Lecture 5: Algorithm Analysis

Starting with some History

Al Khwarizmi (780 AD - 860 AD)

- Persian Mathematician working in Baghdad the capital of the Abbasid Empire
    - Baghdad was like the New York of 820.
- One of things he did was promote - successfully - the 10-digit place value system (the numbers we use today), which came from India. Those are the Arabic Numerals, but they weren't Arabic Yet.
    - In my modern view: He was promoting DECIMAL and more specifically PLACE VALUE as an INFORMATION TECHNOLOGY; something that's going to make us manipulate information more efficiently.
- To this end, he wrote down, STEP by STEP all the things that you can do with these new numbers. Here's how you do addition. Here's how you do multiplication and so on. In some ways, it was like code!
    - Documented in "Book of Indian Computation" (820)
    - Cited so much that the latinized version of his name became synonymous with these computational instructions: algorithm
- Eventually these were brought to Europe by **Fibonacci** writing for merchants / business.
    - You're going to have a return on investment by using this number system.
    - You're even going to be able to calculate return on investment! (Imagine using Roman numerals)

So fast forward to the 20th century, and we have data and computation at scale.

- So, not only do we need to know how long something takes in raw terms, we need to know how it scales.

The textbook that popularized this is "Art of Computer Programming" (1968)

- **Donald Knuth** - Another name that comes up a lot in CS. B. 1938 - currently prof emeritus at Stanford
- Promoted so-called "Big-Oh notation"
    - The idea is that we have some function of how our running time (or alternatively memory used) increases as the input size increases.
        - What is input size? Could be the actual number - like in your fibonacci program, or usually it could be the more literal size - like if your input is the array, it's the size of the array.
    - But we don't just want to think about all these complicated functions - like you have in your math class - we want to distill it down to what's important.

So first of all, we have a few questions to ask when we ask what we're measuring:

1) Number of operations vs memory allocated (usually number of operations unless you have a memory-hungry application)
2) Average Case, Worst Case, Best Case?

a) Worst Case: Adversarial input-giver. A real jerk.

Let's look at some examples:

Some code takes a fixed amount of steps in order to solve a problem, no matter what the input.

- Examples from yesterday (can go fast!)
- Example: Mathematical function
- Example: If/Then Blocks
- Example: grabbing things from memory, following references

These are called **constant-time algorithms**, or O(1).

- We don't care what the constant is! But the idea is that no matter how big your input is, the algorithm still takes the same amount of time to run.
- For example, the length of the array is stored in the header. So that's constant.
    - Summing the values of an array - well that's going to take more time the larger your array is.

A counter example: summing an array, counting to n

- This is a linear time algorithm.
- The larger the n, the more steps it takes, and it increases IN PROPORTION to the input n
    - Double n, or double the array size, double the number of computations you need
- Same would be for looking at an array and printing out all the items.
    - Also, deleting a random item from the array (because the values after it need to be recopied)
    - Adding a random item to an array
- We call this linear-time algorithms, or O(n)
- Note that if we go through the loop several times (NOT loop-within-loop, but one loop finishes and another begins), it's still O(n) - because it still increases in proportion to n

Here's a third example:

- Let's write a power function, a^n, and look at an algorithm
- Note that we recursively call it on n/2
- So instead of counting down from n, we're dividing it in half each time.
- NOW IF N DOUBLES, DOES THE RUNNING TIME DOUBLE AS WELL?
    - No - it just goes one layer deeper
- How many layers down?
    - Approximately log(n) - log base 2
    - We call these log time algorithms, or O(log(n))

```
public static double raiseToPower(double base, int exponent) {
   // Handle base case for exponent 0
   if (exponent == 0) {
      return 1;
```

```
        }

        // Handle negative exponent
        if (exponent < 0) {
            base = 1 / base;
            exponent = -exponent;
        }

        // Recursively raise base to exponent / 2
        double halfPower = raiseToPower(base, exponent / 2);

        // If exponent is even, the result is halfPower * halfPower
        // If exponent is odd, multiply an extra base
        if (exponent % 2 == 0) {
            return halfPower * halfPower;
        } else {
            return base * halfPower * halfPower;
        }
    }
```

Alright - so now that we've seen some examples, let's take a look at the theory of big-O

- Stands for "on the order of"

Mathematical definition
f(n) is O(g(n)) if lim (n -> infinity) f(n) / g(n) < some constant c. (or equivalently is < +inf, or is bounded)

BUT - what you need to know:

- Any polynomial -> O(n^k) - just look at the highest power of the polynomial!

Let's see how this works in practice in our 3 examples:

If f is O(1) then f(n) itself < some constant c. So that makes sense, it's bounded

If f is O(n) then f(n) /n < c in the long run, or f(n) < c*n

- Note that this technically fits constant algorithms as well, but **by convention we always take the running time in the simplest and lowest form.**
- There are other greek letters that represent different inequalities, but they are rarely used

If f is O(log(n)), then f(n) < c*log(n) for some c.

2^n algorithm: EXPONENTIAL TIME

- Example - list all the configurations of n bits. The code is simple, but you are now counting up to 2^n so it will take a long time
- Recursive Fibonacci
- In general, O(3^n) is not O(2^n); it depends on the base, but all of these just go under the heading of “exponential time”

Some rules of thumb when thinking about algorithmic complexity

Most basic operations constant time:

- Addition, multiplication, following a reference in an object, array operations

O(f(n) + g(n)) = O(max(f(n), g(n))

- Practically, this means that if you have two separate sections of code in sequence, always look at the more expensive one.
- if/then - look at the condition (usually O(1)), and then look at the worst case in the THEN and the ELSE

A loop is a multiplier of what’s inside the loop.

- Arithmetic progression to n: then it’s n times the inner loop
- Geometric progression to n: then it’s log(n) times the inner loop

Putting it together:

- Double counting loop: O(n^2)
- Triangular double loop: Also O(n^2)
    - SUM: triangular number formula
    - 1 + 2 + 3 + 4 + … + n = n (n + 1) / 2
    - Also look at the geometric argument

Integer Element Uniqueness: 3 examples of how to tell if every item in an array is unique

a) Double loop - every element with every other element
b) Sort and check.
    i) The check is linear time
    ii) The sort is O(n*log(n)) - trust me!
        1) A little note on log(n). While O(log(n)) is a more than O(1) and O(n log(n)) is more that O(n), in practice log(n) feels like a constant.
c) Keep an array of booleans: what have I seen before?
    i) Can be memory-hungry, but O(n)

Amortized

(Move to linked list lecture)
Appending an item to an array that could have excess capacity

- If we add to the size every time we need to increase capacity, then its quadratic time (triangular sum)

- If we double the size of the array every time, then we average linear time (geometric sum)
    - LOOK AT GEOMETRIC SUM