Aufgabe 1: Sortierverfahren 1 (10 Punkte)

Sortieren Sie die folgende Liste von Zahlen aufsteigend mit dem Verfahren *Insertion Sort*:

38 18 5 21 29 14 35

Geben Sie die Liste nach jedem Durchlauf der inneren Schleife an, d.h. nach jedem vollständigen Einsortieren eines Elements.

Aufgabe 2: Sortierverfahren 2 **(10 Punkte)**

Sortieren Sie die folgende Liste von Zahlen aufsteigend mit dem Verfahren *Quick Sort*:

38 18 14 29 21 5 35

Geben Sie für jeden Rekursionsschritt an, welches Pivot-Element gewählt wurde.

Aufgabe 3: AVL-Bäume **(10 Punkte)**

Fügen Sie die folgenden Zahlen nacheinander in einen *AVL-Baum* ein:

42 16 12 19 38 1

Zeichnen Sie den Baum vor und nach jeder durchgeführten Rotation. Geben Sie auch jeweils an, was für Rotationen Sie durchführen.

Aufgabe 4: Sortierverfahren (10 Punkte)

Erläutern Sie die Funktionsweise des folgenden Sortierverfahrens. Erläutern Sie auch, welche Einschränkungen bzw. Anforderungen an die Liste gelten müssen, damit das Verfahren korrekt funktioniert, und was daran ggf. nicht optimal oder sinnvoll ist.

```
1  bool Bar(std::vector<int> list) {
2    if (list.size() <= 1) {
3      return true;
4    }
5    int a = list[0];
6    list.erase(list.begin());
7    if (a > list[0]) {
8      return false;
9    }
10   return Bar(list);
11 }
12
13 void FooSort(std::vector<int> &list) {
14   while (!Bar(list)) {
15     size_t i = rand() % list.size();
16     size_t j = rand() % list.size();
17
18     int h = list[i];
19     list[i] = list[j];
20     list[j] = h;
21   }
22 }
```

Aufgabe 5: Datenstrukturen (10 Punkte)

Erläutern Sie die Idee und Funktionsweise des folgenden Datentyps. Diskutieren Sie kurz die Vor- und Nachteile gegenüber ähnlichen Datentypen aus der Vorlesung.

```
struct FooType {
  std::vector<int> values;
  std::vector<FooType *> children;

  void AddToChild(int i, int value) {
    while (children.size() <= i) {
      children.push_back(new FooType());
    }
    children[i]->Add(value);
  }

  void Add(int value) {
    if (values.size() < 2) {
      values.push_back(value);
      return;
    }
    if (value < values[0]) {
      AddToChild(0, value);
      return;
    }
    if (value < values[1]) {
      AddToChild(1, value);
      return;
    }
    AddToChild(2, value);
  }
};
```

Aufgabe 6: Komplexitaet (10 Punkte)

Betrachten Sie die folgende Funktion `SameElements()`. Die Funktion erwartet zwei `int`-Listen und prüft, ob diese beiden Listen die gleiche Menge an Elementen enthalten. D.h. ob jedes Element aus der einen Liste auch in der anderen enthalten ist. Dabei spielt es keine Rolle, ob die Länge der Listen gleich ist bzw. ob die Elemente in der gleichen Anzahl vorkommen.

```
1  bool same_elements(std::vector<int> list1, std::vector<int> list2) {
2    for (int el : list1) {
3      bool contained = false;
4      for (int el2 : list2) {
5        if (el == el2) {
6          contained = true;
7        }
8      }
9      if (!contained) {
10       return false;
11     }
12   }
13
14   for (int el : list2) {
15     bool contained = false;
16     for (int el2 : list1) {
17       if (el == el2) {
18         contained = true;
19       }
20     }
21     if (!contained) {
22       return false;
23     }
24   }
25   return true;
26 }
```

a) Bestimmen Sie die Komplexität dieser Funktion. Geben Sie in O-Notation an, wie oft die Vergleiche `if v1 == v2` durchgeführt werden. Begründen Sie Ihr Ergebnis.

b) Machen Sie einen Vorschlag, wie diese Funktion optimiert werden kann. Erläutern Sie, inwiefern dieser eine bessere Komplexität hat.

Hinweis: Sie müssen keinen konkreten, vollständigen Algorithmus angeben.