CS31 (Algorithms), Spring 2020 : Lecture 0

Topic: Introduction, Worst Case Running Time

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please discuss in Piazza/email errors to deeparnab@dartmouth.edu

---

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[1] $A[1:n]$. 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[2] 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, 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] Most of my arrays are indexed from 1 unlike most programming languages.

[2] "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 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 now you can quickly see 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 the notion of time when measuring the performance of algorithms, and not spend too much time in these weeds. At least for the high level understanding of the algorithm.

The notion of time is often the *number of elementary operations* that needs to be performed to solve the problem. What is an elementary operation often 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 see 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.

```
1: procedure MAX(A[1 : n]):
2:     ▷ Returns A[j] where A[j] ≥ A[i] for all 1 ≤ i ≤ n.
3:     rmax ← 1. ▷ Initialize running-max to 1
4:     for 2 ≤ j ≤ n do:
5:         if A[j] > A[rmax] then: ▷ If A[j] bigger than running-max, swap.
6:             rmax ← j
7:     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.