Introduction, Worst Case Running Times[1]

Algorithms solve *computational problems*. Any computational problem has a clearly specified *input* which defines an *instance* of the problem. An algorithm is simply a *function* which maps *every* input of the problem to *solutions* or *outputs* for this problem. *Good* algorithms are correct and efficient.

Correctness. Every well-defined computational problem must have a *spec*, that is, a clear definition of desired input/output behavior. An algorithm is correct if for *every* input, the output/solution it returns satisfies the spec.

Let us take an example. Suppose my computation problem is the following.

Given an array, find the maximum element of the array.

The spec of the problem is as follows. The input is an array[2] $A[1:n]$. The output could be many things. For concreteness, we wish the output to be any *index* j with $1 \leq j \leq n$, such that $A[j] \geq A[i]$ for any $1 \leq i \leq n$. This is the spec of the problem. An algorithm is correct if for every input array $A[1:n]$ it returns a correct output. Note: there can be multiple correct answers if there are repetitions allowed in the array. The spec doesn't specify how to disambiguate, and so neither is the algorithm required to do so.

Remark: *It is an extremely good practice to write down the spec before writing any algorithm, or indeed, writing any piece of code. Like most good practices, it is easier to preach it than to practice it.*

How do we know if an algorithm is correct? It is not trivial and one need proofs[3] of correctness. Indeed often times (in this course) the design of the algorithm keeps correctness in mind – the proof of correctness will often follow from the way we designed it. Having said that, in this course, mainly since we have finite time, we will not spend too much energy in trying to formally prove correctness of algorithms.

Efficiency. In plain English, this is asking how "fast" is the algorithm given the "size" of the input. What is fast? What is size?

Every input of a computation problem is associated with a notion of *size*, a parameter which defines the magnitude of the input. This often needs to be spelled out (or will be implicitly assumed, but it is good to know what it is). For instance, in the illustrative example of finding the maximum entry of an array, the input is the array $A[1:n]$. What is the size of the array? At the most detailed level, it is the number of bits required to store the array in memory. This involves the number of bits for each element, the number of bits required to maintain an array, etc, etc. This number differs *for the same array* on different programming languages, and is thus hard to concretely define.

One of the first things one needs to learn in this course is to let go of the "weedy" details and focus on the "higher-order" stuff. Aka abstraction.

[1]*Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

[2]Most of my arrays are indexed from 1 unlike most programming languages.

[3]"Proof" is not a four-letter word. It is just a synonym for argument, justification, logical derivation. Proof doesn't necessarily mean Greek symbols and tons of math. Math is to be used when it helps. And it usually helps a lot.

What do we mean? Instead of bothering on how many bits precisely are needed to store the array, one asks given an array which parameter *really* affects the size. In most applications, it is probably the number of elements in the array. Think about it. If your array had integers in the range $[-10^{10}, +10^{10}]$, then each such integer maybe requires 35 bits to store. Maintaining the array may require 5 or so bits per element. But the number of elements, n, is the main driver of size. And so, in this particular example, it is not too bad an approximation to say that the *size* of the input is simply n. Or, as we will say soon, "order of" n.

Next, we come to the notion of *running time* of an algorithm. Again, one could measure the "wall-clock" time, but you can see that this is not well-defined. Whose machine are you running it on? What else is running on that machine? etc, etc. Just like size, one has to abstract out the notion of time when measuring the performance of algorithms, and not spend too much time (no pun intended) in these weeds. At least for the high level understanding of the algorithm.

One notion of time often used is the *number of elementary operations* that needs to be performed to solve the problem. What an elementary operation is needs to be specified, but is often clear from context. Again using the array example, one notion of elementary operation could be the number of *comparisons* one makes. Or it could be the number of *iterations* in a for-loop. It could be the number of *arithmetic operations*. The number of comparisons is the one that is usually used for this particular problem (but again, if your particular application needs something else, one may need to change this).

Once the notion of size and time are defined, we can be more precise. Formally, let's use Π to denote the computational problem, and let $\mathcal{I}$ be an instance of Π and let $|\mathcal{I}|$ denote the ***size*** of the instance. Given an algorithm $\mathcal{A}$ for Π, let $T_{\mathcal{A}}(\mathcal{I})$ denote the ***time*** taken by $\mathcal{A}$ to solve $\mathcal{I}$. Throughout, we will be using the *worst case running time* notion. Given a natural number $n > 0$, we define

$$T_{\mathcal{A}}(n) := \max_{\mathcal{I}\in\Pi:|\mathcal{I}|\leq n} T_{\mathcal{A}}(\mathcal{I})$$

In plain English, any instance of Π whose size is $\leq n$ can be solved by $\mathcal{A}$ in time $T_{\mathcal{A}}(n)$.

Again, let us take the "finding the maximum element in the array" example. One can probably define many algorithms. For instance, one could have an algorithm that compares *every pair* of elements in the array. In the worst case, one can see that the runtime of this algorithm $T_{\mathcal{A}}(n)$ would be roughly the number of pairs which is $\binom{n}{2} \approx n^2/2$. But hopefully you can devise an algorithm which makes only $(n-1)$ comparisons no matter what the array: the algorithm scans the array from left-to-right keeping a running max index (initialized to 1) which is replaced by j if $A[j]$ is bigger than the array element the current running max points to. A pseudocode description is below.

```
procedure Max($A[1:n]$):
    ▷ Returns $j$ where $A[j] \geq A[i]$ for all $1 \leq i \leq n$.
    rmax ← 1. ▷ Initialize running-max to 1
    for $2 \leq j \leq n$ do:
        if $A[j] > A[\text{rmax}]$ then: ▷ If $A[j]$ bigger than running-max, swap.
            rmax ← $j$
    return rmax.
```

✍

Exercise: *Given the above example, you can probably see how to find the maximum* and *the minimum in $(n-1)+(n-2)$ comparisons. Can you think of an algorithm which makes roughly $\frac{3n}{2}$ comparisons?*

In this course, you will learn some basic algorithm design principles. But, it takes creativity and ingenuity to *design* good algorithms. Like all creative forms, one gets better at it only by hours and hours of practice.

1 Supplement: Correctness of Procedure Max

To formally prove correctness of algorithms, one takes two broad approaches. If the algorithm is ***recursive***, then one uses *induction* to prove correctness. Here I will refer to these notes[4] to see how this is done.

All algorithms, however, are not necessarily written in a recursive fashion; like the Max algorithm above. To prove the correctness of this algorithm, the main idea is to introduce **loop invariants**. These are *assertions* which we will make for every loop such that after the loop is over, some other useful assertion is true. For the above algorithm, here I am writing it below with loop invariants.

```
1: procedure Max($A[1:n]$):
2:     ▷ Returns $j$ where $A[j] \geq A[i]$ for all $1 \leq i \leq n$.
3:     rmax ← 1. ▷ Initialize running-max to 1
4:     for $2 \leq j \leq n$ do:
5:         ▷ Pre: $A[\text{rmax}]$ is the maximum element in $A[1:j-1]$
6:         if $A[j] > A[\text{rmax}]$ then: ▷ If $A[j]$ bigger than running-max, swap.
7:             rmax ← $j$
8:         ▷ Post: $A[\text{rmax}]$ is the maximum element in $A[1:j]$
9:     return rmax.
```

The *Pre* invariant asserts that as soon as one enters the jth loop (the for-loop with index j), $A[\text{rmax}]$ stores a maximum element of $A[1:j-1]$. The *Post* invariant asserts that after running the jth loop, $A[\text{rmax}]$ stores a maximum element of $A[1:j]$. We begin with a crucial observation which is common to *Pre* and *Post* conditions for any for-loop correctness.

Observation 1. For any j, the *Post* condition for loop j is the *Pre* condition for loop $(j+1)$.

We now make a few claims to prove the algorithm's correctness.

Claim 1. If the *Pre* and *Post* invariants hold for all $2 \leq j \leq n$, then on termination $A[\text{rmax}]$ is a maximum element of $A[1:n]$.

Proof. Indeed, at the end of the last for-loop, $j = n$, and thus $A[\text{rmax}]$ will contain the maximum element of $A[1:n]$. And after the for-loop, rmax is not modified. □

So, we are now left with proving the *Pre* and *Post* conditions hold for all $2 \leq j \leq n$. This is done via (no surprises) induction.

Claim 2 (Base Case.)**.** At the beginning of the first for-loop (the case $j = 2$), *Pre* condition holds.

Proof. At the beginning of the first for-loop, rmax $= 1$ (this is set in Line 3). We therefore need to show $A[1]$ is a maximum element of $A[1:1]$. Indeed, there is only one element in $A[1:1]$. Nothing to show. □

Claim 3 (Inductive Case.)**.** Suppose the *Pre* condition is true *before* loop j, for any $2 \leq j \leq n$. Then the *Post* condition is true *after* loop j.

[4] https://www.cs.dartmouth.edu/~deepc/Courses/W20/lecs/lec8.pdf

Proof. Let $r_1 = \textsf{rmax}$ be the value of rmax at the beginning of loop j. Since *Pre* condition is true, we have $A[r_1] \geq A[i]$ for all $1 \leq i \leq j-1$. There are now only two cases possible:

Case 1. $A[r_1] \geq A[j]$. In this case, $A[r_1] \geq A[i]$ for all $1 \leq i \leq j$ as well. That is, $A[r_1]$ is a maximum element of $A[1:j]$. Indeed, in this case rmax doesn't change. Thus, *Post* is true after loop j.

Case 2. $A[j] > A[r_1]$. In this case, $A[j] > A[r_1] \geq A[i]$ for all $1 \leq i \leq j-1$. That is, $A[j]$ is the maximum (note: this is actually at this point the unique maximum...but that is beside the point) of the array $A[1:j]$. Indeed, in this case rmax is changed to j. And thus, even in this case *Post* is true after loop j. □

The above three claims gives us the following theorem which formally asserts the correctness of MAX.

Theorem 1. The algorithm MAX returns index of a maximum element for any array $A[1:n]$.

Remark:

- *As you can see, writing the formal proof can be a bit tedious. Important, but a bit tedious.*
- *Coming up with a loop invariant takes some practice; due to time-constraints, we will not focus too much on proving correctness of algorithms. But hopefully the above example gave a flavor.*

2 Supplement: A "Lower Bound"

Theorem 2. Any correct algorithm for finding the maximum of an n element array must make $(n-1)$ comparisons among the array elements.

Remark: *One should be careful about what the above theorem and the forthcoming proof is really saying. The "time" is calculated as the number of comparisons made between elements of the array. So, if for some reason, your algorithm compares $A[1] + A[2]$ with $A[3] + A[4]$, which elements are being compared? The proof below will assume that there are* four *comparisons being made:* $(A[1], A[3])$, $(A[1], A[4])$, *and* $(A[2], A[3])$, $(A[2], A[4])$. *More formally, it is a theorem in the "comparison model" where the array items are assumed to be immutable objects which cannot be arithmetically manipulated, but only compared against.*

Proof. Any algorithm in the comparison model runs as follows: at any step it makes a comparison between $A[i]$ and $A[j]$ for some two indices, and given the result of this comparison, it makes some calculations, and then makes the next comparison. Fix any array $A[1:n]$ and consider the run of the algorithm. As the algorithm runs, construct the following graph $G = (V, E)$. The vertices $V = \{1, 2, \ldots, n\}$ correspond to the *indices* of the array. Every time the array compares $A[i]$ and $A[j]$, add the edge (i, j) to the set E. After the algorithm completes, and returns an answer, say $A[k]$, our graph is fully defined. If the algorithm makes $< (n-1)$ comparisons, then this graph has $< (n-1)$ edges.

To illustrate this, Figure 1 below shows what the graph is for the MAX algorithm we in the lecture notes, on the array $[20, -3, 14, 39, 7, 16]$.

Now, the graph G has n vertices but $< (n-1)$ edges. This implies that the graph G is *not connected*. This spells trouble for the algorithm. The algorithm has decided to return the number $A[k]$. Let U be the

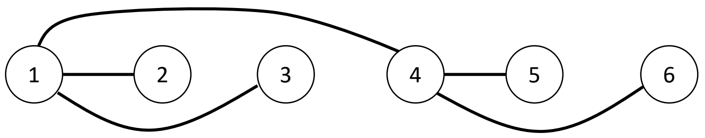

Figure 1: The graph obtained upon running MAX on $[20, -3, 14, 39, 7, 16]$. First $(20, -3)$ are compared, then $(20, 14)$, and then $(20, 39)$. At this point, we move running max to 39, and then compare $(39, 7)$ and $(39, 16)$.

set of vertices v which are connected to k, that is, have a path in G from v to k. Since G is not connected, there is some vertex ℓ which is *not* connected to k. Let X be the set of all these vertices. Note that there is *no* edge (i, j) in G with $i \in X$ and $j \notin X$.

Here is the kicker. Consider an array $B[1 : n]$ which is defined as: $B[i] = A[i] + M$ for $i \in X$, and $B[i] = A[i]$ otherwise, where M is a large enough integer such that $B[k]$ (which is $A[k]$) is *not* the maximum in $B[1 : n]$. On the other hand, note that the *answers* to the comparisons made by the algorithm is *exactly the same* in both B and A. This is because, whenever the algorithm is comparing i and j, these two either both lie in X (in which case the answer in B as the same as in A), or both outside X (in which case B is the same as A). Why can't it be that $i \in X$ and $j \notin X$? Because, then (i, j) would be an edge in G, and such edges don't exist.

Since the comparison results are the same in both A and B, the answers given by the algorithm also must be the same. (Here we are using the comparison model; the algorithm is not using any "external data" to make its decision.) So, the algorithm would again answer $B[k]$, and thus would be wrong.

$\square$