CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Homework #0

September 15, 2019

Name: FIRST LAST *Extension: Yes/No*

General Advice About Homeworks

- **Start early:** Difficult problems are not typically solved in one sitting. Start early and let the ideas come to you over the course of a few days.
- **Be rigorous:** A complete answer to each problem, unless specifically stated otherwise, consists of *three* parts: a description of the algorithm, a proof of correctness, and a running time analysis.
- **No pseudo-code**: English is the best way to express an algorithm; do not use pseudocode unless it is absolutely necessary.
- **Be concise:** Express your solutions at the proper level of detail, enough to clearly present all aspects of your solution, but not so many that the main ideas are obscured.

The following is a simple example of a homework and its solution following the guidelines above.

Problem 1. You are given an array A of n integers. Design an $O(n)$ time algorithm for finding the *largest* element in A, namely, $\max\{A[1], \ldots, A[n]\}$. **(20 points)**

Solution. A complete answer consists of *three* parts: a description of the algorithm, a proof of correctness, and a running time analysis. In the following, we give two slightly different solutions to this problem.

- **Solution one:** *Algorithm.* Iterate over elements of A in order $A[1], A[2], \ldots, A[n]$. Let $max \leftarrow A[1]$ initially and in each iteration i, if $A[i] > max$, then set $max \leftarrow A[i]$. At the end, return max.

  Proof of Correctness. We prove by induction that for every $i \in [n]$, the value of max at the end of iteration i is equal to $\max\{A[1], \ldots, A[i]\}$. This implies that for $i = n$ at the end of the algorithm, max is equal to the largest element of A as desired.

  The base case of induction for $i = 1$ is true by definition as $max = A[1]$. Suppose the induction hypothesis is true up until iteration i and we prove it for $i + 1$. By induction hypothesis, $max = \max\{A[1], \ldots, A[i]\}$ at the beginning of iteration $i + 1$. Since after this iteration, we have $max \leftarrow \max\{max, A[i + 1]\} = \max\{A[1], \ldots, A[i + 1]\}$, hence proving the induction step.

  Runtime Analysis. The algorithm iterates over all the n elements of A once, and each iteration takes $O(1)$ time for comparing the numbers and updating the variables, thus the total runtime is $O(n)$.

- **Solution two:** *Algorithm.* Consider the following recursive algorithm: if $n = 1$, return $A[1]$; otherwise, *recursively* compute maximum of $A[1], \ldots, A[n-1]$; return *maximum* number of this number and $A[n]$.

  Proof of Correctness. The proof is by inductively showing that the recursive algorithm finds the maximum of n numbers in A for all values of n. For $n = 1$, namely the base case, the correctness follows since the algorithm returns $A[1]$. Suppose the induction hypothesis is true for some $n = i$ and we prove it for $n = i + 1$. By induction hypothesis, the recursive call on an instance of size i returns $\max\{A[1], \ldots, A[i]\}$. Since the algorithm returns maximum of this number and $A[n] = A[i + 1]$, the returned answer is equal to $\max\{A[1], \ldots, A[i + 1]\}$, proving the induction step.

Runtime Analysis. Let $T(n)$ denote the runtime of the algorithm on an array of size n. We claim that:

$$T(n) = T(n-1) + O(1);$$
$$T(1) = O(1).$$

This is because the algorithm on an array of size n makes a single recursive call on an array of size $n-1$ and then spends $O(1)$ time to compute the final solution. The base case of reduction also follows from the definition of the algorithm. There are many different ways now to prove that $T(n) = O(n)$. For example, we can write:

$$\begin{aligned} T(n) &= T(n-1) + O(1) = T(n-2) + O(1) + O(1) = \cdots \\ &= T(n-i) + \sum_{j=1}^{i} O(1) = T(1) + \sum_{j=2}^{n} O(1) = O(1) + O(n) = O(n). \end{aligned}$$

The following two problems are extra. You do not need to turn in a solution for extra problems and they will not be graded. Throughout the course, we will typically have one extra problem of the type "Challenge Yourself" and may also have another one of the type "Fun With Algorithms".

The problems of the first type are usually (much) more challenging variants of the questions we study in the course. If you have no idea how to solve some of them, do not worry at all: some of them may really be challenging! You can always come to the Instructor's office hours and discuss your progress on these questions with the Instructor.

The problems of the second type are typically not necessarily more challenging than your regular homework problem (although sometimes they can still be quite challenging); however, these problems typically focus on aspects that are out of the scope of this course. For instance, in this "homework", the problem asks about a *non-asymptotic* analysis of *number of comparisons* done by an algorithm – throughout this course, we almost always only care about asymptotic analysis and usually (but not always) only care about runtime of the algorithms and not other measures.

Finally, as a general rule, only attempt to solve the extra problems if you enjoy them.

Challenge Yourself. You are given an array A of n integers and another integer $k < n$. Your goal is to output an array B containing the maximum number of *every k consecutive indices* of A, i.e.,

$$\max\{A[1], \ldots, A[k]\}, \max\{A[2], \ldots, A[k+1]\}, \max\{A[3], \ldots, A[k+2]\}, \ldots, \max\{A[n-k+1], \ldots, A[n]\}.$$

This problem can be solved easily in $O(nk)$ time using the algorithm in Problem 1. Design an algorithm that solves this problem in $O(n)$ time.

Hint. Use appropriate data structures such as queues and stacks. **(0 points)**

Fun with Algorithms. Once more, you are given an array A of n integers and your goal is now to design an algorithm that finds the *second largest* element of A. The algorithm in Problem 1 can be called twice to solve this problem in $O(n)$ time. However, suppose we now only care about measuring the number of *comparisons* made in the algorithm, i.e., the number of times we check which of given two numbers is larger.

The previous algorithm solve this problem with $n - 1 + n - 2 = 2n - 3$ comparisons. Design an algorithm for this problem that makes only $n + \lceil \log n \rceil$ comparisons (here log is in base two). For simplicity, you may assume n is a power of two, i.e., $n = 2^k$ for some $k \geq 1$ and hence $\log n = k$. **(0 points)**