Complexity Analysis: Big O Notation

Algorithm Efficiency

- In this class we're focused on comparing multiple solutions to a problem
- How do we know which to choose?
- Lab 2 showed an “experimentalist” approach
- Today we'll look at the “theoretician” approach that we'll follow for most of the course, which we'll reconcile with experimental results

Example: Looking up a word in the dictionary

- Algorithm 1:
  - start on page 1
  - look at the page, if the word's on that page, return the definition
  - advance to the next page

Algorithm 1

- Is this a “correct” algorithm? (Not quite... unknown word)
- Is it “Efficient?” Seems like no, but can we be precise?

Analysis of Algorithm 1

```
for page 1 to dictionary length:
    if word is this page: return the definition
return not found
```

- To analyze: count (estimate) the number of CPU instructions executed by the algorithm
- What does that number depend on?

Algorithm 2:

- Open the dictionary in the middle
  - Is the word on that page? If yes, return the definitiion
  - If the word will be after this page, rip out this page and all previous pages and start over on the upper half
  - Otherwise, throw away the upper half and search the lower half

Algorithm 2 Analysis:

- Is this algorithm correct?
- What is the estimated number of CPU instructions needed?
- What does it depend on?

Comparison:

- Which algorithm is “better?”
- Can we quantify this in a simple way?
- Counting individual instructions is hard and not useful
- Not all CPU instructions take equal time (Can vary by ~100x!)
- Usually we compare methods with VERY different characteristics so that level of detail isn't helpful

Big O Notation:

- A super mathy definition for a simple idea: throw away all the messy details
- Big scary math definition: f(x) is in the set of functions O(g(x)) if for some constants a, b > 0, f(x) <= a*g(x) for all x > b
- $f(x) \in O(g(x))$ if $f(x) \leq a * g(x)$ for all $x \geq b$
- When we talk about it, we should say “f(x) is in big O of g(x)" but often just say “f(x) is O of g(x)"
- What's the point?

Big O Examples:

- f(x) = $10.5x^2 - 3.2x + 1532.4$ — lots of messy details
- f(x) is in O(x^2) because we can pick our constant "a" to be something smaller than 10.5 and b to be "big enough" that the −3.2x term doesnt' matter
- We usually only care what the "leading term" is: x^2 in this case and can ignore any constant factors, or smaller terms
- Since we know ahead of time what we care about, we won't bother calculating those constant factors or low order terms!

Quiz: Is $f(N) = 3N^2 + 10N + 1000$ in:

- $O(1)$?
- $O(N)$?
- $O(N^2)$?
- $O(N^3)$?
- O(2^N)?

Not helpful: Most functions are in O(2^n) !!

- Big O gives us an upper bound which is useful, but you can be overly pessimistic and still technically correct
- Sometimes we need an upper bound (basically one function is >= another instead of <=)
- $f(x) \in \Omega(g(x))$ if $f(x) > a * g(x)$ for some a, b > 0
- This is read as “big Omega”
- If a function is upper bounded and lower bounded by the same function, we use “big Theta” which means:
- $f(x) \in \Theta(g(x))$ if $f(x) \in O(g(x)$ and $f(x) \in \Omega(g(x))$

In practice:

- Usually we care most about a “good” upper bound, so we usually use “Big O” notation
- Usually we just say “O of whatever” when we mean “big o” since it's most common
- It's a good idea to be precise. Knowing that something is big Omega vs big Theta vs big O can actually be pretty important!

Determining algorithm complexity

- Usually we are interested in the running time of an algorithm as a function of the input size, which we usually call N
- The general strategy is to try to figure out the big O cost of each line of our algorithm
- If something is inside of a loop, we multiply the cost of the loop body times the number of times the loop runs
- Since we're only interested in Big O bounds, we can ignore any constant factors

Common patterns:

- If we loop through an entire array, that means we run it's body N times, so our complexity is O(N * loopBodyCost)
- Nested loops through an array would mean our algorithm is $O(N^2)$
- What about a loop pattern like :

```
for i in 0..N:
   for j in 0 .. i:
       someConstantTimeOperation
```

Imporant, slightly trickier pattern:

```
int binarySearch(arr, start, stop, target):
    middle = (stop + start)/2
    while(start < stop){
        if arr[middle] == target:
            return middle
        if arr[middle] < target:
            start = middle + 1
        else:
            stop middle - 1
```

Functions to know:

- O(1)
- O(Lg(N))
- O(Sqrt(N))
- O(N)
- $O(N^2)$
- $O(N^3)$
- O(2^N)
- O(N!)

Analysis Tricks:

- Repeated halving → Probably Lg(N)
- For some complex nested loops, sometimes it's usefull to consider the loops together rather than separately and multiplying
- N^2 algorithms are a lot more common than 2^N algorithms
- It's helpful to think about what happens if you make the input bigger by 1. How does the running time change?