1. Permutations

Given a string S, find all permutations of the string that contain all characters of the original string.

Example:

```
Input: S = “ABC”
Output: {“ACB”, “CAB”, “CBA”, “BCA”, “BAC”}
```

Think about an algorithm that can perform this task. You may implement it in Java, but you can also write it down in pseudocode or, at least, try to visualize the execution of an algorithm.

What is the complexity of this algorithm?

Now let’s change the algorithm a little bit.

Given a string S, find all *k* sized permutations of the string that contain all characters of the original string. Where *k* is a positive integer and $k < |S|$ that represents the length of each resulting permutation.

Example:

```
Input: S = “ACB”
k = 2
Output: {“AB”, “AC”, “BA”, “BC”, “CB”, “CA”}
```

Seeing that the size of the output changed, surely the complexity must also have changed. What do you think?

What is the complexity of this algorithm?

2. Minimum binary subarray

Given binary array find the length of a subarray with the *minimum* number of 1s.
Note: There is at least a single 1 present in the array.

Example:

```
Input : arr[] = {1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1}
Output : 3
Minimum length subarray of 1s is {1, 1, 1}.

Input : arr[] = {0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1}
Output : 2
Minimum length subarray of 1s is {1, 1}.
```

Think about an algorithm that can perform this task. You may implement it in Java, but you can also write it down in pseudocode or, at least, try to visualize the execution of an algorithm. What is the complexity of the solution you found? Can you think of a way to improve the complexity?

3. List comparison

Below you find the pseudo code of four algorithms to compare two lists (each element is unique in the list). Have a look the implementations.
Analyse the run time of each of the algorithms? Consider different implementations of the list (e.g. Linked List, Array List) and describe the run time for each of these implementations.

```
compare1 (data a, data b):
      for each i in a
            found <-false
            for each j in b
                  if i = j
                        found <- true
            if not found
                  return false
      return true
```

```
compare2 (data a, data b):
      for each i in a
            found <-false
            for each j in b
                  if i = j
                        found <- true
                        break;
            if not found
                  return false
      return true
```

```
compare3 (data a, data b):
      for each i in a
            found <-false
            for each j in b
                  if i = j
                        found <- true
                        remove j from b;
            if not found
                  return false
      return true
```

```
compare4 (list a, list b):
        a <- sort(a)
        b <- sort(b)
        for (int i = 0; i < len(a) && i < len(b);i++)
                if (a[i] != b[i])
                        return false

        return true
```

```
compare5 (list a, list b):
      b <- sort(b)
      for (int i = 0; i < len(a) && i < len(b);i++)
            if(not b.find(a[i]))
                  return false
      return true
```

4. Graph algorithms

In a graph, let the value of a neighbourhood be the sum of the values of the node itself and each of its neighbors. The task is now to find the nth largest neighbourhood in a graph. Below you find the pseudo code of 3 different algorithms to calculate this value.
One which parameters does the run time depend?
What is the run-time in big O notation?
What type of list() should be used in find1 and fine2? How does it effect the run time?
How does the memory consumption differ in the three algorithms?

```
find1(n,graph):
      values = list()
      for each node node:
            sum = node.getValue()
            for e in graph.getNeighbours(node):
                  sum += e.getNode().getValue()
            values.add(sum)
      values.sort()
      return values[n]
```

```
find2(n,graph):
      values = list()
      for each node node:
            sum = node.getValue()
            for e in graph.getNeighbours(node):
                  sum += e.getNode().getValue()
            values.add(sum)
      i = 0
      max = 0
      while (i < n):
      max = 0
            for each e in values:
            if (e > max):
                  max = e
      remove max from values
return max
```

```
find3(n,graph):
i = 0
      max = 0
      while (i < n):
max = 0
for each node node not marked:
      sum = node.getValue()
      for e in graph.getNeighbours(node):
                        sum += e.getNode().getValue()
      if (sum > max):
                  max = sum
      mark max
      return max
```