TECHNISCHE UNIVERSITÄT BERLIN
Fakultät IV – Elektrotechnik und Informatik
Fachgebiet Neurotechnologie (MAR 4-3)
Prof. Dr. Benjamin Blankertz
Röhr / Schlegel

Algorithmen und Datenstrukturen, SoSe 24

Bitte füllen Sie alle folgenden Felder aus:

Vorname: ______________________

Nachname: ______________________

TUB-Kontoname: ______________________

Matrikelnummer: ______________________

Studiengang: ______________________

Hochschule: ______________________

Durch meine Unterschrift bestätige ich die Korrektheit obiger Angaben sowie meine Prüfungsfähigkeit und die Anmeldung zur Prüfung!

______________________ ______________________
Ort, Datum Unterschrift

Beachten Sie die folgenden Hinweise!

- Sie brauchen Ihren Namen **nur** auf das Deckblatt zu schreiben. Die restlichen Blätter können über die Klausur-ID zugeordnet werden.
- Diese Klausur besteht mit diesem Deckblatt aus den (nummerierten) Seiten **1-15**.
- Am Ende der Klausur befinden sich zwei leere Seiten, die Sie für Notizen verwenden können. Sollten Sie mehr Papier benötigen, können Sie dies von der Aufsicht bekommen. Notieren Sie in diesem Fall die Klausur-ID auf dem Zusatzblatt.
- Notieren Sie Ihre Antworten nur auf dem Blatt (inklusive Rückseite), auf dem die zugehörige Aufgabe steht, da die Aufgaben getrennt korrigiert werden.
- Falls Sie eine Antwort auf ein Zusatzblatt schreiben, markieren Sie dies klar bei der zugehörigen Aufgabe und auf dem Zusatzblatt.
- Geben Sie nur eine Lösung pro Aufgabe ab, streichen Sie alle alternativen Lösungsansätze auf Schmier-/Notizblättern durch.
- Schreiben Sie **nicht** mit roter Farbe, grüner Farbe (Korrekturfarben) oder Bleistift. Diese Lösungen werden nicht bewertet!
- Insgesamt können in der Klausur **100 Punkte** erreicht werden.

Zusatzblätter:

Punktetabelle

Aufgabe	Punkte		
1	/8		
2	/11		
3	/13		
4	/13		
5	/15		
6	/12		
7	/12		
8	/7		
9	/9		
$\sum$	/ 100		

Aufgabe 1: Grundlagen in Java (8 Punkte)

Die Klasse `Supermarket` soll eine sehr abstrakte, virtuelle Supermarkt-Kasse darstellen, wobei die Produkte als Strings gespeichert werden. Ihr Coding Assistent hat Ihnen zwei Methoden vorgeschlagen. Mit `emptyBag()` soll der Einkaufsbeutel auf dem Kassenband `conveyorBelt` geleert werden und ein Warentrenner hingelegt werden. Mit `completePurchase()` wird ein Einkauf bis zum Warentrenner dann wieder ohne den Trenner in den Beutel gepackt. Sie können davon ausgehen, dass alle Funktionsaufrufe innerhalb der Methoden korrekt sind, funktionieren und eine konstante Laufzeit haben.

```
1  import java.util.LinkedList;
2  import java.util.Queue;
3  import java.util.Stack;
4
5  class Supermarket {
6      Queue<String> conveyorBelt = new LinkedList<>();
7
8      // to evaluate:
9      void emptyBag(Stack<String> shoppingBag) {
10         int size = shoppingBag.size();
11         for (int i = 0; i < size / size; i++) {
12             conveyorBelt.add(shoppingBag.pop());
13         }
14         conveyorBelt.add("Divider");
15     }
16
17     // to evaluate:
18     void completePurchase(Stack<String> shoppingBag) {
19         while (!conveyorBelt.isEmpty()) {
20             String item = conveyorBelt.poll();
21             if (item.equals("Divider")) {
22                 continue;
23             }
24             shoppingBag.push(item);
25             System.out.println("Item " + item + " paid.");
26         }
27     }
28 }
```

Geben Sie die Größenordnung der Laufzeit von den Methoden `emptyBag` und `completePurchase` an, so wie sie hier (ggf. fehlerhaft) implementiert sind, mit den richtigen abhängigen Größen. Geben Sie außerdem an, ob die jeweilige Methode funktioniert. **Begründen Sie kurz**, falls eine Methode fehlerhaft ist.

Aufgabe 2: Java-Programmierung ($4 + 5 + 2 = 11$ Punkte)

Sie haben eine Klasse `WeightedDigraph`, welche einen gerichteten Graphen mit gewichteten Kanten darstellen soll. Kanten werden als `DirectedEdges` dargestellt, wobei jeder Knoten einen `Bag` an **ausgehenden** Kanten zugewiesen bekommt.

```
public class WeightedDigraph {
    private final int vertices;
    private int edgesTotal;
    private Bag<DirectedEdge>[] edges;

    public WeightedDigraph(int V) {
        vertices = V;
        edgesTotal = 0;
        //TODO: a)
    }


    public void addEdge(DirectedEdge e) {
        //TODO: b)
    }

}
```

(a) Vervollständigen Sie den Konstruktor, der die Anzahl der Knoten übergeben bekommt und den `Bag`-Array sowohl initialisiert als auch mit leeren Bags befüllt. Die Anzahl an Bags entspricht der Anzahl an Knoten.

```
public WeightedDigraph(int V) {
    vertices = V;
    edgesTotal = 0;







}
```

(b) Schreiben Sie nun die Methode, die dem korrekten `Bag` im Array die übergebene Kante `e` hinzufügt. `DirectedEdge` hat die Methoden `from()` und `to()`, welche den jeweiligen Knoten als Integer zurückgibt. `Bag` hat die Methode `add()`, die Objekte akzeptiert, welche bei der Erstellung des Bags als Typ gesetzt wurden. Vergessen Sie außerdem nicht, den Counter für insgesamte Kanten hochzuzählen.

```
public void addEdge(DirectedEdge e) {







}
```

(c) Welchen Vorteil hat es, die Bag Klasse mit Generics zu definieren, so dass bei der Definition von `edges` der Element-Typ extra übergeben werden muss? Antworten Sie in maximal 2 Sätzen.

Aufgabe 3: Backtracking $(6 + 4 + 3 = 13$ Punkte)

Zur Lösung des **teilbaren** Rucksackproblems kann ein einfacher backtracking Ansatz genutzt werden. Das teilbare Rucksackproblem wurde in der Vorlesung so eingeführt: Es sind n Objekte mit Gewichten w_i und Werten v_i (für $1 \leq i \leq n$), sowie ein Rucksack (*knapsack*) mit einer maximalen Kapazität W gegeben. Wähle Objekte **oder Teile der Objekte**, sodass ihr Gesamtwert maximal ist, aber ihr Gesamtgewicht die Kapazität nicht überschreitet.
Dafür sind Ihnen zwei Klassen gegeben. Die erste Klasse `Item` ist fehlerfrei und repräsentiert die Objekte, die in den Rucksack gelegt werden.

```
1  public class Item {
2      double weight, value;
3      String name;
4      public Item(String name, double weight, double value) {
5          this.name = name;
6          this.weight = weight;
7          this.value = value;    }
8  }
```

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

```
1  class FractKnapsackBacktracking {
2     private double maxValue = 0;
3     private List<Item> bestItems = new ArrayList<>();
4     private List<Double> bestFractions = new ArrayList<>();
5     // Current state variables
6     private List<Item> currentItems = new ArrayList<>();
7     private List<Double> currentFractions = new ArrayList<>();
8     // Constructor
9     public FractKnapsackBacktracking(List<Item> items, double capacity) {
10       for (int i = 0; i < items.size(); i++) {
11          backtracking(items, 0, capacity, 0);
12          items.add(items.remove(0));
13       }
14    }
15    private void backtracking(List<Item> items, int index, double
         remainingCapacity, double currentValue) {
16       //Case 0: Done
17       if (index == items.size() || remainingCapacity <= 0) {
18          if (currentValue > maxValue) {
19             maxValue = currentValue;
20             bestItems = new ArrayList<>(currentItems);
21             bestFractions = new ArrayList<>(currentFractions);
22          }
23          return;
24       }
25       Item currentItem = items.get(index);
26       // Case 1: Skip Item
27       backtracking(items, index + 1, remainingCapacity, currentItem.value);
28       // Case 2: Do not skip Item
29       currentItems.add(currentItem);
30       double fraction = Math.min(1, remainingCapacity / currentItem.weight);
31       currentFractions.add(fraction);
32       double newValue = currentValue + (currentItem.value * fraction);
33       backtracking(items, index + 1, remainingCapacity, newValue);
34    }
35 }
```

(a) Korrigieren Sie die Zeilen 26-34 der Klasse `FractionalKnapsackBacktracking`, sodass der Backtracking Ansatz die richtigen Ergebnisse liefert.
Sie brauchen dabei nur die Zeilen aufzuschreiben, die berichtigt oder hinzugefügt werden müssen. Geben Sie dabei die Zeilennummern an. Wenn Sie zur Berichtigung neue Zeilen einfügen, geben Sie an, zwischen welchen Zeilen diese eingefügt werden müssen.
Hinweis: Es sind keine reinen Syntaxfehler im Code. `fraction` ist immer eine Zahl zwischen 0 und 1 und gibt den Anteil des ausgewählten Gewichts für jedes Objekt an. Die Berechnung im Code ist korrekt.

Zeile(n) des Fehlers	Berichtigung

(b) Sie kennen einen Greedy Ansatz aus der Vorlesung, mit dem dieses Problem effizient gelöst werden kann. Was ist eine Greedy Strategie, mit der das teilbare Rucksackproblem optimal gelöst werden kann? Antworten Sie in 2 Sätzen.

(c) Lösen Sie das teilbare Rucksackproblem für das hier gegebene Beispiel. Dabei ist `capacity` die maximale Kapazität W des Rucksacks und `items` ist die Liste mit möglichen Früchten, die in den Rucksack gelegt werden könnten. Füllen Sie die untenstehende Tabelle mit den gewählten Früchten und dem Anteil, zu dem Sie in den Rucksack kommen.

```
List<Item> items = new ArrayList<>();
items.add(new Item("Apples", 2, 4));    // Weight 2, Value 4
items.add(new Item("Berries", 2, 8));   // Weight 2, Value 8
items.add(new Item("Mangos", 1, 5));    // Weight 1, Value 5
items.add(new Item("Melons", 4, 12));   // Weight 4, Value 12
double capacity = 5;
```

Objekt (`Item.name`)	Anteil (`fraction`)

Aufgabe 4: Hashing ($4 + 5 + 4 = 13$ Punkte)

(a) Im Folgenden werden Hashtabellen zur Speicherung von Integer-Werten betrachtet, d.h., die Schlüssel sind vom Typ Integer. Kollisionen werden durch Lineares Sondieren (*linear probing*) mit Inkrement 1 aufgelöst.

Die folgende Hashtabelle der Größe 10 ist durch das Einfügen von Schlüsseln mit der Hashfunktion

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

gefüllt worden.

Index	0	1	2	3	4	5	6	7	8	9
Key			22	13	23	15	14	7	28	

Die Reihenfolge, in der die Schlüssel eingefügt wurden, ist unbekannt. Aber es können gewisse Aussagen über die Reihenfolge getroffen werden.

Nennen Sie alle Schlüssel, die möglicherweise als erstes eingefügt wurden:

Nennen Sie alle Schlüssel, die möglicherweise als letztes eingefügt wurden:

(b) Um noch weitere Schlüssel zu speichern, soll die Hashtabelle auf die Größe 15 vergrößert werden. Dazu müssen die Schlüssel aus der obigen Tabelle durch Rehashing in die neue, größere Hashtabelle übertragen werden. Durchlaufen Sie die ursprüngliche Hashtabelle entlang der Hashadressen und fügen Sie die Schlüssel mit der **neuen** Hashfunktion $hash_2(k)$ in die untere Hashtabelle ein.

$$hash_2(k) = k \mod 15$$

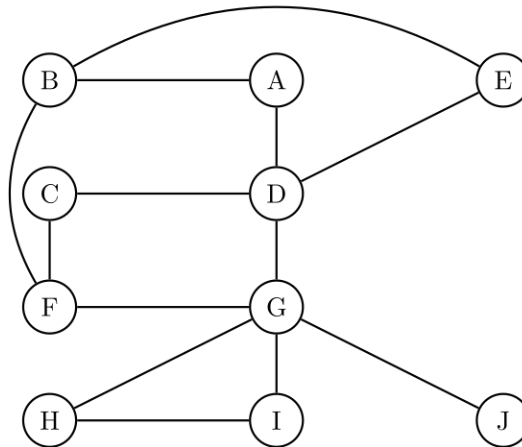
Key	$hash_2$(**key**)
7	7
13	13
14	14
15	0
22	7
23	8
28	13

Hinweis: Zur Vermeidung von Rechenfehlern sind die Hashadressen der Schlüssel unter der $hash_2$-Funktion in der rechten Tabelle gegeben.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Key															

Aufgabe 5: Breitensuche ($9 + 2 + 4 = 15$ Punkte)

Gegeben ist folgender Graph G:



(a) Führen Sie die Breitensuche auf dem obigen Graphen G aus. Fangen Sie bei **Knoten A** an und notieren Sie alle Knoten in der Reihenfolge, in der sie von der Breitensuche **in die Warteschlange geschrieben** werden.
Gehen Sie dabei davon aus, dass von jedem Knoten die benachbarten Knoten in alphabetischer Reihenfolge besucht werden. Z.B. wird die Kante **B**-**E** vom Algorithmus vor der Kante **B**-**F** bearbeitet.

A

(b) Welche Kante müssten Sie hinzufügen, damit J vor G in die Warteschlange geschrieben würde?

(c) Geben Sie für die Knoten $v =$ **A**, **F**, **G** und **J** vom Graphen G einen Zyklus minimaler Länge an, der den Knoten v enthält, wenn er auf einem Zyklus liegt. Andernfalls geben Sie an, dass v nicht auf einem Zyklus liegt.

A

F

G

J

Aufgabe 6: SSSP ($7 + 3 + 2 = 12$ Punkte)

Gegeben ist der folgende gerichtete und gewichtete Graph G:

(a) Führen Sie den Dijkstra-Algorithmus auf dem Graphen G aus. Beginnen Sie im Knoten s und notieren Sie die Knoten in der Reihenfolge, in der sie aus der indizierten Vorrangwarteschlange (IndexPQ) entfernt werden, d.h. in der Reihenfolge in der der Dijkstra-Algorithmus die kürzeste Distanz zu diesem Knoten findet. Geben Sie außerdem für jeden Knoten des Graphen, der vom Dijkstra-Algorithmus erfasst wird, die Distanz des kürzesten Pfades vom Startknoten aus zu dem jeweiligen Knoten an.

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

Bewertet werden in der unten stehenden Tabelle nur die Spalten **Knoten** und **Distanz zum Startknoten**. Die Spalten *Iteration* und *IndexPQ* müssen nicht ausgefüllt werden, können aber zur Lösung der Aufgabe verwendet werden.

Iteration	Knoten	Distanz zum Startknoten	*IndexPQ*
0	s	0	

(b) Zeichnen Sie den in a) gefundenen Baum der kürzesten Wege ein:

(c) Unter welcher Voraussetzung findet der Dijkstra Algorithmus mit indizierter Vorrangwarteschlange effizient kürzeste Wege? Welche Laufzeit kann dann garantiert werden?

Aufgabe 7: Minimaler Spannbaum (6 + 6 = 12 **Punkte**)

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

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

(b) Markieren Sie außerdem in jedem Graphen folgendes:

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

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

○ Prim ○ beide
○ Kruskal ○ keiner

○ Prim ○ beide
○ Kruskal ○ keiner

○ Prim ○ beide
○ Kruskal ○ keiner

○ Prim ○ beide
○ Kruskal ○ keiner

○ Prim ○ beide
○ Kruskal ○ keiner

○ Prim ○ beide
○ Kruskal ○ keiner

Aufgabe 8: Dynamische Programmierung ($3 + 4 = 7$ Punkte)

(a) In dieser Aufgabe geht es um eine kürzeste gemeinsame Supersequenz (SCS: *shortest common supersequence*) zu zwei gegebenen Strings. Mit Supersequenz wird hier eine Zeichenkette bezeichnet, die die beiden gegebenen Strings als Teilzeichenketten enthält. Die ursprünglichen Zeichenketten müssen in der Supersequenz nicht mehr zusammenhängen. Zu "ABCC" und "ACDC" ist "ABCDC" eine SCS. Zwei weitere Beispiele sind:

- Zu "GLUT" und "ULTRA" sind "GLULTRA" und "GULUTRA" beides SCSen.
- Zu "RASEN" und "KASINO" ist "KRASEINO" eine SCS.

Es sind Zeichenketten X und Y der Länge M, bzw. N gegeben. Definieren Sie eine rekursive Opt-Funktion, sodass $\textsc{Opt}(m, n)$ der Länge einer SCS der Zeichenketten $X[:m]$ und $Y[:n]$ entspricht ($0 \leq m \leq M$, $0 \leq n \leq N$). Dabei ist $X[:m]$ als der Präfix von X bis zum m-ten Zeichen definiert. Insbesondere ist $X[:0]$ eine leere Zeichenkette und es gilt $X[:M] = X$. Entsprechend ist $Y[:n]$ definiert.

Hinweis: Eine SCS der Zeichenketten ($X[:m]$, $Y[:n]$) kann wie folgt rekursiv bestimmt werden: Falls die letzten Zeichen der Strings übereinstimmen, hängt man dieses Zeichen an eine SCS von ($X[:m-1]$, $Y[:n-1]$) an. Falls sie nicht übereinstimmen, nimmt man die kürzere der SCS von ($X[:m-1]$, $Y[:n]$) und von ($X[:m]$, $Y[:n-1]$) und hängt das ausgelassene Zeichen ($X[:m]$ bzw. $Y[:n]$) an. Die Opt-Funktion kann entsprechend definiert werden. Es muss aber nur die *Länge* einer SCS bestimmt werden. Denken Sie auch an die Anfangsfälle.

(b) Für die Speicherung von Teilergebnissen bei der Dynamischen Programmierung wurden in der Vorlesung zwei unterschiedliche Ansätze besprochen. Nennen Sie für die folgenden beiden Opt-Funktionen das **jeweils** nach den Kriterien der Vorlesung effizientere Verfahren, und begründen Sie Ihre Wahl in einem Satz.

$$\textsc{Opt}_1(N) = \begin{cases} 0 & \text{falls } N = 0 \\ 1 + \min\{\textsc{Opt}(N - k^2) \mid k = 1, \dots, \sqrt{N}\} & \text{sonst} \end{cases}$$

$$\textsc{Opt}_2(N) = \begin{cases} 0 & \text{falls } N \leq 0 \\ 1 + \min\{\textsc{Opt}(N - c[k]) \mid k = 1, \dots, \text{len}(c)\} & \text{sonst} \end{cases}$$

Bei $\textsc{Opt}_2$ ist c mit $c[] = \{15, 20, 55\}$ gegeben.

Aufgabe 9: Zusammenhangskomponenten ($3 + 6 = 9$ Punkte)

(a) Skizzieren Sie einen ungerichteten Graphen (ohne reflexive Kanten und ohne Mehrfachkanten), der 9 Knoten, 9 Kanten und 3 Zusammenhangskomponenten der gleichen Größe (Anzahl der Knoten in der Zusammenhangskomponente) besitzt.

(b) Beschreiben Sie ein Verfahren als *Pseudocode*, mit dem die Zusammenhangskomponenten eines ungerichteten Graphen $G = (V, E)$ bestimmt werden können. Dazu soll in einem knotenindizierten Array `id[]` für jeden Knoten eine Kennung der entsprechenden Zusammenhangskomponente (Zahl zwischen 1 und der Anzahl von Zusammenhangskomponenten) gespeichert werden.

Das Verfahren muss auf dem Prinzip der *Breitensuche* basieren und eine Laufzeit in $\boldsymbol{O}(V+E)$ besitzen.

Hinweis: Schreiben Sie das gesamte Verfahren, **auch die angepasste Breitensuche**, als Pseudocode auf.

Diese Seite können Sie für Notizen verwenden. Bitte nur im Ausnahmefall für Lösungen verwenden!

Diese Seite können Sie für Notizen verwenden. Bitte nur im Ausnahmefall für Lösungen verwenden!