Asymptotic Analysis

Anthony Ding, CS 61B

Why does runtime matter?

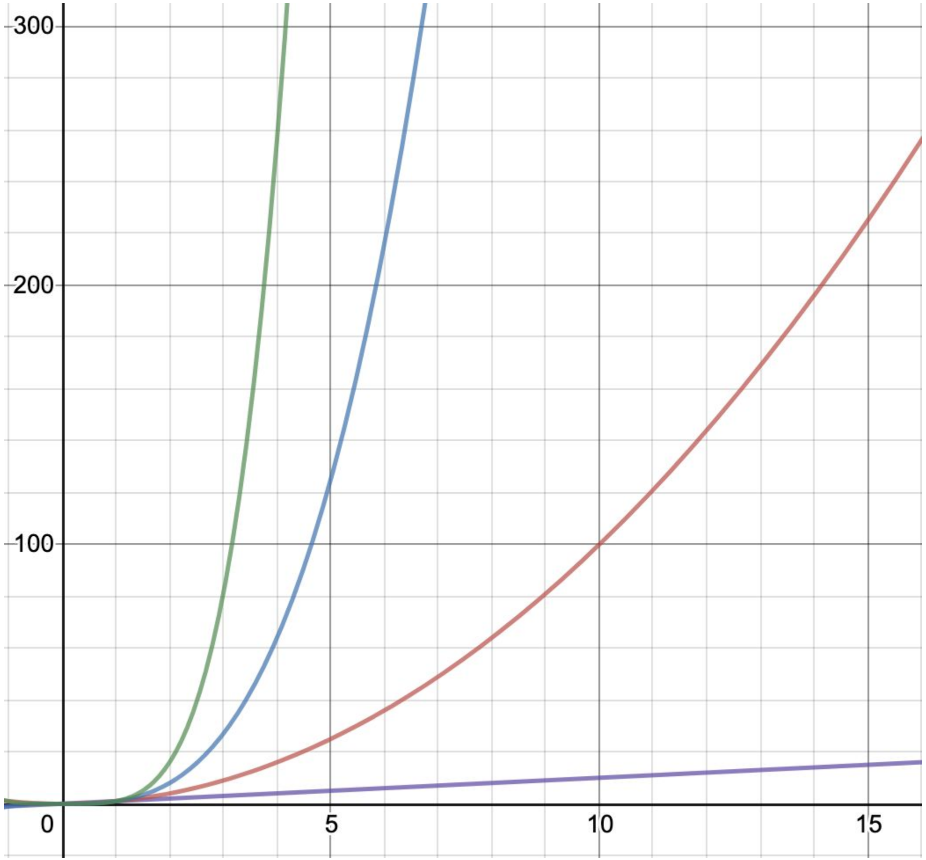
- Two programs
    - Program A: given x inputs, returns a result in x milliseconds
    - Program B: given x inputs, returns a result in x^2 milliseconds
- If we run both A and B on an array of size 1000 (number of inputs, x = 1000):
    - Program A will take roughly 1 second
    - Program B will take 1000 seconds = roughly 17 min!
- In the real world, we care about **correctness**, but also **efficiency**

Informal Approaches

- Physically measuring with a stopwatch (can start your phone's stopwatch as soon as you press run and stop it)
  - Not very precise
- `java.lang.System.currentTimeMillis()` gives us a precise reference in milliseconds
- Drawbacks:
  - Different programs can run faster or slower on different computers
  - If Program A on my CPU runs slower than Program B on yours, unclear if my program is simply worse, or if your computer is just faster
- We would like an abstract way of determining and modeling runtime

O(n), Θ(n), Ω(n)

- Measure orders of growth in runtime, given an input n
- Abstractly, how does the runtime, as a function, grow as input size n gets bigger?
- $O(n)$ = upper bound
- $\Omega(n)$ = lower bound
- $\Theta(n)$ = combination of O and Ω, both upper and lower bounded



Green (x^4), Blue (x^3), Red (x^2), Purple (x)

Mathematical Definition

- "For large n" is needed because at small values, even 2n can be greater than n^2 ($n=1$)
  - This isn't useful to us, as both will take minimal time to run
- Given two runtime functions $f(n)$ and $g(n)$, we say $f = O(g)$ if for large n, $f(n) <= g(n)$
- We say $f = \Omega(g)$ if for large n, $f(n) >= g(n)$
- $f = \Theta(g)$ when both O and Ω are satisfied
  - This means both f and g grow at roughly the same rate
  - As input n grows large, the derivatives of f and g (rate of growth) are roughly same

Solving for Runtime

- Drop constants and coefficients ($3x^2 + 4x$ can be simplified to $x^2 + x$)
- Any exponential dominates any polynomial (2^x grows faster than $1000000 * x^5$)
- Any polynomial dominates any logarithm
- Higher order polynomials beat lower order ones (x^3 beats x^2, x^2 beats x, and so on)
- Therefore, we can rewrite any sum of terms by the term that dominates
    - $x^5 + x^3 + \log x$ can be simplified to x^5

Example #1

```
for (int i = 0; i < N; i += 1) {
    doSomething();
}
```

Assuming that doSomething() takes constant time, it is called exactly N times, whether N is 1 or a million. We can say this is $\Theta(n)$

Example #2

```
for (int i = 0; i < N; i += 1) {

    for (int j = 0; j < i; j += 1) {

        doSomethingElse();

    }

}
```

At each pass through the outer loop, we do a total of i calls to doSomethingElse(). This results in the sum 1 + 2 + 3 + … + N = N(N+1)/2.

Now, we can simplify this expression to solve for our runtime. N(N+1)/2 = 0.5N^2 + 0.5N. Dropping constants, we get N^2 + N, which simplifies to N^2.

Example #3

```
public void foo(int bar) {

    return 2 * foo (bar / 2);

}
```

Think of it as a tree. At each level, we have a total of bar. We split this into bar/2 but call it twice, so the level underneath will also have bar.

Every level up until the leaves will have bar. There are a total of log(bar) levels.

Therefore, the runtime is O(bar log bar) [we usually just say it is O(nlogn)]