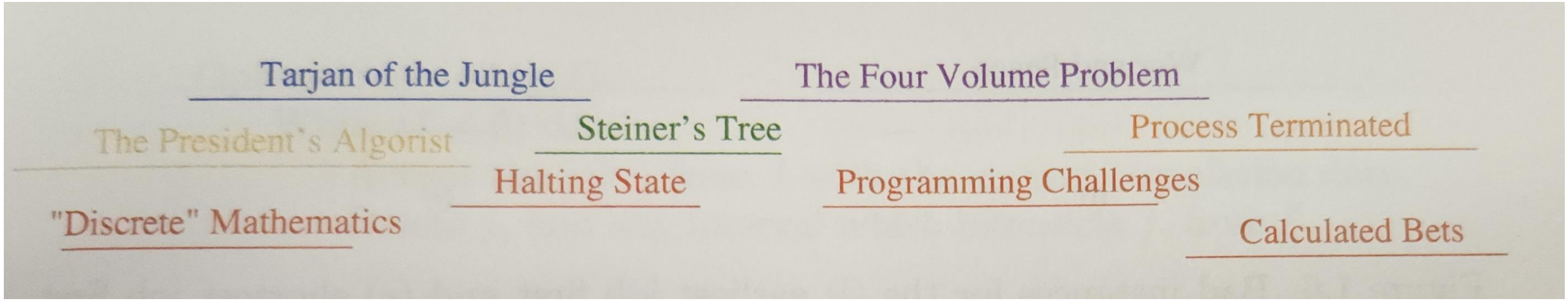
Discrete Math (v.2) & Asymptotics

CS 374

Imagine we are actors

We could schedule to work in the following movies:

- Goal: work in the most movies possible.
- Displayed as a set of *intervals*



Option 1:

- EarliestJobFirst:
  - Accept the earliest-starting job *j* that does not overlap any previously accepted job. Repeat until no more such jobs remain.

Option 2:

- ShortestJobFirst:
  - While the set of movies remains nonempty,
    - Accept the shortest possible job *j*.
    - Delete *j* from the list, as well as any other job that intersects with *j*.

Option 3:

- ExhaustiveScheduling:
  - Test all 2^n subsets of intervals from the set of movies, and return the largest subset consisting of mutually non-overlapping intervals.

Option 4:

- OptimalScheduling:
  - While the set of movies is not empty:
    - Accept the job *j* with the earliest completion date.
    - Then delete *j* and any interval which intersects *j* from the set.

Option 5:

- GreedyScheduling:
  - Select the job *j* which overlaps the smallest number of other jobs.
  - Remove *j* and repeat until no jobs remain.

Which option is the best?

- EarliestJobFirst:
  - Accept the earliest-starting job *j* that does not overlap any previously accepted job. Repeat until no more such jobs remain.
- ShortestJobFirst:
  - While the set of movies remains nonempty,
    - Accept the shortest possible job *j*.
    - Delete *j* from the list, as well as any other job that intersects with *j*.
- ExhaustiveScheduling:
  - Test all 2^n subsets of intervals from the set of movies, and return the largest subset consisting of mutually non-overlapping intervals.
- OptimalScheduling:
  - While the set of movies is not empty:
    - Accept the job *j* with the earliest completion date.
    - Then delete *j* and any interval which intersects *j* from the set.
- GreedyScheduling:
  - Select the job *j* which overlaps the smallest number of other jobs.
  - Remove *j* and repeat until no jobs remain.

Tarjan of the Jungle

The Four Volume Problem

The President's Algorist

Steiner's Tree

Process Terminated

Halting State

Programming Challenges

"Discrete" Mathematics

Calculated Bets

Present the following…

- Is it correct? Will it produce the longest possible list of jobs?
  - If it is incorrect, provide a counterexample.
- Is it efficient?

- Rate each of these categories out of 10 (1=worst, 10=best)

Induction, attempt 2

Is this correct?

```
Increment(y)
    if (y = 0) then return(1) else
        if (y mod 2) = 1 then
            return(2 · Increment(⌊y/2⌋))
        else return(y + 1)
```

- Prove with induction!

Is this correct?

```
Increment(y)
    if (y = 0) then return(1) else
        if (y mod 2) = 1 then
            return(2 · Increment(⌊y/2⌋))
        else return(y + 1)
```

Extra slide if needed

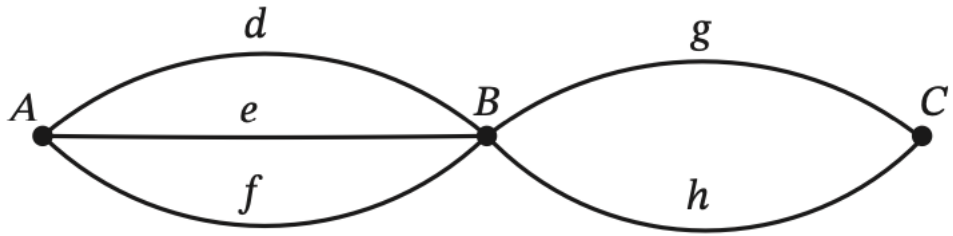
```
Increment(y)
    if (y = 0) then return(1) else
        if (y mod 2) = 1 then
            return(2 · Increment(⌊y/2⌋))
        else return(y + 1)
```

Combinatorics Cheat Sheet

Why do we need them?

- Useful for knowing how many possible options there are.
- Helpful for assessing efficiency options

Cheat Sheet



- **Addition Rule**:
  - If there are several events/decisions that happen, but are only available in separate "universes", add the number of possibilities together.
  - E.G. I could take one path xor another (impossible to take both). 3 + 2 paths total
- **Multiplication Rule**:
  - If events must happen in sequence
  - E.G. I have to choose out of 3, then out of 2. 3*2.
  - E.G. choosing students to stand in a line
- **Factorial:**
  - n! = n(n-1)(n-2)(n-3)...(2)(1)

Permutation, Combination

- **Combination:**
  - "n choose k" nCk = $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
  - Choosing k elements from a set of n things
  - Without replacement
  - Order chosen *does not* matter.
- **Permutation:**
  - "n permute k" ${}_nP_k = \frac{n!}{(n-k)!}$
  - Choosing k elements from a set of n things
  - Without replacement
  - Order chosen *does* matter.

Example Problem

- A hungry group of professional Settlers of Catan players stop at Loard's Ice Cream on the way to their next tournament. Their coach plans to buy each team member a cone. The shop is stocked with 16 flavors. For cones, having chocolate ice cream as the bottom scoop is different from having chocolate as the top scoop. For bowls, it doesn't matter in what order the scoops are added.
  - If a flavor cannot be chosen more than once, how many possible 3-scoop cones could be ordered?
  - If a flavor can be chosen more than once, how many possible 5-scoop bowls could be ordered?
  - If a flavor cannot be chosen more than once, how many possible 4-scoop bowls could be ordered?

Asymptotics

Outline

- RAM model of computation
- Big-Oh
- Growth rates & dominance
- How-to do "big O" math
- Summations
- Logarithms

Let's make up a machine

- "Random Access Machine" (RAM)
- Gives us a computational model where:

- Every simple operation (+, -, *, =, if, call, etc) takes 1 "time step"
- Each memory access takes 1 time step
  - Have as much memory as needed
  - Doesn't matter if memory is in cache or on disk
- Loops/subroutines are *not* simple operations

RAM Model

- This is an abstract machine that helps us assess algorithm performance.
- Lets us evaluate algorithms machine-independently.

RAM model

- Counts the steps taken, the amount of *time* taken to complete operations

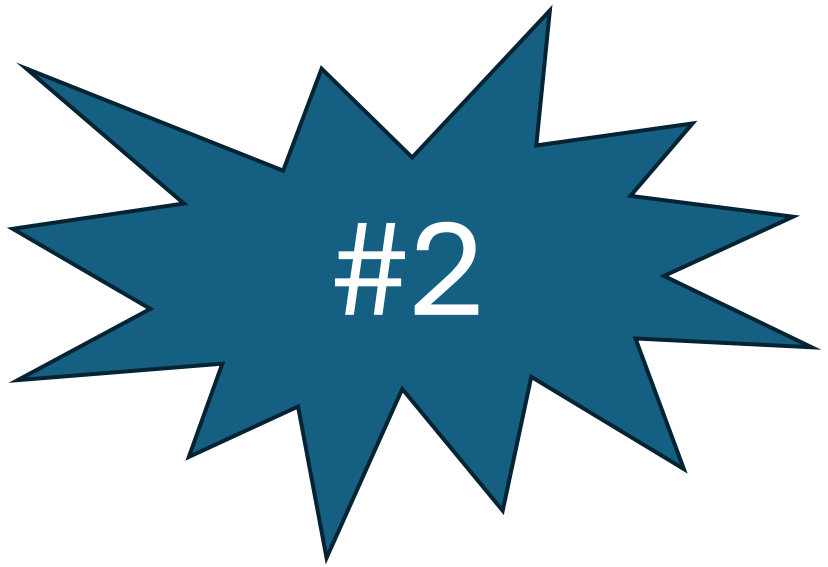
- Machine-independent
- Language-independent

How many instructions are completed?

```
int[] a = {1, 2, 3, 4, 5};
int total = 0;
total += a[0];
total += a[1];
total += a[2];
total += a[3];
total += a[4];
System.out.println(total);
```

How many instructions are completed?

```
a = [1,2,3,4,5]
total = 0
for x in a:
    total += x
print(total)
```



How many instructions done generally?

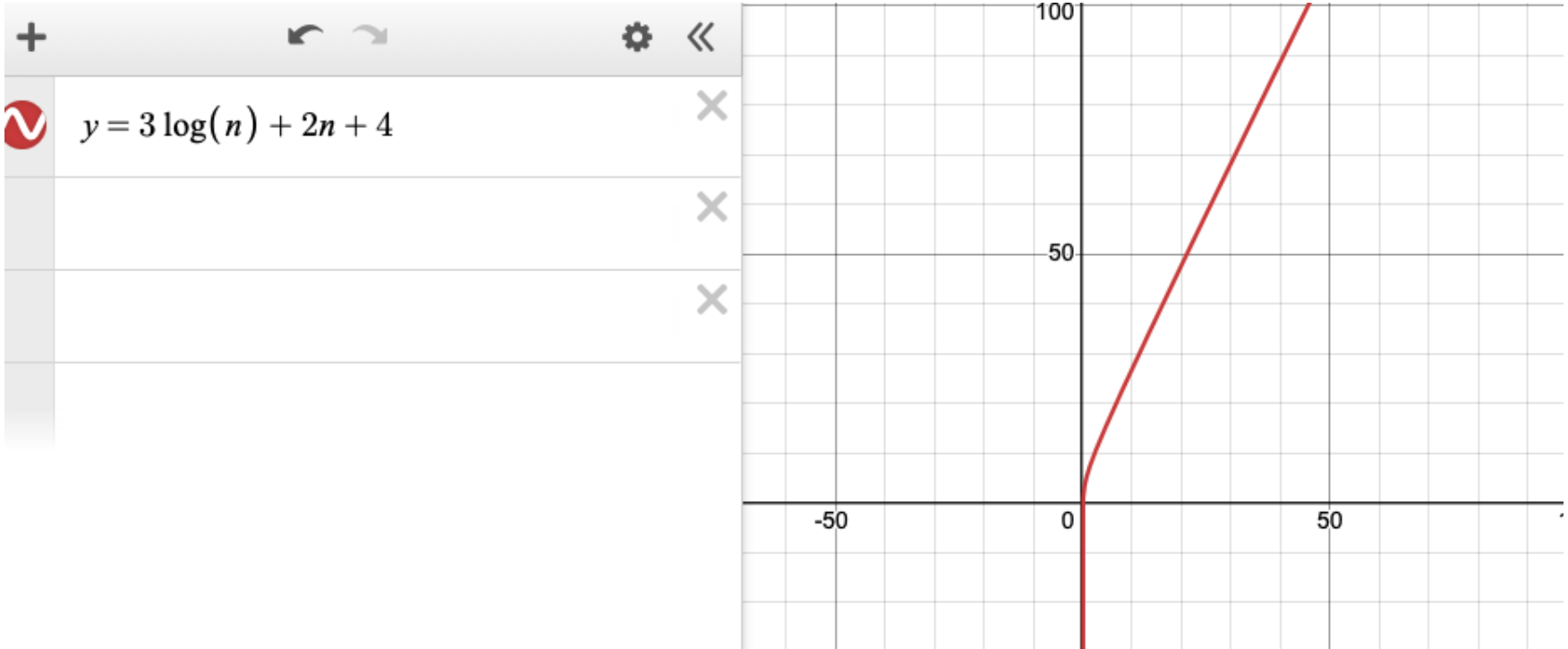
- Write as an equation: ex. $f(n) = 3n^2 + 100n + 6$
  - $f(n)$ or $T(n)$ often used
    - T = time

- Can simulate with any input – concrete instances
- But we want to generalize it.
  - How does our algorithm work under any possible circumstances?

- Best-case: minimum possible number of steps
- Worst-case: maximum possible number of steps
- Average-case: average number of possible steps

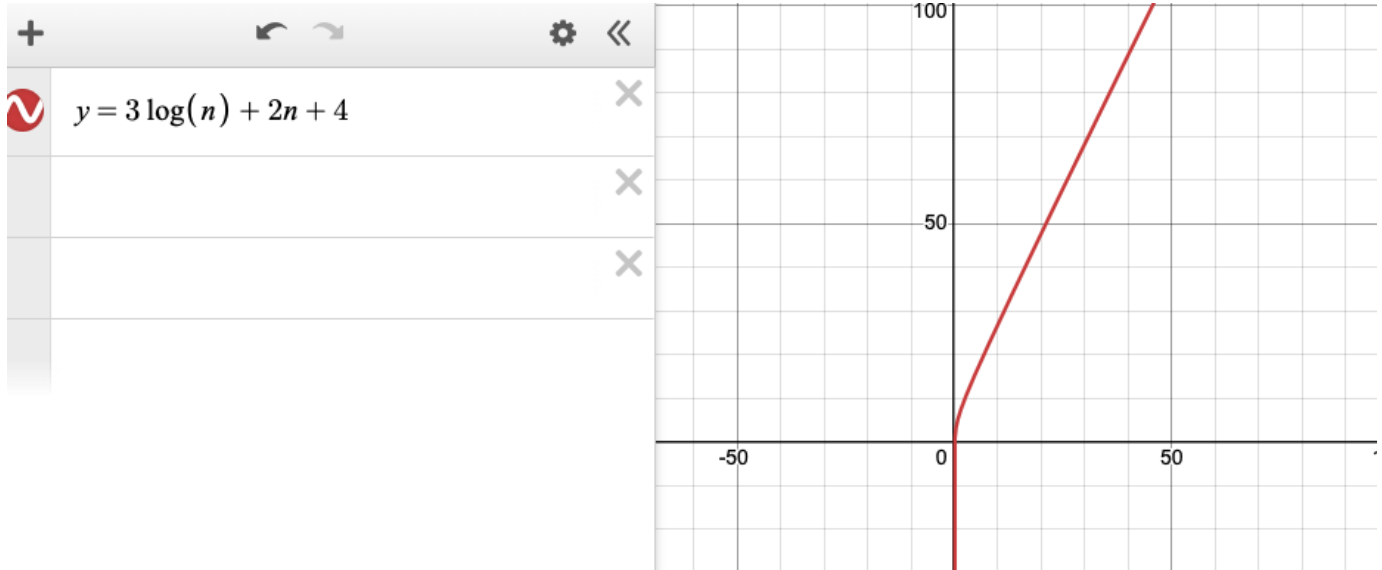
That equation is a function

- Can plug it in to desmos



That equation is a function

- Positive X-axis: size of data
  - Doesn't make sense to have 0 or negative data
- Positive Y-axis: number of steps the algorithm takes based on the size of the data input
- We also care about its *long-term behavior*



That equation is a function

- Long-term behavior = limits!
- But Calculus I is not a prerequisite for this course.
- We have another way to express this.

Search for 2 other functions

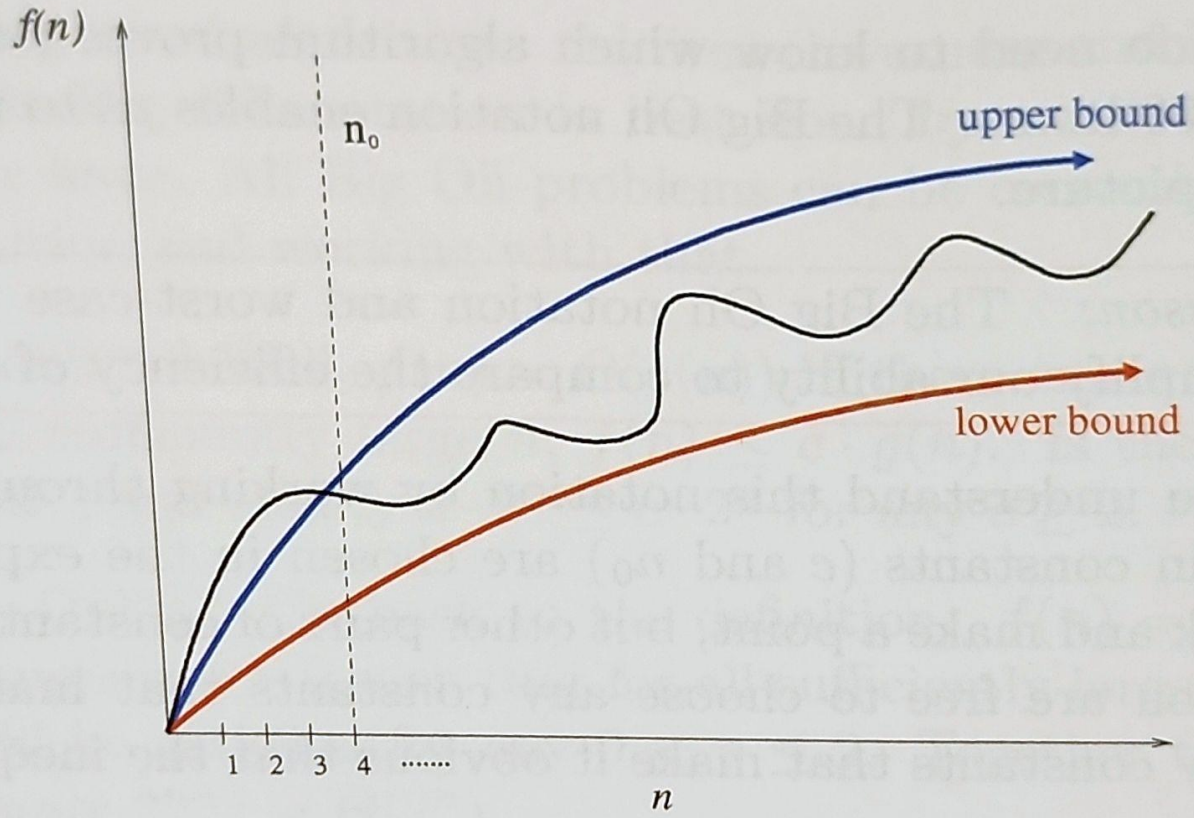


Figure 2.2: Upper and lower bounds valid for $n > n_0$ smooth out the behavior of complex functions.

Search for 2 other functions

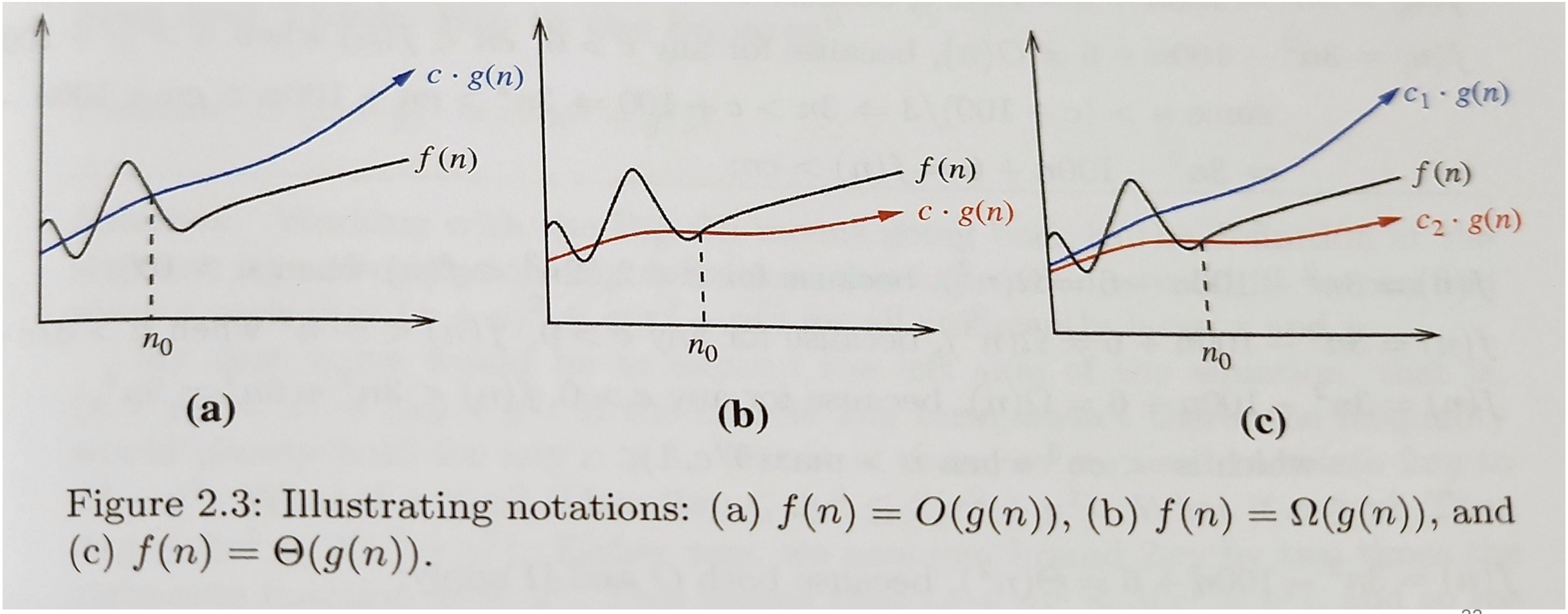


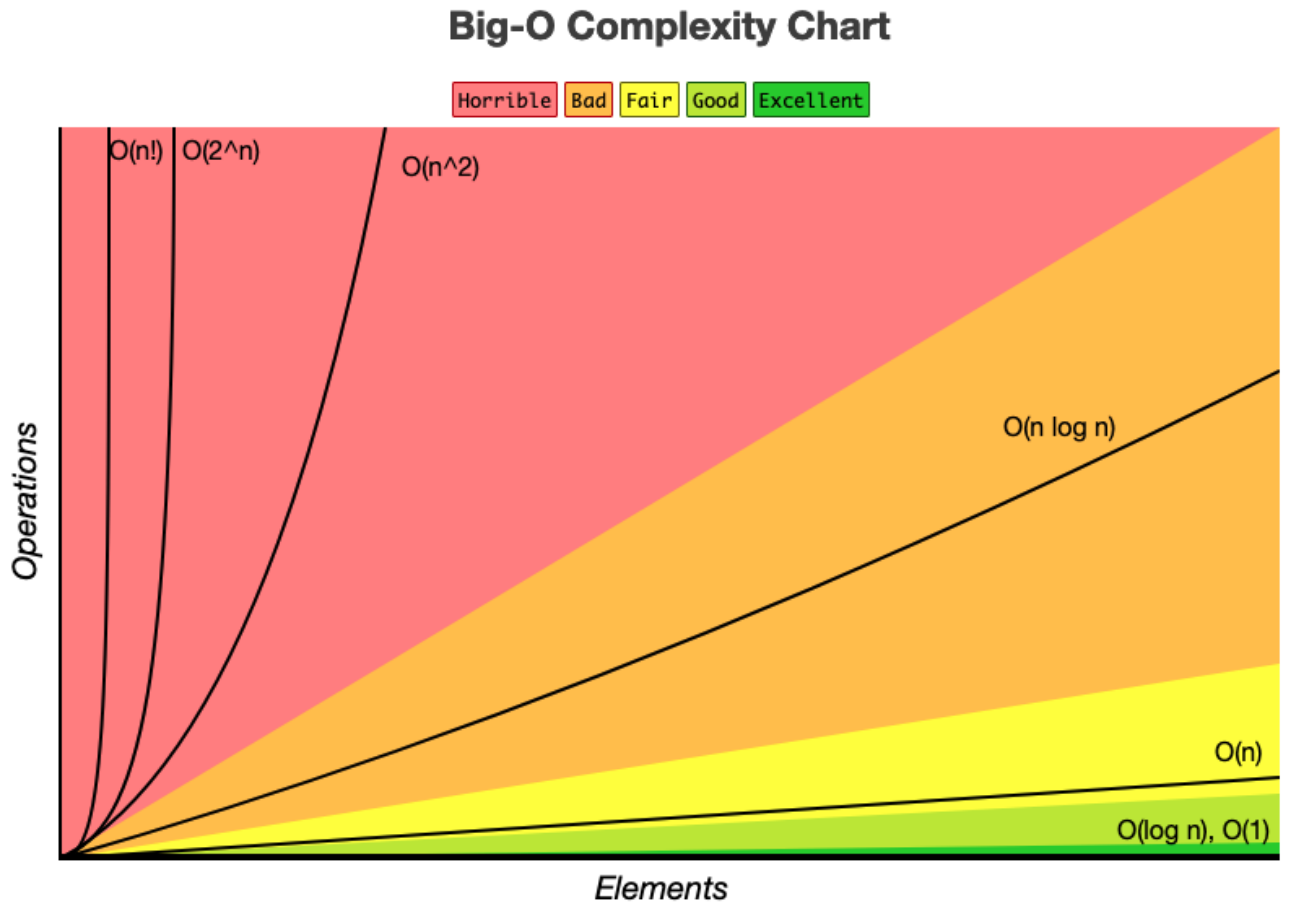
Figure 2.3: Illustrating notations: (a) $f(n) = O(g(n))$, (b) $f(n) = \Omega(g(n))$, and (c) $f(n) = \Theta(g(n))$.

Why we're doing this

- Once we write the time function for a given algorithm,
- It could be unpredictable
- Hard to "eyeball" what's good:
  - $f(n) = 3n^2 - 100n + 6$
  - $f(n) = 2^{n+1}$
  - $f(x, y) = x^2 + y^2$
  - $T(n) = 1275n^2 + 4353n + 834log_2\, n + 13546$
- We like to specify what "bucket" an algorithm is in

Specifying a bucket

- Something that roughly looks like…



Specifying a bucket

- We're going to use symbols to mean worst/best/average.
  - O - worst
  - Ω – best
  - Θ - average

- We find a simplified bucket to assign to the function.
- That goes in parentheses after the symbol.
  - Ex: $O(n \log n)$
- These symbols *look* like functions, but they're statements.
  - That means no $O(x) = n$. Just write "$O(n)$", or "Big-O is n".

Upper bound

- $f(n) = O(g(n))$ when there exists some constant c such that $f(n) \leq c \cdot g(n)$. Then we can say
  - $c \cdot g(n)$ is an *upper bound* on f(n).

- A function $g(n)$ is a worst-case (upper) bound for $f(n)$
  - When we can choose a constant c
  - And look at $c * g(n)$
  - And pick *values* of n for where $c * g(n)$ is bigger than $f(n)$
    - As n increases

Lower bound

- $f(n) = \Omega(g(n))$ when there exists some constant c such that $f(n) \geq c \cdot g(n)$. Then we can say
  - $c \cdot g(n)$ is a *lower bound* on f(n).

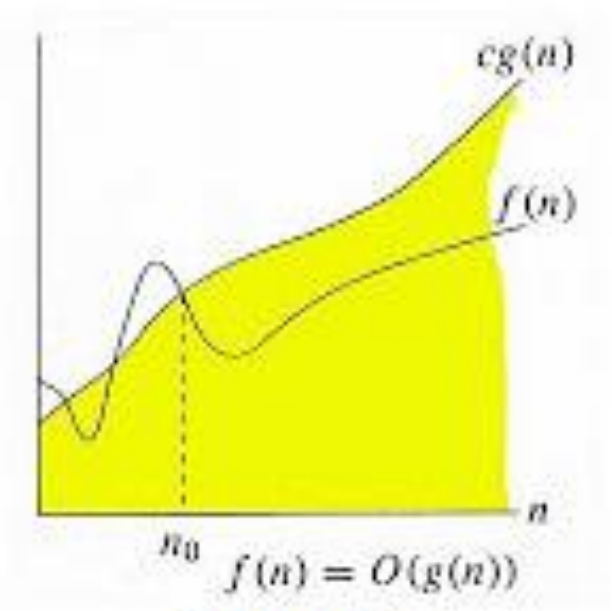
- A function $g(n)$ is a best-case (lower) bound for $f(n)$
  - When we can choose a constant c
  - And look at $c * g(n)$
  - And pick *values* of n for where $c * g(n)$ is smaller than $f(n)$
    - As n increases

Average-case bound

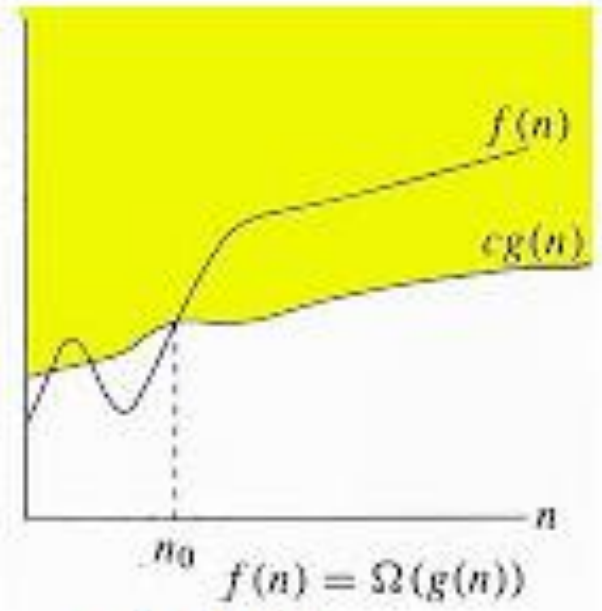
- $f(n) = \Theta(g(n))$ when there exists some c_1 and c_2 such that $c_1 \cdot g(n) \geq f(n)$ and $c_2 \cdot g(n) \leq f(n)$. Then we can say
  - $g(n)$ is an average-case bound on $f(n)$.

- A function $g(n)$ is an average- case bound for $f(n)$ when:
  - We can find both an upper and lower bound function for $f(n)$
- A function $g(n)$ is only in Big-Theta for an algorithm when it's BOTH an upper bound AND a lower bound

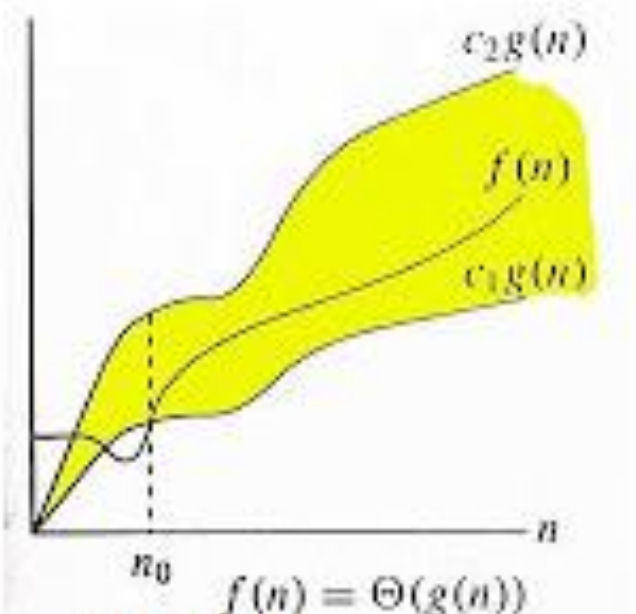
Just different constants multiplied with the function



Big Oh



Omega



Theta

Big-Theta is a set

- Big-Theta is the "average case" function
- So, needs both an upper and lower bounds

Tips and Tricks

- Look at the leading (highest-powered) term.
  - Ex. n^2 or $n \log n$.
  - This is $g(n)$.
- Choose a sufficiently large constant.
  - You've got c.
- Choose a place on the X-axis to start measuring from.
  - This is x_0, or n_0, depending on what variable you've called it.
- Plug numbers chosen into Desmos/a calculator.
  - Is the Y-value of $c \cdot g(x_0)$ bigger (resp. smaller) than $f(n)$? Then you can say "The Big-O (resp. Big-Omega) of $f(n)$ is $g(n)$", or just write $O\big(g(n)\big)$
  - Do that twice, once for Big-O, once for Big-Omega, and you get Big-Theta.

Problems

- What is the worst-case (O(n)), best-case ($\Omega(n)$), and average-case ($\Theta(n)$) for the following functions?
  - $f(n) = 3n^2 - 100n + 6$
  - $f(n) = 2^{\{n+1\}}$
  - $f(n) = x^2 + y^2$
  - $T(n) = 1275n^2 + 4353n + 834log_2\, n + 13546$

$$f(n) = 3n^2 - 100n + 6$$

$$f(n) = 3n^2 - 100n + 6$$

$$f(n) = 3n^2 - 100n + 6$$

$$f(n) = 2^{\mathrm{n+1}}$$

Growth Rates & Dominance Relations

- $f(n) = 0.001n^2$ vs. $g(n) = 1000\,n^2$
- $g(n)$ is a million times larger

- But $O(f(n)) = O(g(n)) = n^2$
  - What gives?

n	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure 2.4: Running times of common functions measured in nanoseconds. The function $\lg n$ denotes the base-2 logarithm of n.

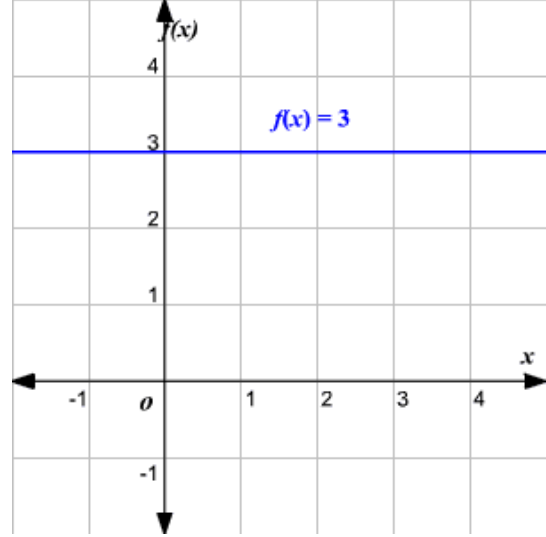
Insights

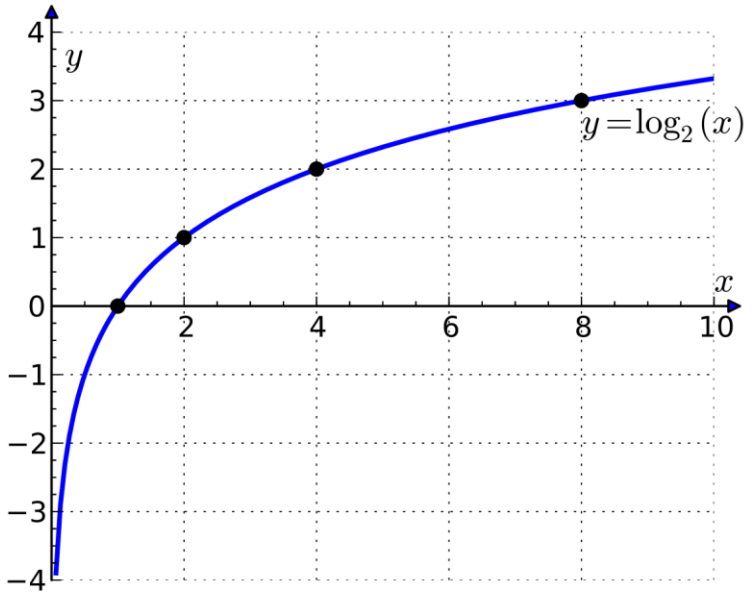
- All algorithms at $n = 10$ take roughly same amount of time
- n! is useless when $n >= 20$
- 2^n not good for n>40
- n^2 ok up to 10,000, but bad for n>1,000,000
- Linear, n log n fail at 1 billion
- O(log n) good pretty much forever

Classes of functions

- Constant functions, f(n) = 1
  - Adding two numbers
  - Print "The Star Spangled Banner"
  - Find the minimum of n and 100

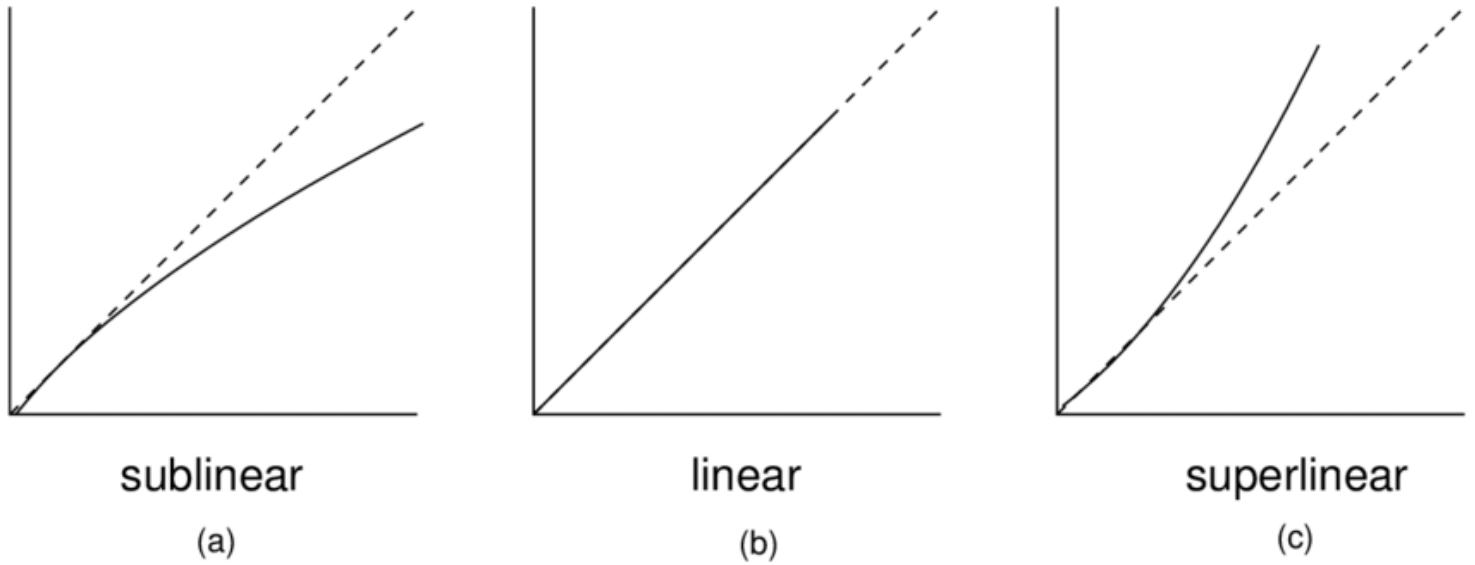
- Logarithmic functions, f(n) = log n
  - Binary search
  - Also called sublinear





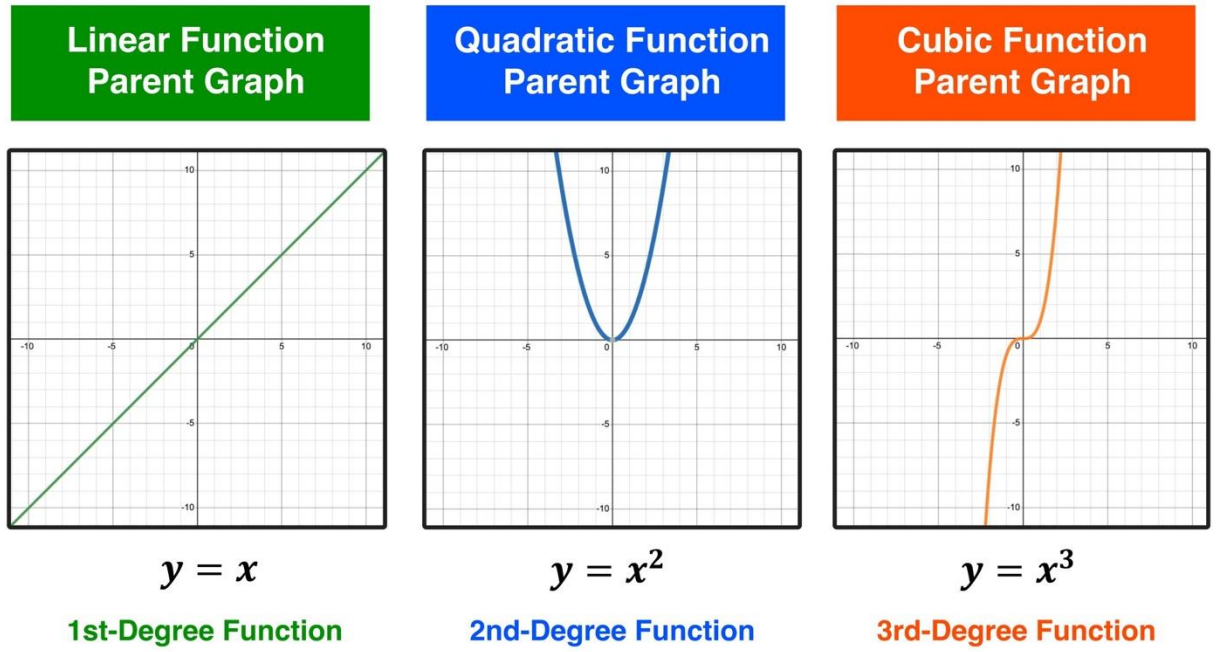
Classes of functions

- Linear functions, f(n) = n
  - Looking at each item once in an n-element array
    - Find the biggest/smallest
    - Compute average value
- Superlinear functions, f(n) = n log n
  - Quicksort
  - Mergesort



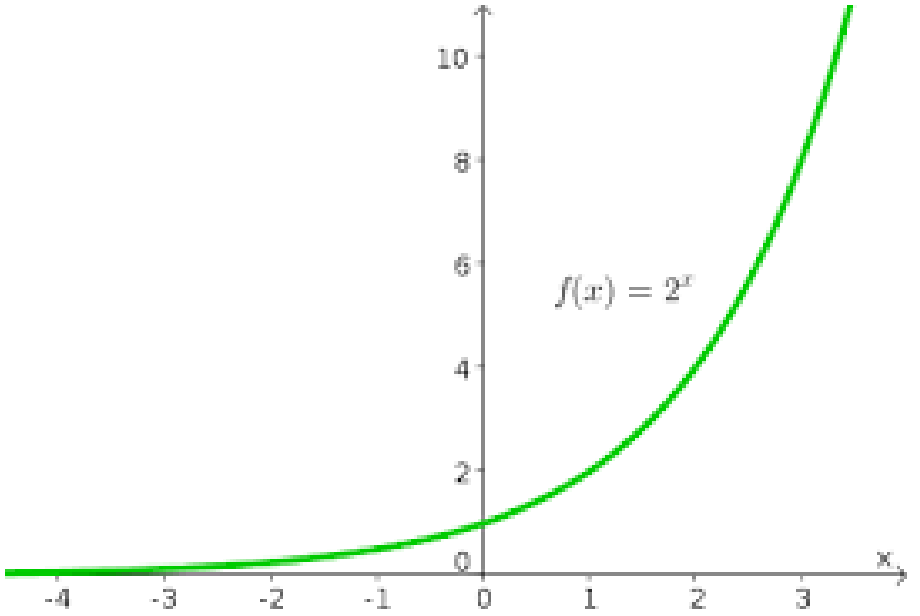
Classes of functions

- Quadratic functions, f(n) = n^2
  - Look at all pairs of items
  - Insertion, selection sorts
- Cubic functions, f(n) = n^3
  - Enumerating all triples
  - Some dynamic programming algorithms

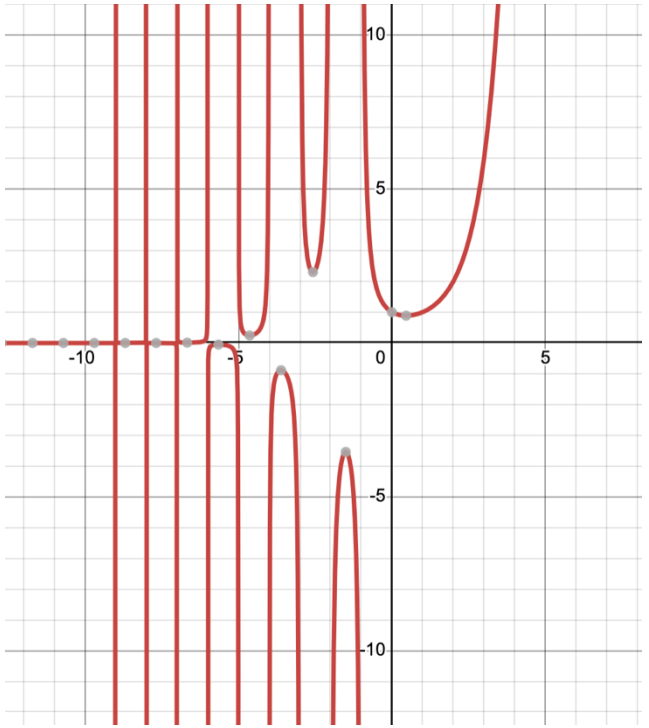


Classes of functions

- Exponential functions, f(n) = c^n given c > 1
  - Enumerating all subsets (2^n)



- Factorial functions, f(n) = $n!$
  - Generating all permutations / ordering of items



How they relate

- $n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$
- $x \gg y$ means "x grows faster than y"

Working with these functions

- Addition:
  - Focus on the biggest term.

- Big-Theta of $n^3 + n^2 + 1 = \Theta(n^3)$
- If $f(n) = O(n^2), g(n) = O(n^2)$, then $f(n) + g(n) = O(n^2)$ too.

Ignore the constants

$$O(c \cdot f(n)) \rightarrow O(f(n))$$
$$\Omega(c \cdot f(n)) \rightarrow \Omega(f(n))$$
$$\Theta(c \cdot f(n)) \rightarrow \Theta(f(n))$$

Multiplying by a constant does not change the asymptotics.

$$O(f(n)) \cdot O(g(n)) \rightarrow O(f(n) \cdot g(n))$$
$$\Omega(f(n)) \cdot \Omega(g(n)) \rightarrow \Omega(f(n) \cdot g(n))$$
$$\Theta(f(n)) \cdot \Theta(g(n)) \rightarrow \Theta(f(n) \cdot g(n))$$

When both functions in a product are increasing, both are Important.

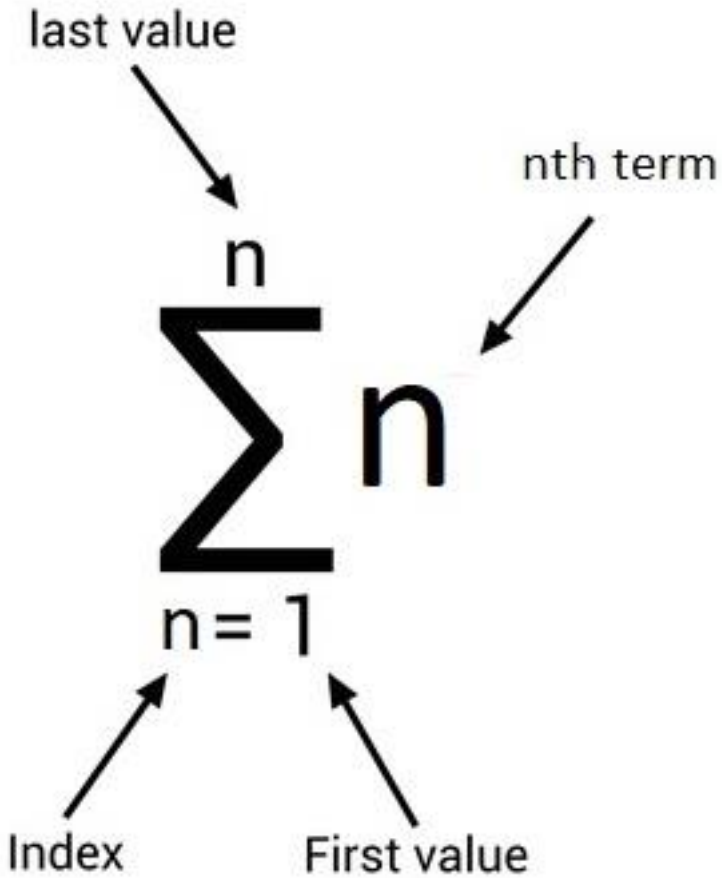
This is why nested loops take O(n^2)!

Tools we need

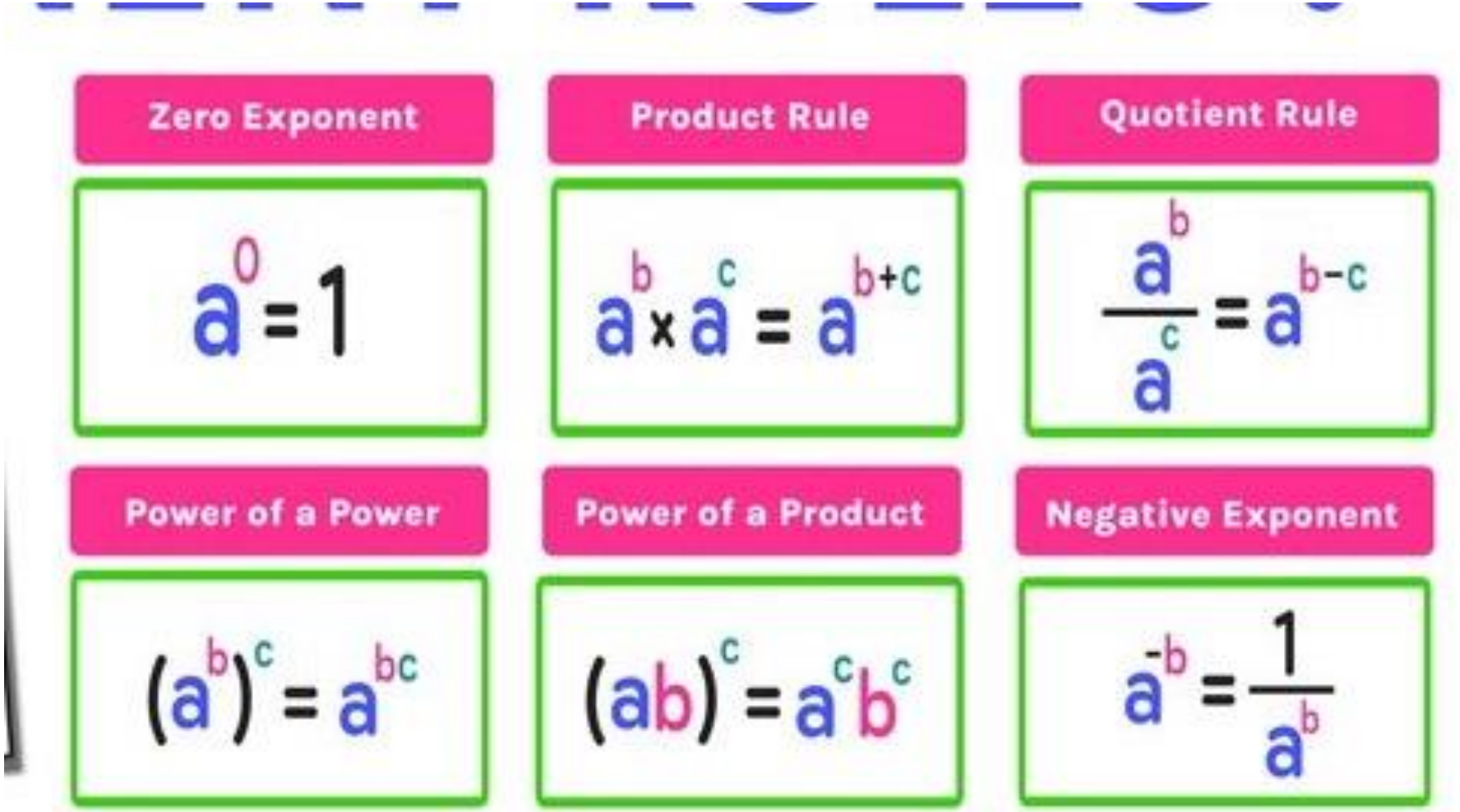
- Summations
- Logarithms

Summations

- Help us add things up



Exponents



From: mashupmath

Logarithms

- Repeated divisions.