Aufgabe 1: Sortieren von Strings

Wenn man in Java mit Hilfe der `compareTo`-Methode Strings mit deutsche Wörtern sortiert, erhält man oft unbefriedigende Ergebnisse. Beispielsweise zeigt die folgende **sortierte Liste** einiger Buchstaben, dass Großbuchstaben vor den Kleinbuchstaben stehen und Umlaute nach den Kleinbuchstaben.

A B C a b c Ä Ö Ü ä ö ü

Eine alternative Sortiermethode besteht darin, die Groß/Klein-Schreibung zu ignorieren und Umlaute durch ihre Ersatzdarstellungen ("ae", "ue", "oe") und 'ß' durch "ss" zu ersetzen.

Dazu soll eine Klasse `public class MyStringComparator implements Comparator<String>` programmiert werden, die eine Normierungs-Methode `String normalize(String x)` und eine Vergleichsmethode für Strings enthält.

```
import java.util.*;

public class MyStringComparator implements Comparator<String> {
   private String normalize(String x) { // Fehlende Implementierung }

   public int compare(String a, String b) {
      String local_a = normalize(a);
      String local_b = normalize(b);
      return local_a.compareTo(local_b);
   }
}
```

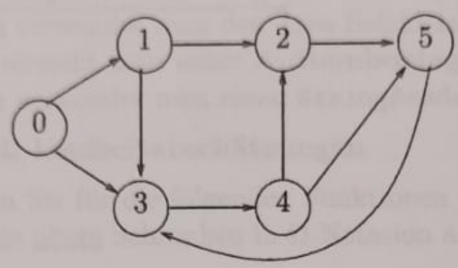
a) Programmieren Sie die Methode `String normalize(String x)` in Java. Dazu sollten Sie in einer Schleife den String `x` zeichenweise durchlaufen und zu jedem einzelnen Zeichen die normierte Darstellung erzeugen und diese zum Ergebnis-String zusammenfügen.

b) Programmieren Sie ein Java-Hauptprogramm, das alle Strings der Kommandozeile `String argv` in ein `TreeSet<String>` einträgt, das die Strings unter Verwendung der neuen `compare`-Methode sortiert. Anschließend soll die sortierte Folge der Strings auf `System.out` ausgegeben werden.

Aufgabe 2: BFS-Algorithmus

Ein gerichteter Graph mit der Knotenmenge $V = \{0, 1, 2, 3, \dots, n-1\}$ kann durch die Anzahl n seiner Knoten und eine `HashMap<Integer, ArrayList<Integer>>` dargestellt werden. Dabei enthält die `HashMap` zu jedem Knoten i ~~die~~ die Liste der direkt von ihm erreichbaren Nachbarknoten.

Beispiel:



i	Nachbarn des Knotens i
0	1, 3
1	2, 3
2	5
3	4
4	2, 5
5	3

a) Programmieren Sie den BFS-Algorithmus unter Verwendung einer solchen "Nachbarschafts-HashMap" als eine Java-Methode:

```
ArrayList<Integer> BFS(int n, int s,
                       HashMap<Integer, ArrayList<Integer>> nachbarn)
```

Der Ablauf des Algorithmus ist im folgenden (wie im Skript) grob skizziert:

```
L    ← Liste, die nur das Element s enthält;
d[v] ← undefiniert für alle v ∈ V \ {s};
d[s] ← 0;
while L ≠ ∅
   {
    entferne das erste Listenelement v aus L;
    Für alle von v ausgehenden Kanten (v,w):
      if (d[w] == undefiniert)
        {
         d[w] ← d[v] + 1;
         füge w am Ende der Liste L an;
        }
   }
```

b) Geben Sie die größenordnungsmäßige Laufzeit Ihrer Implementierung in O-Notation an (unter Berücksichtigung der Zugriffe auf die `HashMap` und `ArrayList`).

ToDo

c) Welche größenordnungsmäßige Laufzeit würde sich ergeben, wenn man statt der `ArrayList` eine `LinkedList` verwendet hätte? Begründen Sie Ihre Antwort.

Aufgabe 3: OOP in Java

a) Was versteht man unter einer **Klassenvariable**?
b) Worin unterscheiden sich **Interfaces** und **abstrakte Klassen**?
c) Welche Bedeutung hat das Schlüsselwort `throws`?
d) Erklären Sie, warum das Interface `Comparable` mit seiner Methode `int compareTo(Object o)` als "nicht typsicher" bezeichnet wird?
e) Wozu verwendet man den Java-Befehl `instanceof`?
f) Was versteht man unter **Autounboxing**?
g) Wozu verwendet man einen `StringReader`?

Aufgabe 4: Laufzeitabschätzungen

a) Geben Sie für die folgenden Funktionen jeweils möglichst gute größenordnungsmäßige obere Schranken in Ω-Notation an.

$$f(n) = \left(n^2+1\right)\cdot(3n+1)$$

$$g(n) = \begin{cases} 2^n & \text{für } n \le 4 \\ 16-(n-4)^2 & \text{für } 4 < n \le 8 \quad -n^2 \\ 2\cdot(n-8) & \text{für } n > 8 \quad n \end{cases}$$

$$h(n) = n\cdot\log_2 n + \sqrt{n}$$

b) Geben Sie für die angegebene rekursive Java-Methode `int f(int n)` den rekursiven Ansatz zur Abschätzung der Laufzeit $T_f(n)$ an und berechnen Sie damit eine obere Schranke für die Laufzeit.

```
int f(int n) {
 if (n>=2)
   return n%2 + f(n/2);
 else
   return n;
}
```

c) Die Laufzeit $T_g(n)$ der folgenden Java-Methode `int g(int n)` ergibt sich aus dem Rechenaufwand für die verwendeten Schleifen. Geben Sie eine möglichst gute größenordnungsmäßige obere Schranke für die Laufzeit an.

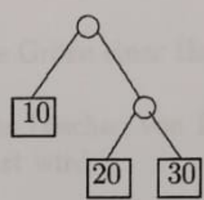
```
int g(int n) {
  int y=0;
  for (int i=0; i*i<n; ++i)        log n
     for (int k= i+1; k<n; ++k)    n
        ++y;
  return y;
}
```

Aufgabe 5: Karatsuba-Multiplikation

a) Berechnen Sie nach der Karatsuba-Methode den Wert a^2 für die 6-stellige Binärzahl $a = 010110$ indem Sie die 6-stellige Multiplikation auf drei kleinere Multiplikationen zurückführen. Geben Sie alle dabei entstehenden Zwischenergebnisse an.
(Die kleineren Multiplikationen müssen nicht weiter zerlegt werden.)

b) Berechnen Sie nach der Karatsuba-Methode das Produkt $a \cdot b$ für die 4-stelligen Dezimalzahlen $a = 1203$ und $b = 5025$ und geben Sie alle dabei entstehenden Zwischenergebnisse an.
(2- und 3-stellige Zahlen müssen nicht weiter zerlegt werden).

Aufgabe 6: Balancierte Bäume



a) Fügen Sie in den gegebenen Baum von beschränkter Balance $\alpha = \frac{1}{3}$ nacheinander die Werte 25, 26, 27 und 31 in dieser Reihenfolge ein und führen Sie dabei jeweils eine Rebalancierung durch, wenn der Baum nicht mehr von beschränkter Balance $\frac{1}{3}$ ist.

b) Wieviele Knoten eines Baumes von beschränkter Balance α können maximal aufgrund einer einzelnen Einfüge-Operation "aus der Balance geraten?"

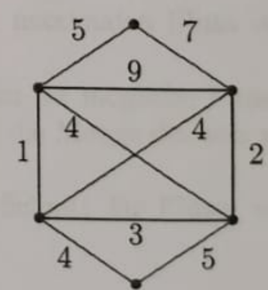
c) Welche größenordnungsmäßige Laufzeit braucht man in einem balancierten Baum von beschränkter Balance α zur Durchführung einer "Doppelrotation"?

d) Wie wirkt es sich auf die Daten-Zugriffszeiten und die Anzahl der notwendigen Rebalancierungen aus, wenn man den α-Wert für balancierte Bäume verkleinert?

Aufgabe 7: Kruskal-Algorithmus

a) Bestimmen Sie für den gegebenen Graphen mit dem Kruskal-Algorithmus einen minimalen aufspannenden Baum und geben Sie sein Gewicht an.

b) Skizzieren Sie die *Union-Find*-Datenstruktur, die bei der Ausführung des Kruskal-Algorithmus in Teil a) entsteht.



Aufgabe 8: Hashing mit offener Adressierung

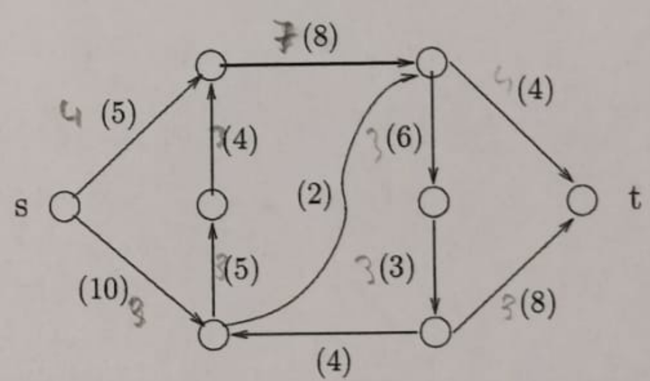
Adresse:	0	1	2	3	4	5	6
Daten:	70			17	18	12	20

In der angegebenen Hashtabelle der Größe $m = 7$ wurden bereits 5 Daten eingetragen. Die Hashfunktion lautet:

$$h(x, i) = [(x \bmod 7) + i \cdot (1 + (x \bmod 3))] \bmod 7$$

a) Geben Sie für das angegebene Zahlenbeispiel die durchschnittliche Anzahl der Tabellenzugriffe für eine erfolgreiche Suche an.
b) Ermitteln Sie für die Werte $x = 4$, $x = 5$ und $x = 6$ wieviele Zugriffe auf die Hashtabelle benötigt werden, um jeweils festzustellen, dass diese Daten **nicht eingetragen** sind.
c) Erklären Sie kurz, warum die Größe einer Hashtabelle immer eine Primzahl sein sollte.
d) Erklären Sie kurz, warum das Löschen von Einträgen beim Hashing mit offener Adressierung nicht unterstützt wird.

Aufgabe 9: Flussproblem



a) Skizzieren Sie für das angegebene Flussproblem einen maximalen Fluss von s nach t. (Es ist egal, wie Sie diesen Fluss finden)
b) Konstruieren Sie mit dem Fluss aus a) den Hilfsgraphen für mögliche Flussverbesserungen und bestimmen Sie in diesem Hilfsgraphen die Menge der von s aus erreichbaren Knoten.
c) Skizzieren Sie im gegebenen Graphen den minimalen Schnitt für Flüsse von s nach t.