Module IN2002—Data Structures and Algorithms
Exercise Sheet 1 (Sample Answers)

Note that you may not have enough time to finish this during the tutorial slot. Get started, see where you have difficulties, study, ask questions, and try again. You should aim to finish all the questions before the next lecture.

- IMPORTANT LESSON FROM THIS TUTORIAL: You have to know how to understand code. In your professional life you will have to work on code developed by others, develop code with others, and fix existing code (whether you have written it or not).
- What is the best way to find out what some algorithm does? It is most certainly not staring at it and hoping that it will come to you. The best way is to choose a small (if a positive integer then say 0 or 1) input value and run through the algorithm taking note of how the values in the variables change and what output is generated. Then try again for another value (if integer then one above the previous one) and so on, until you can identify what the pattern is.

You will find some sample answers below. Your justifications to support your answers might be slightly different, and you algorithm may also be different. This does not mean that they are wrong. If in doubt post your solution on Moodle, ask the module leader during office hours, or ask the TAs during surgery hours
.

1. Consider the following pseudocode :

```
Function foo(array)
i ← 1
WHILE array[i] mod 23 != 0 AND i <= length of array
        i ← i+1
Return i
```

(a) Describe in plain English what this algorithm does.
- This algorithm finds the position of the first element in an array that is divisible by 23.

(b) What is the time complexity with respect to the array size *n* of the algorithm described in the worst case, best case, average case? Justify your answer and state assumptions you make for the average case.
- In the best case, the first element is divisible by 23. Then we have only 1 processing step independent of the size of the array, therefore the algorithm's time complexity is in O(1) for the best case.
- In the worst case, the whole array needs to be searched (no element or only the last element are divisible by23). Then for an array of size *n*, *n* processing steps are needed, therefore the algorithm's time complexity is in O(n) for the worst case.
- In the average case we assume that there is one element divisible by 23 and it is equally likely to appear at any position in the array. Then we need O(*n/2)*steps to find the position on average (which is O(n)).

2. Suppose a program consists of two phases executed one after the other. The program takes a string as input, which may be of any length $n \in \mathbb{N}$. What is the time complexity of the whole program, with the phases being of the following complexities?

- When two phases are executed sequentially, the execution times add. The O notation depends on the fastest growing term in a sum, as it will dominate when n is large.

(a) $O(n^2)$ and $O(n \log n)$
- $O(n^2)$

(b) $O(\log n)$ and $O(n)$
- $O(n)$

(c) $O(n)$ and $O(n)$
- $O(n)$

3. Consider the following pseudocode:

```
Function foo(k)
s ← 1
WHILE k > 0
        s ← s * 2
        k ← k-1
Return s
```

(a) What does *foo* compute?

i. nothing
- No, it returns a value

ii. k^2
- No, k controls only the number of loop executions.

iii. s * k
- No, s is just an internal variable.

iv. 2^k
- Yes, in each loop s is multiplied by 2, and the loop is executed k times.

(b) What is the time complexity of *foo* with respect to *k* in the worst case?

i. O(1)
- No, the number of loop executions depends on k.

ii. $O(2^k)$
- No, there are k loop executions where each takes the same amount of time.

iii.O(s)
- No, s is not part of the input.

iv. O(k)
- Yes, there are k loop executions.

(c) How is the complexity different for the best or average case?
Justify your answers.

- There is no difference, the running time is determined uniquely by *k*.

4. Create an algorithm that given an array of *k* integers returns the number of times that the number 100 is present in the array. Then answer the questions below.

- Here is one possible solution.

```
Function count100s(array)
index ← 1
counter ← 0
WHILE index <= length of array
        IF array[index] = 100
            counter ← counter + 1
        index ← index+1
Return counter
```

(a) What is the time complexity of your algorithm? Justify your answer.
- There is one single loop, which is executed as many times as the length of the array. So, its time complexity is O(k).

(b) What is its space complexity? Justify your answer.
- The only new space required by the algorithm is that of the two variables "index" and "counter". Hence the space complexity is constant, O(1).

(c) Would it make a difference if the input array was sorted? Justify your answer.
- As it stands (unsorted) we must traverse the whole array. If the array was sorted, the search could stop as soon as we found the first number larger than 100. We might even consider something like binary search to not even need to check all elements that are below 100.
- All of this would mean an improvement in the average and best case time complexities. Worst case would then be if all values in the array are 100, in which case we would indeed need to traverse the full array.
- In terms of space complexity it depends. If we went for a recursive solution then we would change the space complexity, making it higher than constant.

5. Consider the following pseudocode:

```
Function bar(k)
WHILE k > 0
        i ← k
        s ← 0
        WHILE i > 0
                s ← s + i
                i ← i -1
        array[k] ← s
```

k ← k – 1
Return array

(a) What does *bar* compute?

i. []
- No, there are values being written into the array.

ii. [1,2,3, ... , k]
- No, the inner loop calculates something else.

iii. [1,1+2,1+2+3, ... , 1+ ... +k]
- Yes, each inner loop calculates the sum 1 + ... + k. Note that the array is filled from back end.

iv. k * s
- No, the output is an array.

(b) What is the time complexity of *bar* with respect to *k* in the worst case:

i. O(1)
- No, there are loops that depend on k.

ii. O(s)
- No, s is just an internal variable.

iii. O(k)
- No, there are two nested loop depending on k.

iv. $O(k^2)$
- Yes, there are two nested loops. The outer loop is executed k time, the inner loop k/2 times on average. The overall complexity is therefore $O(k*k/2) = O(k^2)$.

(c) Is the complexity different for the best or average case?
- No difference, the running time is determined uniquely by *k.*

(d) How could the computation in *bar* be done in a more efficient way? (You need to remember your Maths knowledge to do this. This link will help: http://www.cut-the-knot.org/Curriculum/Algebra/GaussSummation.shtml). Justify your answers.
- By using 1+ .. +n = n(n+1)/2, the inner loop can be removed, making bar run in O(k) time.