile (x 4) in der PQ-Entwicklung der ersten Tabelle. Die Zeile darf keinen Fehler enthalten.
0.5P für korrekte Spalte “Knoten” in zweiter Tabelle.
0.5P für jede korrekte Zeile (x 5) in zweiter Tabelle.
1 P für richtige kürzeste Route (falls richtige kürzeste Route aus einer falschen Tabelle abgeleitet wurde, dann gibt es diesen Punkt)

(b) (3 Punkte) Für die folgenden drei Fragen verwenden wir die Evaluationsfunktion $f(n)$ für ein heuristisches Suchproblem, die wie folgt definiert ist:

$$f(n) = (w \cdot g(n)) + ((1 - w) \cdot h(n)).$$

Hier ist $h(n)$ eine *zulässige* Heuristik und der Gewichtungsfaktor w liegt im Bereich von $0.0 \leq w \leq 1.0$. Die Funktion $g(n)$ ist analog zum Aufgabenteil (a) definiert.

Welchen Suchalgorithmus erhalten Sie, wenn:

1.) $w = 0.0$

- ☐ Breitensuche (Breadth-First Search)
- ☒ Best-First Greedy-Algorithmus
- ☐ Dijkstras Algorithmus
- ☐ Tiefensuche (Depth-First Search)
- ☐ A* Algorithmus (A* Suche)
- ☐ Keine der oben genannten Antworten

Bei $w = 0.0$ wird $f(n) = h(n)$. Dies entspricht einer rein heuristikbasierten Suche (Greedy Best-First Search).

2.) $w = 0.5$

- ☐ Breitensuche (Breadth-First Search)
- ☐ Best-First Greedy-Algorithmus
- ☐ Dijkstras Algorithmus
- ☐ Tiefensuche (Depth-First Search)
- ☒ A* Algorithmus (A* Suche)
- ☐ Keine der oben genannten Antworten

Bei $w = 0.5$ wird $f(n) = 0.5 \cdot g(n) + 0.5 \cdot h(n)$. Dies ist die Standardform des A*-Algorithmus.

3.) $w = 1.0$

- ☐ Breitensuche (Breadth-First Search)
- ☐ Best-First Greedy-Algorithmus
- ☒ Dijkstras Algorithmus
- ☐ Tiefensuche (Depth-First Search)
- ☐ A* Algorithmus (A* Suche)
- ☐ Keine der oben genannten Antworten

Bei $w = 1.0$ wird $f(n) = g(n)$. Dies bedeutet, dass der Algorithmus Pfade basierend auf den tatsächlich angefallenen Kosten vom Startzustand priorisiert, was dem Dijkstras Algorithmus entspricht.

Punkte:

1 P für jede korrekte Antwort.