CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 1

January 17, 2022

Instructor: Sepehr Assadi

1 Problems and Algorithms

- **Problem:** A *mapping* from the set of all potential inputs to the correct answer(s) for each input. For instance:
  1. Finding maximum:
     Input: any given array of numbers, Output: the largest element in the array;
  2. Sorting:
     Input: any given array of numbers, Output: a permutation of the array in non-decreasing order;

     ⋮
- **Algorithms:** A sequence of *simple* and *well-defined* steps for outputting the correct answer to *any* input of a given problem. For instance:
  1. An algorithm for finding maximum:
     (a) Pick the first element of the array as the candidate maximum.
     (b) Iterate over the elements one by one and if the current element is larger than the candidate maximum, choose this element as the new candidate maximum.
     (c) Output the candidate maximum at the end.
  2. NOT an algorithm for finding maximum:
     (a) Find the maximum of the array.
     (b) Output this number.
     (the first step is not simple enough for the purpose of this problem)

2 Algorithm Design

Our goal in this course is learning how to design *efficient* algorithms for different problems and how to analyze them rigorously. In the process of designing an algorithm, we should be able to find an answer to the following questions:

- **What** is the exact problem we aim to solve? (problem)
- **How** are we going to solve the problem or in other words how our algorithm works? (algorithm)
- **Why** is our algorithm correct for the problem we aim to solve? (proof of correctness)
- **How efficient** is our algorithm for the given problem? (runtime analysis)

Let us see these steps in action using the first problem mentioned above, i.e., finding maximum, as a running example.

Problem to solve. Here, the problem to solve is defined clearly already: given an array $A[1:n]$ of n numbers, find the value of the largest entry in A.

Algorithm. We can again just use the algorithm described earlier but written slightly more carefully.

1. Let MAX be a variable for the candidate maximum, set equal to $A[1]$ originally.
2. Iterate over the elements one by one and for $i = 1$ to n: if $A[i] > MAX$, set $MAX = A[i]$.
3. Output MAX at the end.

In this course, when specifying an algorithm, we use a mixture of normal text and simple pseudocode; the goal is *always* to maintain the highest level of clarity without going into too much detail; remember that we are designing an algorithm and not writing a code in any particular programming language.

Proof of correctness. We need get to perhaps the most important step of designing an algorithm, proving that actually it does what it is supposed to do; an algorithm without a proof of correctness is effectively useless because we have no guarantee that it actually solves the problem we want it to solve. A proof of correctness for an algorithm is basically the answer to the question that: *why this algorithm is correct?* the answer should then be mathematically accurate enough that can formally convince anyone (and yourself!) that the algorithm is going to work correctly on *every possible* input—we shall see many examples of proofs in this course and develop a better understanding of what constitutes a proof and what does not.

Let us now see a proof of correctness for this algorithm. To do so, we make the following mathematical claim:

Claim: For any $i \in \{1, \ldots, n\}$, at the end of iteration i, the variable MAX is equal to the largest entry of the subarray $A[1:i]$.

Proof. This is true for $i = 1$ because in step one of the algorithm, we set $MAX = A[1]$ and the largest entry of the array $A[1:1]$ is $A[1]$ (since $A[1]$ is the only entry of this subarray).

Now suppose that this claim is true for all integers $i \leq j$ and we prove it for $i = j+1$. Since we assume this statement is true up until $i = j$, we have that at the end of iteration j, MAX is equal to the largest entry of $A[1:j]$. During iteration $j+1$, we set MAX to be the larger of its previous value and $A[j+1]$; if largest entry of $A[1:j+1]$ is $A[j+1]$, then MAX would become equal to $A[j+1]$; otherwise MAX will not change value but then largest entry of $A[1:j]$ and $A[1:j+1]$ is the same. Thus, MAX would also be the largest entry of $A[1:j+1]$.

We can now conclude the correctness of the claim, because it is true for $i = 1$ and so we can apply the above argument to say it is true for $i = 2$ also; then again we apply this reasoning to say it is also true for $i = 3$, and so on and so forth until we get that it is true for $i = n$ as well[1]. $\square$

The correctness of the algorithm can now be inferred from this claim as follows. When $i = n$, i.e., after the last iteration, the above claim says that MAX is the largest entry of the entire array $A[1:n]$. As the algorithm outputs MAX at this point, we get that it outputs the largest entry, concluding its correctness.

Runtime analysis. Finally, we would like to also understand how "efficient" is our algorithm. Throughout this course, we primarily measure the efficiency of an algorithm by its *runtime* or roughly speaking how long it takes to run the algorithm as a measure of the input size. To analyze the runtime of our example, we first need a detour in to identify how exactly we measure the runtime. This is the topic of the next lecture.

[1]Let us mention that right here, we have been slightly informal. If you are familiar with *mathematical induction*, you notice that the argument we showed is simply a mathematical induction over the value of i and thus the last step should have been proven by induction, instead of a non-formal argument of applying it "so on and so forth". We will introduce induction and its role in the proof of correctness of algorithms in the next lectures and we can come back and formalize this argument completely.