Aufgabe 1: Java-Programmierung (13 Punkte)

(a) (3 Punkte) Vervollständige die Klasse `Product`. Die Schnittstelle `Sellable` ist vorgegeben.

- `Product` soll die Schnittstelle `Sellable` implementieren. Das `protected`-Attribut `stock` soll die aktuelle Anzahl an Produkten im Lager speichern.
- Ergänze die Methodendefinition und die fehlenden Zeilen in `sell()`: Wenn noch mindestens ein Produkt im Lager ist, verkauft die Methode genau ein Stück. Die Methode `sell()` gibt `true` zurück, wenn der Verkauf geglückt ist, sonst `false`.

```
1  public interface Sellable {
2      public boolean sell();
3  }
4
5  public class Product _____________________ _____________________ {
6      _____________ _____________ _____________;
7
8      public Product(int stock) {
9          ______._____________ = _____________;
10     }
11
12     @Override
13     _____________ _____________ sell() {
14         if (_____________________________) {
15             _____________________________;
16             return true;
17         }
18         return false;
19     }
20 }
```

```
1  public interface Sellable {
2      boolean sell();
3  }
4
5  public class Product implements Sellable {
6      protected int stock;
7
```


```
8       public Product(int stock) {
9           this.stock = stock;
10      }
11
12      @Override
13      public boolean sell() {
14          if (this.stock > 0) {
15              this.stock--;
16              return true;
17          }
18          return false;
19      }
20  }
```

(b) (2.5 Punkte) Erstelle nun eine Klasse `PerishableProduct`, welche von `Product` erbt. Die Klasse soll zusätzlich das `private`-Attribut `bestBefore` (`String`) für das Ablaufdatum des Produkts enthalten.

```
1  public class PerishableProduct _____________ _____________ {
2      _____________ _____________ _____________;
3
4      public PerishableProduct(int stock, String bestBefore) {
5          _____________(_____________);
6          ______._____________ = _____________;
7      }
8  }
```

```
1  public class PerishableProduct extends Product {
2      private String bestBefore;
3
4      public PerishableProduct(int stock, String bestBefore) {
5          super(stock);
6          this.bestBefore = bestBefore;
7      }
8  }
```

(c) (2.5 Punkte) Erstelle nun eine **abstrakte** Basisklasse `Shop` mit dem `protected`-Attribut `products`. Das Attribut ist eine `LinkedList` von `Product`-Objekten. Die Liste soll anfangs leer sein. Mit der Funktion `addProduct` soll ein Produkt hinzugefügt werden können.

```
1  import java.util.LinkedList;
2
3  public _____________ class Shop {
4      _____________________________________________________________;
5
```


```
6       public void addProduct(Product product) {
7           _____________________________________________________________;
8       }
9   }
```

```
1   import java.util.LinkedList;
2
3   public abstract class Shop {
4       protected LinkedList<Product> products = new LinkedList<>();
5
6       public void addProduct(Product product) {
7           this.products.add(product);
8       }
9   }
```

(d) (3 Punkte) Die Klasse `StockCounter` befindet sich im selben Paket wie `Product`. Die Funktion `totalStock` erhält eine bereits initialisierte Liste `products`. Die Funktion soll zurückgeben, wie viele Produkte insgesamt im Lager sind, ohne die Liste zu verändern.

Der Code enthält 4 fehlerhafte Stellen. Finde sie und korrigiere sie direkt im Code.

```
1   import java.util.LinkedList;
2
3   public class StockCounter {
4       public int totalStock(LinkedList<Product> products) {
5           int sum = 0;
6           for (Product p : products) {
7               products.remove(p);
8               int stock = products.size();
9               sum++;
10          }
11          return;
12      }
13  }
```


Zeile	Korrektur
7	Fehler: Die Produkte sollen nicht gelöscht werden. Korrektur: Zeile `products.remove(p);` entfernen.
8	Fehler: Es wird die Anzahl der Produkte in der Liste statt der Anzahl des aktuellen Produkts im Lager verwendet. Korrektur: `int stock = p.stock;`.
9	Fehler: `sum++` erhöht die Summe nur um `1`. Korrektur: `sum += stock;`. Alternativ ist `sum += p.stock;` zulässig, wenn Zeile 8 passend angepasst oder gestrichen wird.
11	Fehler: Die berechnete Summe wird nicht zurückgegeben. Korrektur: `return sum;`.

(e) (2 Punkte) Wähle für jede Frage die passende Antwort aus.

Welche Komponente gibt den Speicher eines nicht mehr erreichbaren Objekts frei?

- ☐ Der Compiler
- ☒ Der Garbage Collector
- ☐ Der Konstruktor
- ☐ Der Quelltext

Was bedeutet `private` bei einem Attribut?

- ☒ Das Attribut ist außerhalb der deklarierenden Klasse nicht direkt zugreifbar.
- ☐ Das Attribut ist in allen Unterklassen direkt zugreifbar.
- ☐ Das Attribut ist im ganzen Programm direkt zugreifbar.
- ☐ Das Attribut ist nicht veränderbar.

Was gilt für `static`-Methoden?

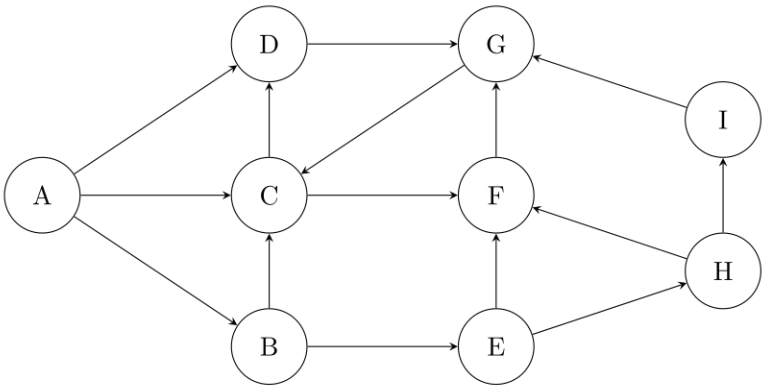
- ☐ Die Methode darf nur einmal aufgerufen werden.
- ☐ Die Methode muss ein Objekt verändern.
- ☒ Sie können aufgerufen werden, ohne vorher ein Objekt der Klasse erzeugen zu müssen.
- ☐ Die Methode startet automatisch beim Programmstart.

Wie werden Parameter in Java immer übergeben?

- ☐ Call by Name
- ☐ Call by Reference
- ☐ Call by Result
- ☒ Call by Value

Aufgabe 2: Tiefensuche (10 Punkte)

(a) (5 Punkte) Gegeben ist ein gerichteter und ungewichteter Graph. Führe die rekursive Tiefensuche (DFS) auf diesem Graphen durch. Beginne im Knoten A. Wenn es mehrere noch nicht besuchte Nachbarn gibt, wähle sie in alphabetischer Reihenfolge.



Pre-Order: Gib die Nebenreihenfolge an.
Post-Order: Gib die Hauptreihenfolge an.

Nebenreihenfolge: A, B, C, D, G, F, E, H, I
Hauptreihenfolge: G, D, F, C, I, H, E, B, A

(b) (2 Punkte) Ist auf dem gegebenen Graphen eine topologische Sortierung möglich? Begründe deine Entscheidung in 1–2 Sätzen.

Falls ja, gib eine mögliche topologische Sortierung an. Falls nein, gib eine Kante an, die entfernt werden kann, damit eine topologische Sortierung möglich wird.

Hier ist keine topologische Sortierung möglich, da der Graph einen Zyklus enthält, z. B. C, D, G, C oder C, F, G, C.
Die Kante G-C kann entfernt werden.

(c) (1 Punkt) Welche Kante kannst du aus dem gegebenen Graphen entfernen, damit der Knoten G vor dem Knoten D in der Nebenreihenfolge steht? Falls mehrere Kanten möglich sind, reicht es, eine Kante zu nennen.

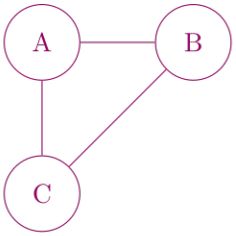
Kante C-D oder B-C

(d) (2 Punkte) Angenommen, du startest in einem beliebigen Knoten eines ungewichteten Graphen eine **Breitensuche** und erhältst den zugehörigen BFS-Baum. Kannst du aus diesem BFS-Baum immer einen kürzesten Pfad zwischen zwei beliebigen Knoten des Graphen ablesen?

Begründe deine Antwort kurz in 1–2 Sätzen oder gib ein Gegenbeispiel mit höchstens 5 Knoten an.

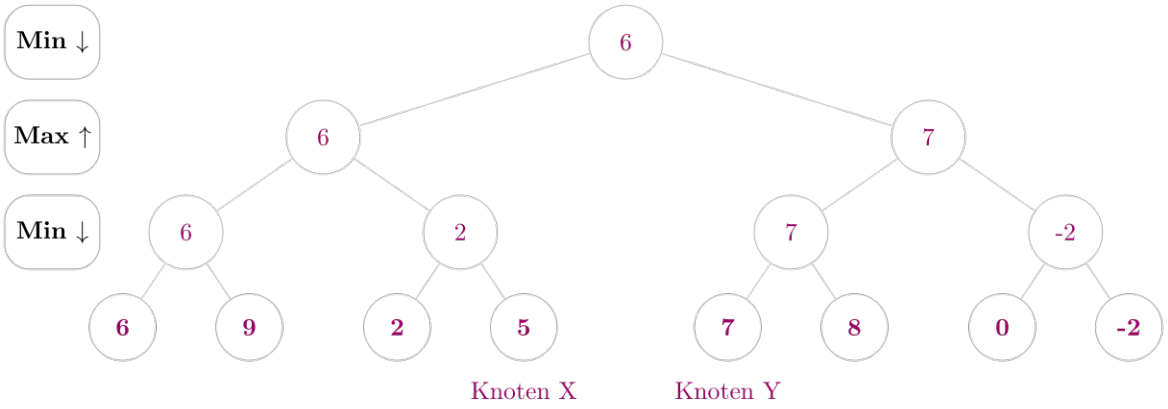
Hinweis: Ein BFS-Baum enthält für jeden erreichten Knoten die Kante, über die er bei der Breitensuche erstmals entdeckt wurde.

Nein. Eine Breitensuche liefert kürzeste Wege vom Startknoten zu allen erreichbaren Knoten, aber nicht notwendigerweise zwischen zwei anderen Knoten. Der BFS-Baum kann daher einen kürzesten Pfad zwischen zwei beliebigen Knoten nicht enthalten.
Gegenbeispiel (Startknoten A, kürzester Pfad zwischen B und C gesucht):



Aufgabe 3: Mini-Max und Alpha-Beta-Suche (8 Punkte)

(a) (1 Punkt) Führe auf dem Graphen den MiniMax-Algorithmus aus und trage die fehlenden Werte in die Knoten ein. Spieler Min minimiert den Wert, Spieler Max maximiert den Wert.



(b) (2 Punkte) Können Knoten X oder Knoten Y den Wurzelwert verändern, wenn an ihnen andere Werte stehen würden? Falls ja, welche Werte müssten an diesen Knoten stehen, damit sich der Wert an der Wurzel ändert? Falls nein, begründe deine Antwort kurz.

Knoten X

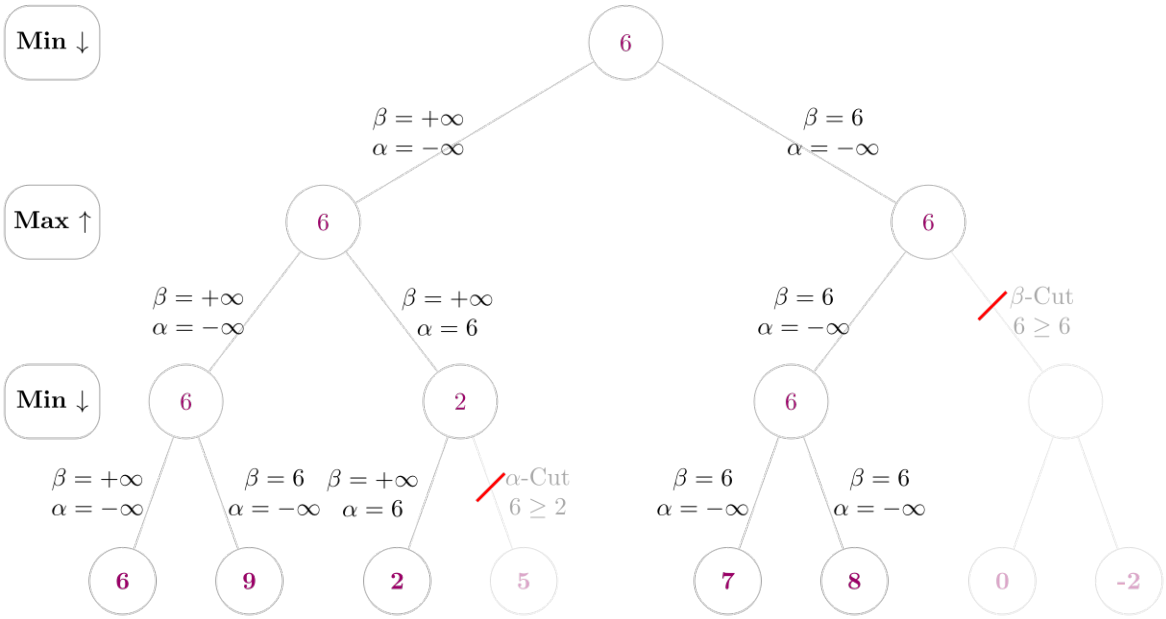
Knoten Y

Knoten X kann den Wurzelwert nicht verändern: Der Min-Elternknoten hat maximal den Wert 2, daher wird der darüberliegende Max-Knoten immer den linken Wert 6 wählen, egal was an Knoten X steht.

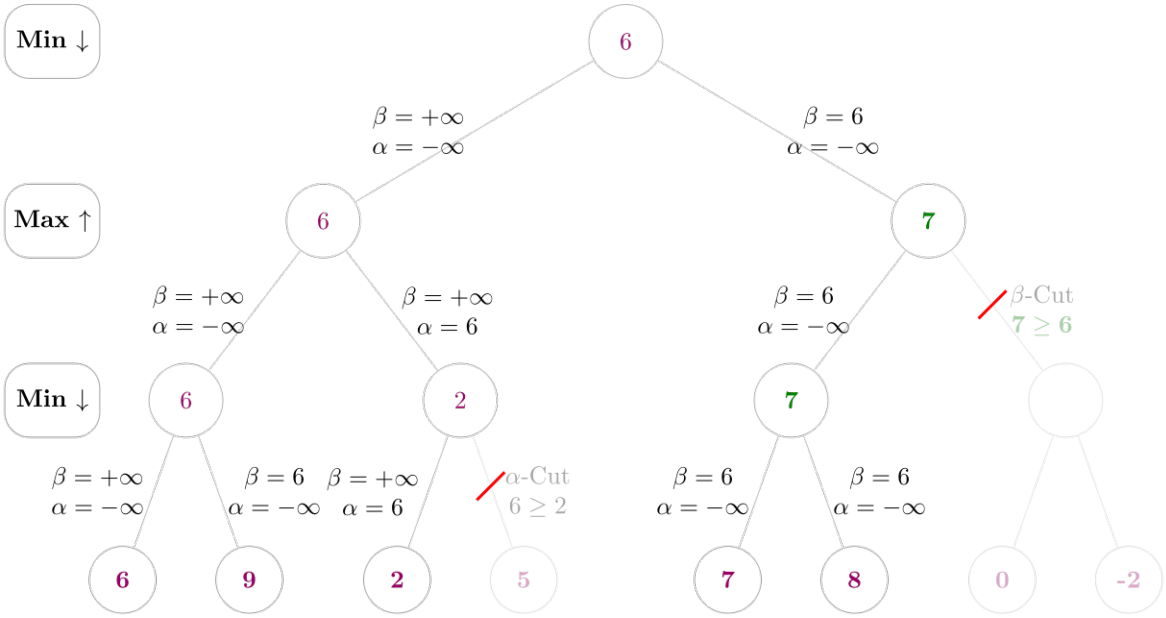
Knoten Y kann den Wurzelwert verändern. Dafür muss an Knoten Y ein Wert kleiner als 6 stehen.

(c) (5 Punkte) Führe auf der folgenden Kopie des Graphen die Alpha-Beta-Suche aus. Werte die Kindknoten immer von links nach rechts aus. Trage an jeder besuchten Kante die aktuellen Schranken beim Aufruf des Kindknotens ein. Trage außerdem die zurückgegebenen Werte in die besuchten inneren Knoten ein. Markiere alle Schnitte und benenne jeden Schnitt als α-Cut oder β-Cut.

Lösung nach der Konvention aus Vorlesung und Tutorium:

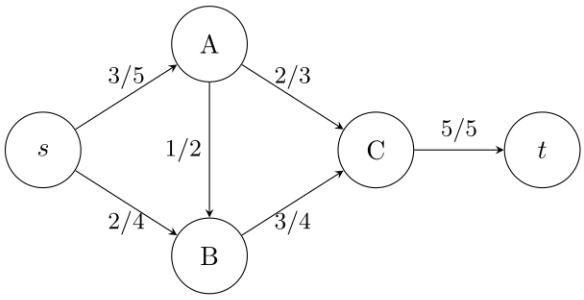


Ebenfalls akzeptierte Lösung nach einer anderen verbreiteten Implementierungskonvention, bei der besuchte innere Knoten ihre exakten Werte zurückgeben:



Aufgabe 4: Flussgraphen und Edmonds–Karp (6 Punkte)

(a) (1 Punkt) Der folgende Flussgraph mit Quelle s und Senke t ist gegeben. Jede Kante ist mit f/c beschriftet. Dabei ist f der Fluss auf der Kante und c die Kapazität der Kante.



Bestimme den Wert des Flusses.

Lösung:

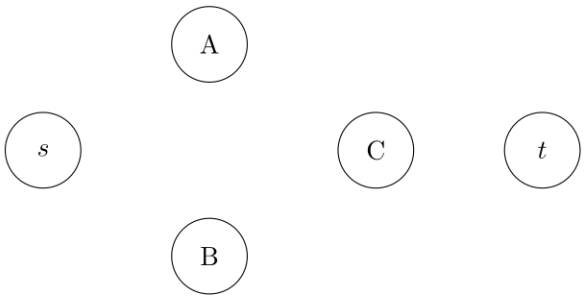
Der Wert des Flusses ist der gesamte aus s ausgehende Fluss:

$$f(s, A) + f(s, B) = 3 + 2 = 5.$$

Gleichwertig kann man den in t eingehenden Fluss verwenden:

$$f(C, t) = 5.$$

(b) (2 Punkte) Zeichne den Restflussgraph zum Flussgraphen aus Teilaufgabe (a). Die Knoten sind schon vorgegeben. Zeichne nur Restkanten mit positiver Restkapazität und beschrifte jede gezeichnete Restkante mit ihrer Restkapazität.

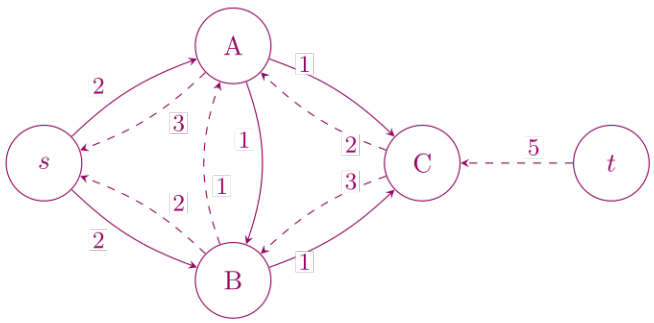


Lösung:

Für jede ursprüngliche Kante $u \to v$ gilt im Restflussgraphen:

$$c_f(u, v) = c(u, v) - f(u, v) \qquad \text{und} \qquad c_f(v, u) = f(u, v).$$

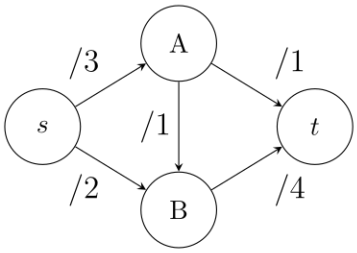
Restkanten mit Restkapazität 0 werden weggelassen.



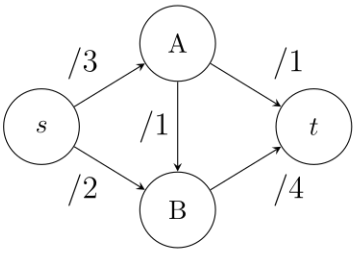
Durchgezogene Kanten zeigen hier Restkanten in Richtung einer ursprünglichen Kante. Gestrichelte Kanten sind Rückwärtskanten.

(c) (3 Punkte) Wende Edmonds–Karp auf den folgenden Flussgraphen an. Die Kanten wurden bereits mit Fluss 0 initialisiert. Fülle die drei Augmentierungen aus. Zeichne in jeder Kopie den jeweils gewählten augmentierenden Pfad deutlich ein. Falls es mehrere kürzeste augmentierende Pfade gibt, darf ein beliebiger davon gewählt werden. Trage links vom Schrägstrich die Flüsse ein, die nach dieser Augmentierung gelten.

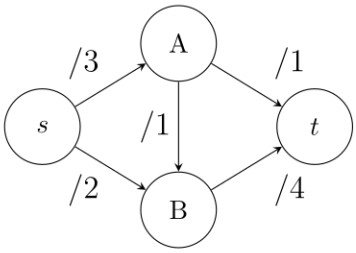
Iteration 1



Iteration 2



Iteration 3

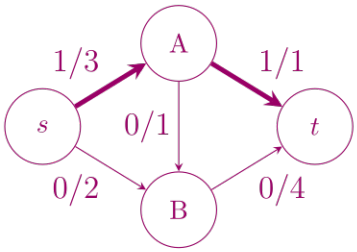


Lösung:

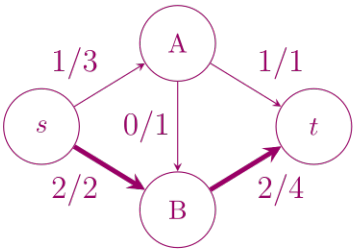
Es gibt zwei mögliche Lösungen, je nachdem, welchen der beiden kürzesten augmentierenden Pfade Edmonds–Karp zuerst wählt.

Falls zuerst $s \to A \to t$ gewählt wird:

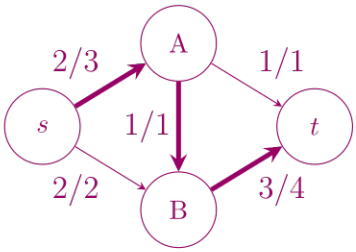
Iteration 1



Iteration 2

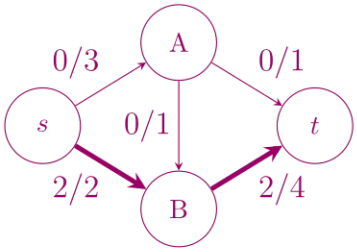


Iteration 3

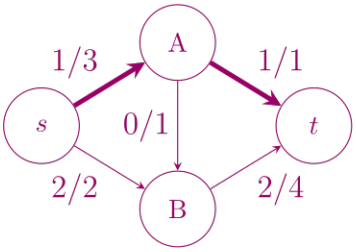


Falls zuerst $s \to B \to t$ gewählt wird:

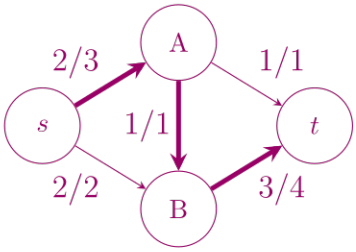
Iteration 1



Iteration 2



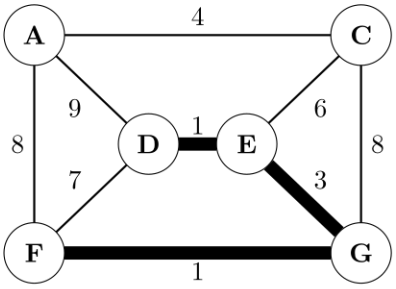
Iteration 3



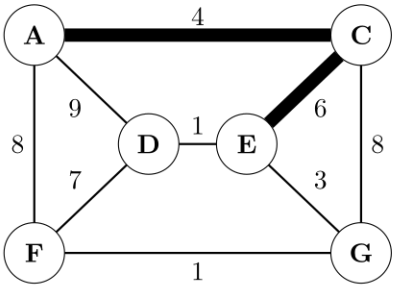
Aufgabe 5: Prim, Kruskal und Graphdarstellungen (10 Punkte)

(a) (4 Punkte) Entscheide für jeden Graphen, ob der markierte Zwischenstand von Prim, Kruskal, beiden oder keinem der Algorithmen stammen kann. Kreuze die richtige Antwort an.

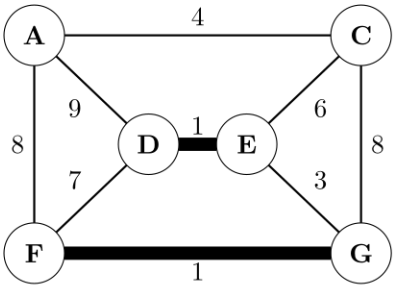
Hinweis: Prim kann hier von einem beliebigen Startknoten aus beginnen.



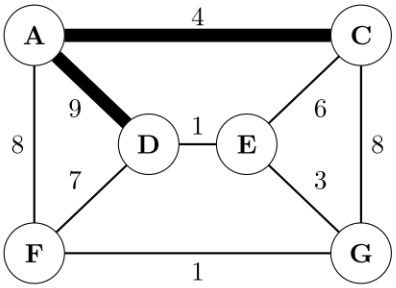
☒ beide ☐ nur Kruskal
☐ nur Prim ☐ keiner



☐ beide ☐ nur Kruskal
☒ nur Prim ☐ keiner



☐ beide ☒ nur Kruskal
☐ nur Prim ☐ keiner



☐ beide ☐ nur Kruskal
☐ nur Prim ☒ keiner

Oben links: beide Kruskal kann zuerst DE und FG mit Gewicht 1 und danach EG mit Gewicht 3 akzeptieren. Prim kann dieselben Kanten z.B. ab Startknoten D wählen.

Oben rechts: nur Prim Prim kann ab Startknoten A zuerst AC und dann CE wählen. Kruskal müsste vorher die leichteren Kanten DE, FG und EG betrachten.

Unten links: nur Kruskal Kruskal kann zuerst DE und FG akzeptieren. Prim kann keinen unzusammenhängenden Wald erzeugen.

Unten rechts: keiner Prim müsste nach AC als nächste Kante CE wählen, nicht AD. Kruskal würde AD mit Gewicht 9 erst nach mehreren leichteren Kanten betrachten.

(b) (2 Punkte) Der folgende Pseudocode soll mit Kruskals Algorithmus einen **maximalen** Spannbaum bestimmen und seine Kanten zurückgeben. Eine Kante wird als (u, v, w) beschrieben: u und v sind die beiden Knoten, w ist das Gewicht. Die Union-Find-Struktur speichert, welche Knoten bereits in derselben Zusammenhangskomponente liegen.

Ergänze Zeile 2 und Zeile 6.

```
1  function maxSpanningTree(n, edges)
2
3      uf ← UnionFind(n)
4      tree ← empty list
5      for each edge (u, v, w) in edges do
6
```

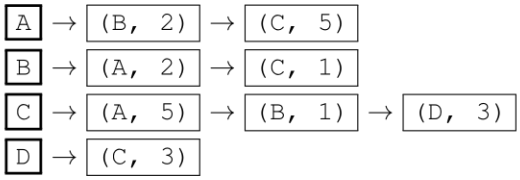
```
7               uf.union(u, v)   // verbindet die Komponenten von u und v
8               add (u, v, w) to tree
9       return tree
```

Vollständiger Pseudocode:

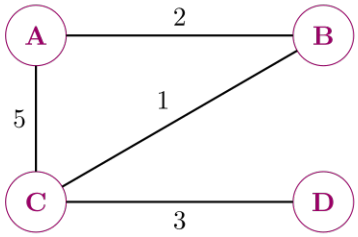
```
1   function maxSpanningTree(n, edges)
2       sort edges in descending order by weight
3       uf ← UnionFind(n)
4       tree ← empty list
5       for each edge (u, v, w) in edges do
6           if not uf.connected(u, v) then
7               uf.union(u, v)   // verbindet die Komponenten von u und v
8               add (u, v, w) to tree
9       return tree
```

- Zeile 2 sortiert die Kanten absteigend, damit zuerst die größten Kanten betrachtet werden.
- Zeile 6 prüft mit `uf.connected(u, v)`, ob `u` und `v` bereits im selben Baum liegen. In diesem Fall würde die aktuelle Kante einen Zyklus erzeugen.

(c) (2 Punkte) Für MST-Algorithmen werden Graphen oft als Adjazenzlisten gespeichert. Folgende Adjazenzlisten beschreiben einen ungerichteten, gewichteten Graphen.



Zeichne die in den Adjazenzlisten kodierten Kanten mit ihren Gewichten ein.



(d) (2 Punkte) Beantworte die folgenden Fragen. Entscheide jeweils, ob die Antwort beide Datenstrukturen, nur Adjazenzlisten, nur Adjazenzmatrizen oder keine der beiden Datenstrukturen ist.

Frage	**beide**	**nur Adjazenz-listen**	**nur Adjazenz-matrizen**	**keine**
Welche Datenstruktur kann gerichtete Graphen darstellen?	☒	☐	☐	☐
Welche Datenstruktur benötigt typischerweise Speicher proportional zu $\|V\| + \|E\|$?	☐	☒	☐	☐
Welche Datenstruktur ist bei sehr großen Graphen mit vergleichsweise wenigen Kanten typischerweise speichereffizienter?	☐	☒	☐	☐
Welche Datenstruktur erlaubt es, in konstanter Zeit zu prüfen, ob zwischen zwei gegebenen Knoten eine Kante existiert?	☐	☐	☒	☐

Frage 1: beide Beide Datenstrukturen können gerichtete und ungerichtete Graphen darstellen.

Frage 2: nur Adjazenzlisten Adjazenzlisten speichern zu jedem Knoten nur die vorhandenen Kanten und benötigen daher typischerweise $\Theta(|V| + |E|)$ Speicher.

Frage 3: nur Adjazenzlisten Bei sehr großen Graphen mit vergleichsweise wenigen Kanten enthält eine Adjazenzmatrix viele Einträge für nicht vorhandene Kanten. Diese Nullen müssen mitgespeichert werden, während Adjazenzlisten nur die vorhandenen Kanten speichern.

Frage 4: nur Adjazenzmatrizen In einer Adjazenzmatrix kann der Eintrag für ein Knotenpaar direkt gelesen werden. In einer Adjazenzliste muss man die Nachbarschaft eines Knotens durchsuchen.

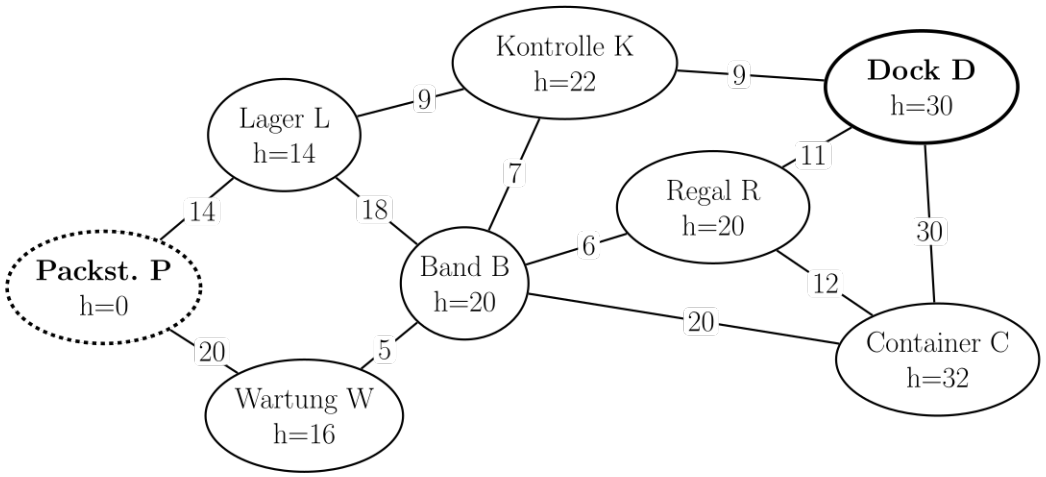
Aufgabe 6: Heuristische, approximative und randomisierte Algorithmen (10 Punkte)

(a) (8 Punkte) Gegeben ist der folgende Graph, der Wegpunkte in einem Lager darstellt. Die Kantengewichte geben die Fahrzeit eines Roboters in Sekunden an. Der Roboter darf entlang der Kanten in beide Richtungen fahren. Die angegebenen h-Werte schätzen die verbleibende Fahrzeit bis zur Packstation P.

Führe den A*-Algorithmus aus, um den schnellsten Weg vom Dock D zur Packstation P zu bestimmen.

Verwende dabei folgende Konventionen:

- Jeder Knoten kommt höchstens einmal in der Priority-Queue PQ vor. Wird sein g-Wert verbessert, wird sein Eintrag in der PQ aktualisiert.
- Bereits aus der PQ entnommene Knoten werden nicht erneut eingefügt.
- Die PQ wird nach aufsteigenden f-Werten sortiert. Bei gleichen f-Werten wird alphabetisch sortiert.
- Sobald P aus der PQ entnommen wird, endet der Algorithmus.



Notiere vor jeder Iteration den Inhalt der PQ in der Tabelle. Gib für jeden Knoten den aktuellen f-Wert in Klammern an.

Iteration	PQ vor der Iteration
1	D(30)
2	K(31), R(31), C(62)
3	R(31), L(32), B(36), C(62)
4	L(32), B(36), C(55)
5	P(32), B(36), C(55)

Trage anschließend für jede Iteration den entnommenen Knoten sowie alle in dieser Iteration erstmals gesetzten oder verbesserten g-Werte in die Tabelle ein. Lass alle anderen Felder leer.

Iteration	Knoten	D	C	R	K	B	L	W	P
0	/	0	∞	∞	∞	∞	∞	∞	∞
1	D		30	11	9				
2	K					16	18		
3	R		23						
4	L								32
5	P								

Gib abschließend die kürzeste Route an:

D, K, L, P

(b) (2 Punkte) Wähle für jede Frage genau eine passende Antwort aus.

Welche Aussage beschreibt eine typische Eigenschaft approximativer Algorithmen?

- ☐ Sie liefern immer eine optimale Lösung.
- ☒ Sie können eine Gütegarantie wie einen Faktor ρ besitzen.
- ☐ Sie dürfen keine Heuristik verwenden.
- ☐ Sie müssen immer randomisiert sein.

Was passiert bei A*, wenn für alle Knoten $h(n) = 0$ gilt?

- ☐ A* findet dann keine kürzesten Wege mehr.
- ☐ A* betrachtet nur den direkten Nachbarn des Startknotens.
- ☐ A* bleibt unverändert.
- ☒ A* verhält sich im Wesentlichen wie Dijkstra.

Welche Aussage über eine zulässige Heuristik bei A* ist richtig?

- ☒ Sie überschätzt die echten Restkosten zum Ziel nicht.
- ☐ Sie muss für jeden Knoten den echten Abstand zum Ziel angeben.
- ☐ Sie erlaubt negative Kantengewichte.
- ☐ Sie sortiert die Priority-Queue nur nach $h(n)$.

Welche Aussage unterscheidet Las-Vegas- und Monte-Carlo-Algorithmen korrekt?

- ☒ Las-Vegas-Algorithmen liefern stets ein korrektes Ergebnis, ihre Laufzeit kann jedoch variieren. Monte-Carlo-Algorithmen haben eine beschränkte Laufzeit, können aber mit einer gewissen Wahrscheinlichkeit ein falsches Ergebnis liefern.
- ☐ Las-Vegas-Algorithmen haben stets eine beschränkte Laufzeit, können aber falsche Ergebnisse liefern. Monte-Carlo-Algorithmen liefern stets korrekte Ergebnisse.
- ☐ Beide Algorithmustypen liefern stets korrekte Ergebnisse; sie unterscheiden sich nur in ihrer Laufzeit.
- ☐ Beide Algorithmustypen können falsche Ergebnisse liefern; nur ihre Fehlerwahrscheinlichkeiten unterscheiden sich.

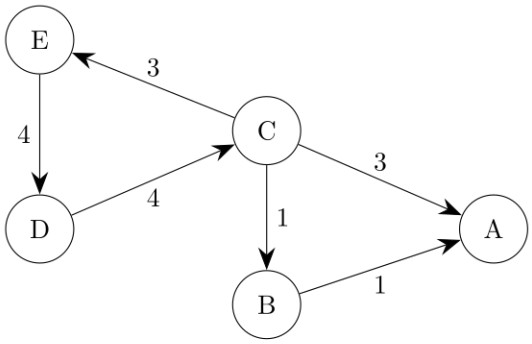
Frage	**Richtige Antwort**
1	Sie besitzen eine Gütegarantie in Form eines Approximationsfaktors ρ.
2	A* verhält sich im Wesentlichen wie Dijkstra.
3	Sie überschätzt die echten Restkosten zum Ziel nicht.
4	Las-Vegas-Algorithmen liefern stets eine optimale Lösung, wobei ihre Laufzeit variieren kann. Monte-Carlo-Algorithmen terminieren effizient, liefern aber nicht immer eine optimale Lösung.

Aufgabe 7: Bellman-Ford-Algorithmus (6 Punkte)

(a) (4 Punkte) Gegeben ist der folgende gewichtete Digraph. Führe den Bellman-Ford-Algorithmus **ohne** Warteschlange aus. Der Startknoten ist E.

Betrachte in jeder Iteration die Knoten in der Reihenfolge A, B, C, D, E. Betrachte die Nachbarn eines Knotens in lexikographischer Reihenfolge.

Übernimm jede Distanzverbesserung sofort. Sie darf bei später betrachteten Knoten derselben Iteration verwendet werden. Trage nach der Initialisierung und nach jeder vollständigen Iteration die aktuellen Distanzen in die Tabelle ein.



Iteration	A	B	C	D	E
Initial	∞	∞	∞	∞	0
1					
2					
3					
4					

Iteration	A	B	C	D	E
Initial	∞	∞	∞	∞	0
1	∞	∞	∞	4	0
2	∞	∞	8	4	0
3	11	9	8	4	0
4	10	9	8	4	0

(b) (2 Punkte) Wir fügen dem gegebenen Graphen eine gerichtete Kante von B nach D mit ganzzahligem Gewicht w hinzu. Dadurch entsteht der Zyklus $D \to C \to B \to D$.

Wir betrachten Kantenfolgen, in denen Knoten und Kanten mehrfach vorkommen dürfen.

Hinweis: Als kürzeste Kantenfolge bezeichnen wir im Folgenden eine Kantenfolge mit minimalem Gesamtgewicht.

1. Bestimme das Gewicht des Zyklus in Abhängigkeit von w.
2. Begründe kurz, warum es für $w = -5$ von E aus **keine** eindeutige kürzeste Kantenfolge gibt.
3. Bestimme die kleinste ganze Zahl w, für die von E aus zu allen Knoten eine eindeutige kürzeste Kantenfolge existiert.

Das Gewicht des Zyklus ist $4 + 1 + w = w + 5$.

Für $w = -5$ hat der Zyklus Gewicht 0. Beispielsweise haben die Kantenfolgen $E \to D$ und $E \to D \to C \to B \to D$ beide Gewicht 4. Der Zyklus kann beliebig oft durchlaufen werden, ohne die Kosten zu verändern. Daher sind die kürzesten Kantenfolgen nicht eindeutig.

Für $w = -4$ hat der Zyklus positives Gewicht. Die Kante $E \to D$ hat Gewicht 4, die Kantenfolge $E \to D \to C \to B \to D$ dagegen Gewicht 5; dadurch entsteht keine weitere kürzeste Kantenfolge. Für $w < -5$ ist der Zyklus negativ. Die kleinste zulässige ganze Zahl ist daher $w = -4$.

Aufgabe 8: Hashing (6 Punkte)

Gegeben sind zwei Hashtabellen. Beide Tabellen enthalten bereits die Schlüssel $3, 11, 19, 6$. Es wird die folgende Hashfunktion verwendet:

$$h(k) = k \bmod 8.$$

In der Tabelle **Verkettung** werden Kollisionen durch Verkettung aufgelöst. Die Schlüssel einer Kette stehen in Einfügereihenfolge von links nach rechts. In der Tabelle **Lineares Sondieren** wird bei einer Kollision zyklisch das jeweils nächste Tabellenfeld untersucht.

Index	0	1	2	3	4	5	6	7
Verkettung				3, 11, 19			6	

Index	0	1	2	3	4	5	6	7
Lineares Sondieren				3	11	19	6	

(a) (2 Punkte) Berechne $h(27)$ und füge den Schlüssel 27 in beide Tabellen ein. Trage die neuen Inhalte direkt in die Tabellen oben ein.

$h(27) =$ _____

Es gilt $h(27) = 27 \bmod 8 = 3$.

Bei Verkettung wird 27 am Ende der Kette an Index 3 eingefügt. Die Kette lautet danach $3, 11, 19, 27$.

Beim linearen Sondieren sind die Indizes $3, 4, 5$ und 6 bereits belegt. Daher wird 27 an Index 7 gespeichert.

(b) (2 Punkte) Ausgehend von den oben gezeigten Ausgangstabellen wird der Schlüssel 19 gesucht. Gib für beide Tabellen die Schlüssel an, die dabei der Reihe nach verglichen werden.

Verkettung:

Lineares Sondieren:

Die Suche nach 19 vergleicht in beiden Tabellen der Reihe nach die Schlüssel $3, 11, 19$.

(c) (2 Punkte) Beantworte jede der folgenden Fragen, indem du genau eine passende Antwort ankreuzt.

Wann ist eine Hashtabelle besonders gut geeignet?

- ☒ Wenn aus einem großen Schlüsseluniversum nur wenige Schlüssel gespeichert und häufig exakt gesucht werden.
- ☐ Wenn die Schlüssel stets sortiert ausgegeben werden müssen.
- ☐ Wenn häufig Schlüssel aus einem Wertebereich gesucht werden.
- ☐ Wenn jede Suche garantiert $\mathcal{O}(1)$ dauern muss.

Welche Laufzeit kann die Suche in einer Hashtabelle mit n gespeicherten Schlüsseln im schlechtesten Fall haben?

- ☐ $\mathcal{O}(1)$
- ☐ $\mathcal{O}(\log n)$
- ☒ $\mathcal{O}(n)$
- ☐ $\mathcal{O}(n \log n)$

Unter welcher Bedingung ist die erwartete Suchzeit in einer Hashtabelle $\mathcal{O}(1)$?

- ☐ Die Schlüssel werden gleichmäßig verteilt, der Füllgrad darf aber wachsen.
- ☐ Der Füllgrad bleibt konstant; alle Schlüssel liegen im selben Feld.
- ☐ Die Hashfunktion wird für jeden Schlüssel nur einmal ausgewertet.
- ☒ Die Schlüssel werden gleichmäßig verteilt und der Füllgrad bleibt konstant.

Welche Art von Hashfunktion wird für eine im Voraus bekannte statische Schlüsselmenge so konstruiert, dass keine Kollisionen auftreten?

- ☐ universelle Hashfunktion
- ☒ perfekte Hashfunktion
- ☐ kryptographische Hashfunktion
- ☐ probabilistische Hashfunktion

- Häufige exakte Suchen in einer großen Schlüsselmenge
- $\mathcal{O}(n)$
- Gleichmäßige Verteilung bei konstantem Füllgrad
- Perfekte Hashfunktion

Aufgabe 9: Dynamische Programmierung (10 Punkte)

Ein Würfel mit 6 Seiten und den Augenzahlen $1, 2, 3, 4, 5$ und 6 wird mehrfach geworfen. Die Augenzahlen werden aufsummiert. Mithilfe dynamischer Programmierung soll für eine gegebene Summe N ermittelt werden, auf wie viele Arten diese Summe geworfen werden kann.

Beispiel: Für $N = 3$ gibt es 4 mögliche Wurffolgen: (3), $(1, 2)$, $(2, 1)$ und $(1, 1, 1)$. In folgender Grafik siehst du die Gruppierung der Wurffolgen nach dem ersten Wurf.

Erster Wurf		**Restsumme**	**Wurffolge inkl. erstem Wurf**
$\boxed{\mathbf{1}}$	$\Rightarrow$	$\boxed{N = 2}$	$(\mathbf{1}, 2)$, $(\mathbf{1}, 1, 1)$
$\boxed{\mathbf{2}}$	$\Rightarrow$	$\boxed{N = 1}$	$(\mathbf{2}, 1)$
$\boxed{\mathbf{3}}$	$\Rightarrow$	$\boxed{N = 0}$	$(\mathbf{3})$

Ziel dieser Aufgabe ist es, eine rekursive Funktion Anz zu bestimmen, mit der das Problem mithilfe dynamischer Programmierung gelöst werden kann.

(a) (1.5 Punkte) Gib alle möglichen Wurffolgen für $N = 4$ an.

Die möglichen Wurffolgen sind (4), $(1, 3)$, $(3, 1)$, $(2, 2)$, $(1, 1, 2)$, $(1, 2, 1)$, $(2, 1, 1)$ und $(1, 1, 1, 1)$.

(b) (4 Punkte) Definiere eine rekursive Funktion $\textsc{Anz}(n)$, die die Anzahl möglicher Wurffolgen mit Summe n berechnet. Verwende eine mathematische Schreibweise und keinen Java- oder Pseudocode.

Hinweis: Gruppiere die Wurffolgen wie in der Abbildung für $N = 3$ danach, welche Augenzahl zuerst geworfen wird. Überlege, welche Restsumme danach noch geworfen werden muss.

Hier exemplarisch drei korrekte Lösungen:

$$\textsc{Anz}(n) = \begin{cases} 0 & \text{falls } n < 0 \\ 1 & \text{falls } n = 0 \\ \textsc{Anz}(n-1) + \textsc{Anz}(n-2) + \textsc{Anz}(n-3) + \textsc{Anz}(n-4) + \textsc{Anz}(n-5) + \textsc{Anz}(n-6) & \text{falls } n > 0 \end{cases}$$

$$\textsc{Anz}(n) = \begin{cases} 0 & \text{falls } n < 0 \\ 1 & \text{falls } n = 0 \\ \sum_{i=1}^{6} \textsc{Anz}(n-i) & \text{falls } n > 0 \end{cases}$$

$$\textsc{Anz}(n) = \begin{cases} 1 & \text{falls } n = 0 \\ \sum_{i=1}^{\min(6,n)} \textsc{Anz}(n-i) & \text{falls } n > 0 \end{cases}$$

Hinweis: Der Basisfall ($\textsc{Anz}(0) = 1$) bedeutet, dass es genau eine Möglichkeit gibt, eine Wurffolge bei Restsumme 0 zu vervollständigen: keine weiteren Würfe zu machen. $\textsc{Anz}(n) = 0$ für $n < 0$, da es keine Möglichkeit gibt, eine negative Summe zu erreichen.

(c) (1.5 Punkte) Gib die Größenordnung der Laufzeit in Abhängigkeit von N an, wenn die ANZ-Werte mithilfe einer dynamischen Programmierungstabelle (DP-Tabelle) berechnet werden. Begründe deine Antwort kurz in 1–2 Sätzen.

Laufzeit des Dynamic-Programming-Algorithmus: Es müssen alle N Einträge der Tabelle berechnet werden. Für jeden Eintrag ist eine konstante Anzahl von Rechenoperationen nötig (ein Vergleich und 6 Additionen). Die Laufzeit beträgt daher insgesamt $\mathcal{O}(N)$.

(d) (1.5 Punkte) Gib eine sinnvolle obere Schranke in Abhängigkeit von N für die Laufzeit einer rekursiven Lösung ohne Memoisierung an. Begründe deine Antwort kurz in 1 bis 2 Sätzen.

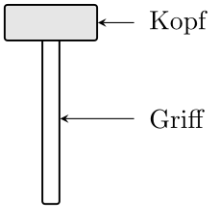
Laufzeit ohne Dynamic-Programming: Der Rekursionsbaum hat Tiefe höchstens N. Pro Aufruf werden bis zu 6 rekursive Aufrufe gestartet. Die Laufzeit beträgt daher insgesamt $\mathcal{O}(6^N)$.

(e) (1.5 Punkte) Wie verändert sich die Laufzeit mit DP-Tabelle, wenn nun ein M-seitiger Würfel verwendet wird? Dabei ist M ein neuer Eingabeparameter des Problems und es gilt stets $M \leq N$. Begründe deine Antwort kurz in 1–2 Sätzen.

Der Algorithmus muss nun pro Zelle M verschiedene Zahlen aufsummieren. **Die Laufzeit beträgt daher insgesamt $\mathcal{O}(M \cdot N)$.**

Alternative Lösung: Mit Matrixpotenzierung lässt sich $\text{ANZ}(N)$ in $O(\log N)$ arithmetischen Operationen berechnen. Dies ist jedoch ein anderer Algorithmus als die geforderte Rekursion.

Aufgabe 10: Greedy Algorithmen (11 Punkte)



Für einen Hammer brauchst du genau einen Griff und genau einen Kopf. Es gibt Maschinen, die Griffe produzieren, und Maschinen, die Köpfe produzieren. Jede Maschine M wird durch ein Tupel aus zwei Werten beschrieben:

$$M = (t_{\text{start}}, d)$$

- t_{start}: der Zeitpunkt, ab dem die Maschine verfügbar ist,
- d: die Dauer deines Auftrags auf dieser Maschine.

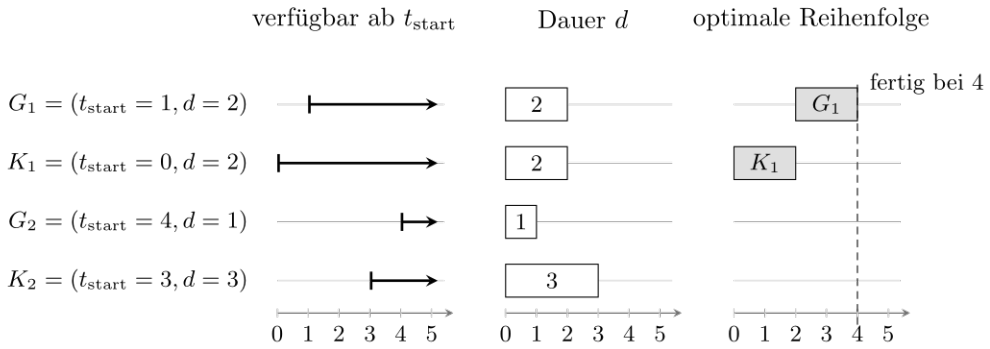
Wenn du eine Maschine zum Zeitpunkt $t \geq t_{\text{start}}$ startest, ist dein Auftrag zum Zeitpunkt $t + d$ fertig. Du kannst immer nur eine Maschine gleichzeitig benutzen. Danach kannst du den nächsten Auftrag starten.

Seien

$G_1, \ldots, G_n$ die Maschinen für Griffe,
$K_1, \ldots, K_m$ die Maschinen für Köpfe.

Gesucht ist der frühestmögliche Zeitpunkt, zu dem der Hammer fertig ist.

Beispiel:



Hier ist es optimal, zuerst K_1 und danach G_1 zu benutzen. Die Lösung ist also 4.

Alle Teilaufgaben sind separat voneinander lösbar.

(a) (4 Punkte) Jemand schlägt folgenden Algorithmus vor:

> Wähle zuerst die Maschine mit der frühesten Endzeit $t_{\text{start}} + d$. Wähle danach für das andere Teil die Maschine, die nach diesem Zeitpunkt am frühesten fertig wird.

Gib ein Gegenbeispiel an, das zeigt, dass dieser Algorithmus nicht immer optimal ist.

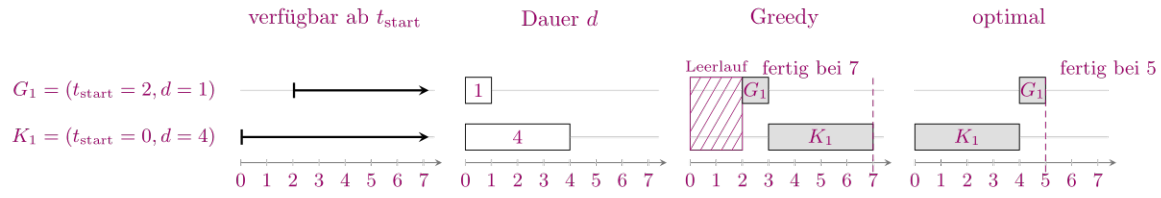
Hinweis: Es gibt ein Gegenbeispiel mit je einer Maschine für Griffe und Köpfe.

$$G_1 = (t_{\text{start}} = 2, d = 1), \qquad K_1 = (t_{\text{start}} = 0, d = 4).$$

Die Struktur des Gegenbeispiels ist: Die eine Maschine ist spät verfügbar und dauert nur sehr kurz, während die andere Maschine früh verfügbar ist und sehr lange dauert. Der vorgeschlagene Algorithmus wählt zuerst die Maschine für den Griff (Endzeit 3) und danach die Maschine für den Kopf. Dadurch entsteht Leerlauf bis zum Start von G_1.

Die Gesamtzeit der Greedy-Auswahl $G_1 \to K_1$ ist 7.

Optimal ist jedoch die Auswahl $K_1 \to G_1$ mit Gesamtzeit 5.



Allgemein scheitert der Greedy-Ansatz genau dann, wenn die Maschine mit der kleineren Endzeit später verfügbar wird als die andere Maschine. Äquivalent: Für zwei Maschinen M_1 und M_2 ist die Greedy-Wahl falsch, wenn $t_{\text{start},1} + d_1 < t_{\text{start},2} + d_2$ und gleichzeitig $t_{\text{start},1} > t_{\text{start},2}$.

(b) (4 Punkte) Du bist nun auf der Suche nach dem optimalen Algorithmus für dieses Problem. Betrachte zunächst den Spezialfall, dass immer zuerst ein Kopf und danach ein Griff produziert werden muss. Beschreibe einen optimalen Greedy-Algorithmus für diesen Spezialfall und gib seine Laufzeit in Abhängigkeit von n und m an. Begründe kurz die Laufzeit.

- Wähle die Kopfmaschine K_i, die am frühesten fertig ist ($t_{\text{start},i} + d_i$ minimal). Merke dir ihre Fertigstellungszeit als t.
- Prüfe jede Griffmaschine G_j und berechne, wann sie nach t fertig wäre: $\max(t, t_{\text{start},j}) + d_j$. Nimm die Griffmaschine mit der kleinsten solchen Fertigstellungszeit.
- Laufzeit: $O(n + m)$, weil jede Maschine höchstens einmal geprüft wird.

(c) (3 Punkte) Entwirf nun einen optimalen Algorithmus mit Laufzeit $O(n + m)$, der das ursprüngliche Problem löst, bei dem die Maschinenreihenfolge frei gewählt werden darf. Begründe, warum dein Algorithmus die Laufzeit $O(n + m)$ hat.

Hinweis: Du kannst den Algorithmus aus Teilaufgabe b) als gegeben voraussetzen und hier wiederverwenden.

- Berechne die beste Lösung für: zuerst Kopf, dann Griff, wie in Teilaufgabe b).
- Berechne analog die beste Lösung für: zuerst Griff, dann Kopf.
- Gib das Minimum der beiden Endzeiten zurück.
- Laufzeit: $O(n + m)$ pro Reihenfolge, also insgesamt $O(2 \cdot (n + m)) = O(n + m)$.