Module IN2002—Data Structures and Algorithms

Exercise Sheet 1 (Sample Answers)

*** IMPORTANT! Make sure that you check Moodle for the pre-requisites to succeed.***

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 three questions during the tutorial. We expect you to finish all of these, all additional questions, as well as have checked the pre-requisites to succeed 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. We STRONGLY encourage you to only look at the sample answers below once you have created your own answers. Your justifications to support your answers might be slightly different, and your algorithms may also be different. This does not mean that they are wrong. If in doubt post your solution on Moodle, or ask a member of the module team during their office hours.

Question 1

Consider the following pseudocode, where *array* is an array of positive integers:

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

(a) Describe in plain English what this algorithm does.

Answer:

- This algorithm finds the position of the first element in an array that is a multiple of 23. If there are no multiples of 23, it returns the length of the array + 1.
- How to get this?? By trying out different values of input (sizes and contents of array - e.g., 2, 23, 46; then 23, 2; then 1).

(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 any assumptions you make.

Answer:

- 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 Ω(1); this is the best case.
- In the worst case, the whole array needs to be searched (no element or only the last element are divisible by 23). Then for an array of size *n*, *n* processing steps are needed, therefore the algorithm's time complexity is in O(n); this is 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 in the order of *n/2* steps to find the position on average (which is in the order of n).

Question 2

Consider the following pseudocode, where ***stk*** is a stack of *n* positive integers:

```
Function abc(stk)
WHILE !stk.isEmpty()
        Print( stk.pop() )
Return
```

(a) Describe in plain English what this algorithm does.

Answer:

- This algorithm prints the contents of the stack received as input. It starts from the latest value pushed into the stack and continues popping (and printing) values until it reaches the value that had been in the stack the longest. If the stack is empty, this function does nothing.
- How to get this?? By trying out different values of input (sizes and contents of the stack – e.g., start with an empty stack, then a stack with one value, then a stack with two values).

(b) What is the time complexity with respect to the number of items in the stack (*n*) of the algorithm described in the worst case, best case, average case? Justify your answer and state any assumptions you make.

Answer:

- The best, worst and average time complexities are all the same here. The loop will always execute the same number of times as there are items on the stack. So, it is linear for all of them, meaning order of n (best is Ω(n), worst is O(n) and average is in order n).

Question 3

Create an algorithm that given a sorted array of *k* integers and a stack of *n* integers returns a sorted array containing all the elements from the original array and the elements from the stack. Assume that all elements from the stack were not originally present in the array, they are in no particular order and there are no duplicate values in either the array or the stack.

Explain in plain English the underlying principles of your algorithm (e.g., does it use some auxiliary structures, what are the composing parts of your algorithm, how do you handle edge cases, does the algorithm for your function call other functions?)

Answer:

- As always with algorithms, there are multiple possibilities for answering this question correctly. Here is one possible solution.

Explanation in plain English: In this algorithm we will add each entry from the stack on its rightful place on the array. For adding each entry, we traverse the array to find the appropriate position, shuffle everything to make space for the new entry, and then add the new entry. This is wasting resources, as it is very time consuming. This is analogous to an insertion sort.

The algorithm itself in pseudocode:

```
Function mergeSarrayNstack(array, stack)
// In this algorithm we will add each entry from the stack on its rightful place on the
// array. We are traversing the array and shuffling everything to make space for the
// new entry. This is wasting resources, as it is very time consuming. This is
// analogous to an insertion sort.
WHILE !stack.isEmpty()
        // if we are here it means that here is still something in the stack to be added
        toAdd ← stack.pop()
        index ← 1
        WHILE (index <= length of array) AND (array[index] < toAdd)
                // here we are looking for the right location to place the value just
                // retrieved from the stack.
                index ← index+1
        IF (index == length of array + 1)
                // the new value needs to be added at the end. Note that this is
                // ignoring the possibility of this action trying to add a value out of
                // bounds (memory allocated for the array). How can one handle that?
                array[index] ← toAdd
        ELSE
                // the new value needs to go where index is, so we must move up all
                // the values after it to make room for the new entry. Here again, what
                // happens if this takes us out of bounds? How can we handle that?
                current ← length of array +1
                WHILE (current > index)
                        // move values up one
                        array[current] ← array[current -1]
                        current ← current -1
                // now we can add the new value in the correct place
                array[index] ← toAdd
Return array
```

- The sample answer above is analogous to having an insertion sort. You already know that this is terribly wasteful in terms of time complexity. Can you provide a better answer? Hint: you can use other structures; you don't need to do the addition in place on the input array.

(a) What is the worst case time complexity of your algorithm? Justify your answer.

Answer:

- The sample answer above has more than one loop, including nested loops. It has no recursive calls and no call to another function.
- The outer loop (while the stack is not empty) will be executed as many times as there are elements in the stack, so O(n).
- The next loop (looking for the place to add the new value) will be executed as many times as we need until we find the position to add the new value; this will be 0 times ($\Omega(1)$) in the best case and O($k+n$) in the worst case (k+n could happen when you are adding some value from the stack that needs to go to the end of the array, after you have already added lots of other values from the stack).
- Within the ELSE, we have another loop to make room for the new value. The best case here would be to add it to the end, going 0 times through the loop, $\Omega(1)$. The worst case is adding it to the first position after having already added lots of other values from the stack, which would give us O($n+k$).
- As the last two loops are sequential, we just add them up, giving us O($n+k$).
- These two are nested in the other one, so we multiply giving us O($n(n+k)$) or O(n^2+nk). Since we don't know whether n is greater than or smaller than k, we cannot reduce this any further.

(b) What is its worst case space complexity? Justify your answer.

Answer:

- The only new space required by the algorithm from the sample answer is that of the three variables "toAdd", "index" and "current". Hence the space complexity is constant, O(1). There is no recursion to consider and no calls to other functions. The amount of space that is being used to extend the array is being freed from the stack.

(c) Would it make a difference if the input array was not sorted? Justify your answer.

Answer:

- If the input array was not sorted, in order to use this solution, we would need to sort it first. So, we could for example run insertion sort on the array first and then insertion sort on each of the new values retrieved from the stack. Justification… if it is not sorted, we cannot guess where to add the new values.
- Again, this is really terribly wasteful. Can you think of better ways of doing this?