CS 3510 Algorithms Last updated: August 26, 2023

Lecture 1: Big O and Bubble Sort

Lecturer: Sahil Singla *Scribe(s): Joseph Gulian, Allen Chang*

Disclaimer: These notes have not been carefully verified and might contain mistakes.

1 Introduction and Syllabus

This course is focused on the development and understanding of algorithms. There are four main topics in this course that are divide and conquer, dynamic programming, graph theory, and complexity theory.

Most of the information for this course can be found on the Canvas page or the website. You can find the official schedule on the website, you can ask questions on Piazza. You'll be expected to upload homework to Gradescope; homework must be typed, and latex is preferred. There are no official textbooks for this course although if you want a separate perspective on the topics covered, materials may be listed for a given lecture. There are several textbooks in the course information sheet. There is no official textbook for the class, but Dasgupta and other books are recommended on the website.

2 Algorithms

Although it's not a formal definition, an algorithm could be thought of as a sequence of instruction to perform some task. Although often, the development of an algorithm is not the only interest: we are often interested in minimizing some resource like space time or communication for analyzing an algorithm. [1] If you think about an algorithm like a recipe, like an algorithm there are many different ways to make the same dish. Sometimes we need to cook fast, but other times we may want to limit the number of dishes use. Other times, there might be multiple cooks, and we want them to work individually without waiting on each other, so we value little communication between them.

The primary course is to deal with these two facets of studying algorithms: their design and their analysis. You will learn to develop algorithms to solve problems, but you will often be expected to prove the algorithms correctness or runtime as well. Hence, the nature of this class is theoretic, so we will not be actually programming algorithms we discuss, simply describing them, and arguing about them. [2]

[1] Although we often care about time, space is useful in situations too, like on some embedded or IOT device where memory is limited. Communication is useful in the development of distributed algorithms wherein we value little communication between processes in a network.

[2] Although this class is theoretic, it's worth noting that a lot of these algorithms have been influenced and actively influence other fields like Economics, Math, and Physics. Nature happens to be very efficient, so there is a lot to learn from.

3 Growth of Functions

Suppose you have some function $f(n)$ which gives the amount of resources (i.e. # of instructions, space, or messages needed) with respect to the input size. [3] Although we might be interested in understanding the amount of resources at one point, we're often more interested in understanding how the resource needs change as the input grows. For instance, if you have the functions $f(n) = n$, $g(n) = n^2$, and $h(n) = 2^n$, the changes from 10 to 100 are drastic.

It's often easier for us to describe these functions by what classes they're members of. [4] The main way we'll describe a class is through some Big $\mathcal{O}$ notation. We'll first develop an understanding of what $\mathcal{O}$ means and then discuss various alternatives.

If we say for some functions $f(n) = \mathcal{O}(g(n))$, this means there exists constants c, x, such that $f(n) \leq c \cdot g(n)$ for $n \geq x$, or more intuitively, after some point, the function f is always less than the function g multiplied by some constant. With this definition, we understand that $n^2 + 2n$ and n^2 are both in the set of $\mathcal{O}(n^2)$. Notice though that they are also in the class $\mathcal{O}(n^3)$.

Now that we have developed the idea of $\mathcal{O}$, let us talk about Ω which you could think of as the opposite: it provides an asymptotic lower bound for a function. Like above, the functions $n^2 + 2n$ and n^2 are both in the class $\Omega(n^2)$, but these functions are not in $\Omega(n^3)$ because that is larger after some point. Then the formal definition for Ω is $f(n) = \Omega(g(n))$ if there exists some c, x where

$$0 \leq c \cdot g(n) \leq f(n)$$

for $n \geq x$ Also it turns out that we see here, that

$$f(n) = \Omega(g(n)) \iff g(n) = \mathcal{O}(f(n))$$

If a function is upper-bounded and lower-bounded, then the notation for Θ can apply. We say functions $f(n) = \Theta(g(n))$ if there exists constants c_1, c_2, s such that

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Another way to thing about this is that it's the combination of $\mathcal{O}$ and Ω, so

$$\Theta(g(n)) \iff f(n) = \mathcal{O}(g(n)) \;\;\&\;\; g(n) = \mathcal{O}(f(n))$$

There are two more notations worth mentioning known as o and ω; both have similarities to their capital counterparts but are different in one way: they're strictly not asymptotically tight. What does this mean? Well, if you think about the two functions n and n^2, while both are $\mathcal{O}(n^2)$, only n^2 is "tight" with $\mathcal{O}(n^2)$. [5] Again o is not tight, so $n = o(n^2)$ but $n^2 \neq o(n^2)$. Formally $f(n) = o(g(n))$ if there exists some constants c, x where $f(n) < c \cdot g(n)$ for all $n \geq x$.

[3]Students sometimes get confused about the meaning of n. Often n is the length of some input to the algorithm. What if the algorithm takes a number? Consider n to be the length in bits needed to store the number unless otherwise specified.

[4]For instance, it's easier to suss out someone's interests by what threads they are.

[5]You could think of this in a limit sense in that the limit of n/n^2 will go to zero, but the limit of n^2/n^2 will not go to zero.

To round out our notation, the last element of our notation is ω which denotes a non-tight lower bound. Formally $f(n) = \omega(g(n))$ if there exists some constants c, x where $c \cdot g(n) < f(n)$ for all $n \geq x$.

You could think of these as sets of functions, and there's much more we could talk about, but most of these tools are going to be used to discuss how algorithms use resources. For instance in a previous class, you may have heard that some algorithm has a worst case runtime of $\mathcal{O}(n^2)$. These other notations give us ways to compare algorithms to other algorithms. For instance, maybe we want to argue that some group of sorting algorithms would in the worst case require all possible matches, this would imply that the number of comparisons would be $\Omega(n \log n)$.

4 Proving Notation

Before stopping our discussion of notation, we'll do a proof that some functions are $\log(n!) = \Theta(n \log n)$. Before we perform We'll split this proof up into two parts, the first part is that $n!iso(n^n)$, and the second part is actually that Looking at the equations above we see that to prove $\log(n!) = \Theta(n \log n)$ we must show that $\log(n!) = \mathcal{O}(n \log n)$ and $\log(n!) = \Omega(n \log n)$ However we also have the transformation that

$$f(n) = \Omega(g(n)) \Longleftrightarrow g(n) = \mathcal{O}(f(n))$$

meaning that we can prove

$$\log(n!) = \mathcal{O}(n \log n)$$

To prove the first statement, we know that $n! \leq n^n$ because $1 \cdot 2 \cdot \ldots \cdot n \leq n \cdot n \cdots n$. To prove the second statement, we need a little more math. We know that $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$, and taking the log of that would give $\log(n!) = \log 1 + \log 2 + \log 3 + \ldots + \log n$. You'll see why we do this later, but for now let's make explicit the $\log \frac{n}{2}$ and assert that the series from $\log 1$ to $\log n$ is greater than the series from $\log \frac{n}{2}$ to $\log n$. We'll write

$$\log n! \geq \log \frac{n}{2} + \ldots + \log n$$

We're going to take another step on the right hand side and replace all the terms with the smallest term $\log \frac{n}{2}$, which becomes

$$\log n! \geq \log \frac{n}{2} + \ldots + \frac{n}{2}$$

Now here's the beauty, we know that there are $\frac{n}{2}$ terms in the right hand expression meaning the entire term is $\frac{n}{2} \log \frac{n}{2}$. By this point, you can do a bit more simplification to get to exactly $n \log n$, but this is reasonably close for lecture.

5 Sorting

Given n numbers $a[0], \ldots, a[n-1]$, we want to rearrange them into non-descending orders such that $a[0] \leq a[1] \leq \cdots \leq a[n-1]$. Now there are two steps to developing algorithms, the first part is to understand that the algorithm is correct, and does what it is supposed to do. The second step is to check your algorithm against constraints like time, for our examples, operations like compare, swap, and creating storage take unit time.

One example is Bubble Sort. The algorithm goes as follows:

Algorithm 1 Bubble Sort

```
for $i = 0$ to $n - 1$ do
    for $j = 0$ to $n - i - 1$ do
        if $a[j] > a[j + 1]$ then
            swap $a[j], a[j + 1]$
        end if
    end for
end for
```

We prove the correctness by induction where the induction statement is that after i steps of the outer loop, the largest i elements will be sorted in the last i locations.

Analysis: The number of steps we run at each stage of the inner loop is n, $n-1$, $n-2$, and then to 2, and 1. We can take the sum of these which we find to be $\frac{n(n+1)}{2} = \mathcal{O}(n^2)$. In the future lectures, we'll see that this is not good when we see better algorithms like merge sort.