Assignment #1
Data Structures and Algorithms for AI
Computer Science Department, VU Amsterdam

Caspar Grevelhoerster

September 24, 2021

Instructions: This is the first assignment of the course. The assignment consists of four questions and they appear below. Your job is to answer the questions and submit your solutions in PDF format via Canvas.

If you like to discuss something regarding the assignment, you are only allowed to discuss with your instructor. In particular, you are NOT allowed to discuss the assignment with someone else. The deadline to submit your file is before the end of Sunday September 26, 2021.

QUESTION 1. [2 POINTS] Please read the following algorithm and then answer the next short questions. You do not need to explain (i.e., justify) your answers and only the short answers are sufficient.

Algorithm 1 Input: An array A of n integers.

```
for i = 0 to n − 1 do
  if A[i] < 0 then
    b = 1
    while b < n do
      b = b × 2
    end while
  end if
end for
```

- What is the running time of the algorithm in the worst-case using O? O (n*log(n))
- What is the running time of the algorithm in the best-case using O? O (n)
- What is the running time of the algorithm in the best-case using Ω? Ω (n)
- What is the running time of the algorithm in the worst-case using Ω? Ω (n*log(n))

QUESTION 2. [3 POINTS] Determine whether each of the following statements is TRUE or FALSE. Justify your answers clearly.

- Let $f(n)$ and $g(n)$ be two functions. If $f(n) \in O(g(n))$, then $2^{f(n)} \in O(2^{g(n)})$. If $f(n) = 2log(n)$ and $g(n) = log(n)$ then $f(n) \in O(g(n))$ holds since clearly $2log(n) \in O(log(n))$. However, when both f as well as g are raised over two, so $2^{f(n)}$, $2^{g(n)}$ then $2^{f(n)} \notin O(2^{g(n)})$ since $2^{f(n)} = O(n^2)$, $2^{g(n)} = O(n)$ and $n^2 \notin O(n)$. Therefore, the statement is FALSE.
- MERGE SORT is a sorting algorithm whose running time is in $\Theta(n \log n)$, in the best-case, and whose running time is in $\Theta(n^2)$, in the worst-case. Merge sort is a divide-and-conquer algorithm that first divides the array into n sub-parts which takes constant time of $\Theta(1)$. The next operation is to recursively merge the sub-arrays. This takes $log(n)$ steps since the number of sub-parts halves for each iteration or in other words: if the divided

arrays are displayed in a tree, there are $log(n)$ levels of the division-tree. At each level of the tree (or in each recursion of the merging operation), each element is analyzed to put it into order correctly - which implies that for each $log(n)$ recursions, n elements are analyzed. Thus $\Theta(log(n)) * \Theta(n) = \Theta(n * log(n))$. This explanation, however, also implies that there is no possibility for the algorithm to perform worse than $\Theta(n * log(n))$: The worst-case of Merge Sort is not of order $\Theta(n^2)$ but is instead "faster". This proves the statement to be FALSE.

- $n! \in O(n^n)$ and $3^n \in \Omega(2^n)$. Since O is upper bound, it is required that for two functions f and g where g is bound by O, that $\forall n \geq 1, f \leq g$. Plotting the functions $n!$ and n^n shows that $\forall n \geq 1, n! \leq n^n$. Therefore, the first sub-statement is TRUE. Since Ω is lower bound, it is required that for two functions f and g where g is bound by Ω, that $\forall n \geq 1, f \geq g$. Plotting the functions 3^n and 2^n shows that $\forall n \geq 1, 3^n \geq 2^n$. Therefore, the first sub-statement is also TRUE. Thus, it can be said that the overall statement is TRUE.

QUESTION 3. [3 POINTS] Describe an algorithm (either in pseudocode or in text) that, given an array A of n integers, determines whether or not there exists a duplicate in A. For instance, if the input is $A : 5, 2, 3, 5, 1, 7, 9, 5$, then the algorithm has to return YES, because of the two 5's in A. (Actually, there are three 5's in A, but two is sufficient for the algorithm to return YES.) But, if the input is, for instance, $A : 6, 9, 1, 0, 4, 7$, then the algorithm has to return NO, because all numbers in A are unique and thus there is no duplicate in A.

Also, analyze the running time and space complexity of your algorithm in the worst-case. Please be as precise as you can in both describing your algorithm and its time and space requirement analysis.

[Note 1. Your algorithm is expected to be a time-efficient algorithm. In other words, if there exist algorithms faster than your algorithm, then you will get only a partial credit, assuming your (inefficient) algorithm is correct.]

[Note 2. If you need, you are allowed to use any of the algorithms and/or data structures we have discussed in the course so far, without discussing their details.]

Answer:

Algorithm 2 Input: An array A of n integers.

```
A = MergeSort(A)
for i = 0 to n − 2 do
  if A[i] == A[i + 1] then
    return YES.
  end if
end for
return NO.
```

This algorithm takes an array of size n and sorts it first using MERGE SORT. As explained in question 2.2, the latter algorithm has a time complexity of $\Theta(n * log(n))$. Afterwards, a for-loop iterates over each element of the array which makes for a time complexity of $\Theta(n)$. These two steps combine to $\Theta(n * log(n)) + \Theta(n) = \Theta(n + n * log(n))$ but since Θ is only about the order and constants are ignored, the time complexity reduces to $\Theta(n * log(n))$.
The space complexity of merge sort is at most $O(n)$ since it is using array(s) n elements, which it divides and then fuses. The linear space complexity of $O(n)$ is never exceeded since the for-loop only iterates over it without affecting space.

QUESTION 4. [2 POINTS] A string over the characters {,},(,),[,] is said to be well-formed if the different types of brackets match in the correct order. For example, the strings ([]){()} and [()[]{()()}] are well-formed, but the strings {), {(}), [()[]{()() are not. The goal of this exercise is to design a time-efficient algorithm that, given a string S over the {,},(,),[,], decides whether S is well-formed or not. [1]

The brute-force (i.e., straightforward) algorithm for this problem is to scan S from left to right and at each closed character c (e.g., } or) or]) in S, find the first open character c' which lives in S before c. If c and c' match, then

[1] This problem has clear applications in designing programming languages when they want to parse and evaluate arithmetic expressions that contain different types of brackets.

replace each of them with a 0 is S. In the end, if S contains only 0s, then return that S is well-formed. Otherwise, S is not well-formed. The following pseudocode summarizes this brute-force lazy algorithm.

Algorithm 3 Input: A string S of length n containing only characters {, }, (,), [, and].

```
for i = 0 to n − 1 do
  if S[i] is a closed character then
    Find the largest 0 < j < i such that S[j] is S[i]'s open version.
    Replace S[i] and S[j] with 0.
  end if
end for
if S contains only 0s then
  Return YES.
else
  Return NO.
end if
```

As you might have guessed, the running time of this algorithm is in $\Omega(n^2)$ in the worst-case. This is because, if the input is, for instance S: [[[[$\cdots$ [ ] $\cdots$]]]], then the brute-force algorithm requires $\Omega(n^2)$ time. (S is of length n and its first half characters are [and its second half characters are].) Now, we are interested to design a faster algorithm—an algorithm whose running time is $O(n)$ in the worst-case!

Present an algorithm (either in pseudocode or in text) that, given a string S of length n, decides whether S is well-formed or not. The running time of your algorithm is expected to be in $O(n)$ in the worst-case. In addition, present a neat analysis of the running time and the space requirement of your algorithm. (Hint. Use a stack.)

Answer:

Algorithm 4 Input: A string S of length n containing only characters {, }, (,), [, and].

```
stack = stack()
for i = 0 to n − 1 do
  if S[i] is the fitting closed character to top element (opened character) on stack then
    remove top element from stack
  else
    add S[i] to stack
  end if
end for
if stack is empty then
  Return YES.
else
  Return NO.
end if
```

TIME COMPLEXITY. This algorithm takes an array of size n and iterates over each element using a for-loop. The latter has a time-complexity of $O(n)$ and $\Omega(n)$. Since this happens in all cases (because all n elements are always visited regardless of the array's structure), the time-complexity of the for-loop is $\Theta(n)$. the operations of creating the stack as well as the if-statements have a constant time-complexity of $O(1)$ and also $\Omega(n)$ and can therefore be ignored. The overall time-complexity of this algorithm is therefore both $\Theta(n)$ in the worst as well as in the best case, which makes it tightly bound.

SPACE COMPLEXITY. In the best case, the input array is of form [[,],(,),{,}, ...] (each opened character is immediately followed by the respective closed character). The latter form would make the stack at most of constant size $O(1)$ and accordingly $\Omega(1)$. In the worst case, the input array is of form [(, [, {, ...] (all n elements are opened characters) or [ ),], }, ...] (all n elements are closed characters). These configurations constitute to a worst-case space complexity of $O(n)$ and $\Omega(n)$ respectively since in those cases, the stack stores all n elements.