CS 5002: Discrete Structures **Fall 2018**

Lecture 9: November 8, 2018 [1]

Instructors: Adrienne Slaughter, Tamara Bonaci

Introduction to Algorithms

Readings for this week:
Rosen, Chapter 2.1, 2.2, 2.5, 2.6
Sets, Set Operations, Cardinality of Sets, Matrices

9.1 Overview

Objective: Introduce algorithms.

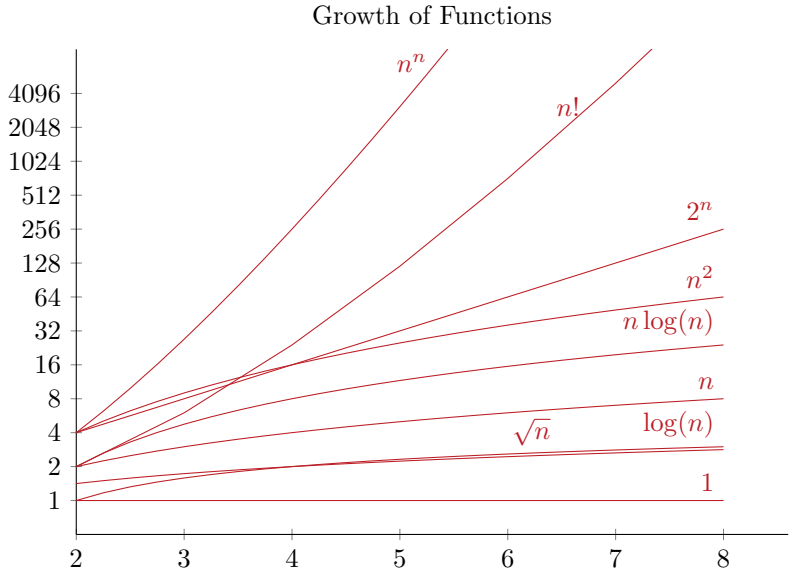
1. Review logarithms
2. Asymptotic analysis
3. Define Algorithm
4. How to express or describe an algorithm
5. Run time, space (Resource usage)
6. Determining Correctness
7. Introduce representative problems

1. foo

9.2 Asymptotic Analysis

The goal with asymptotic analysis is to try to find a bound, or an asymptote, of a function. This allows us to come up with an "ordering" of functions, such that one function is definitely bigger than another, in order to compare two functions. We do this by considering the value of the functions as n goes to infinity, so for very large values of n, as opposed to small values of n that may be easy to calculate.

Once we have this ordering, we can introduce some terminology to describe the relationship of two functions.

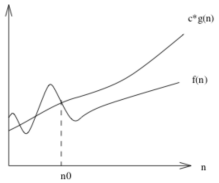


From this chart, we see:

$$1 \ll \log n \ll \sqrt{n} \ll n \ll n\log(n) \ll n^2 \ll 2^n \ll n! \ll n^n \tag{9.1}$$

Complexity	Terminology
$\Theta(1)$	Constant
$\Theta(\log n)$	Logarithmic
$\Theta(n)$	Linear
$\Theta(n \log n)$	Linearithmic
$\Theta\left(n^b\right)$	Polynomial
$\Theta(b^n)$ (where $b > 1$)	Exponential
$\Theta(n!)$	Factorial

9.2.1 Big-O: Upper Bound



Definition 9.1 (Big-O: Upper Bound) *$f(n) = O(g(n))$ means there exists some constant c such that $f(n) \leq c \cdot g(n)$, for large enough n (that is, as $n \to \infty$).*

We say $f(n) = O(g(n))$

Example: I claim $3n^2 - 100n + 6 = O(n^2)$. I can prove this using the definition of big-O:

$$f(n) = 3n^2 - 100n + 6 \tag{9.2}$$
$$g(n) = n^2 \tag{9.3}$$
$$\Rightarrow 3n^2 - 100n + 6 \leq c \cdot n^2 \text{ for some } c \tag{9.4}$$
$$\text{If } c = 3: \quad 3n^2 - 100n + 6 \leq 3n^2 \tag{9.5}$$

To prove using Big-O:

- Determine $f(n)$ and $g(n)$
- Write the equation based on the definition
- Choose a c such that the equation is true.
  - If you can find a d, then $f(n) = O(g(n))$. If not, then $f(n) \neq O(g(n))$.

These statements are all true:

$$3n^2 - 100n + 6 = O(n^2) \tag{9.6}$$
$$3n^2 - 100n + 6 = O(n^3) \tag{9.7}$$
$$3n^2 - 100n + 6 \neq O(n) \tag{9.8}$$

Proving 9.7:

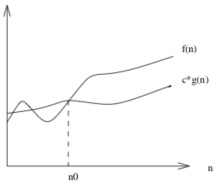
$$f(n) = 3n^2 - 100n + 6 \tag{9.9}$$
$$g(n) = n^3 \tag{9.10}$$
$$\Rightarrow 3n^2 - 100n + 6 = c \cdot n^3 \quad (\text{for some } c) \tag{9.11}$$
$$\text{If } c = 1: \quad 3n^2 - 100n + 6 \leq n^3 \quad (\text{when } n > 3) \tag{9.12}$$

We also know this to be true because order is ***transitive***: if $f(n) = O(g(n))$, and $g(n) = O(h(n))$, then $f(n) = O(h(n))$. Since $n^2 = O(n^3)$, then any $f(n) = O(n^2)$ is also $O(n^3)$.

Proving 9.8:

$$f(n) = 3n^2 - 100n + 6 \tag{9.13}$$
$$g(n) = n \tag{9.14}$$
$$\text{For any } c: \quad cn < 3n^2 \quad (\text{when } n > c) \tag{9.15}$$

9.2.2 Big-Omega: Lower Bound



Definition 9.2 (Big-Omega: Lower Bound) *$f(n) = \Omega(g(n))$ means there exists some constant c such that $f(n) \geq c \cdot g(n)$, for large enough n (that is, as $n \to \infty$).*

We say $f(n) = \Omega(g(n))$ or "f of n is Big Omega of g of n"

Example: I claim $3n^2 - 100n + 6 = \Omega(n^2)$. I can prove this using the definition of big-Omega:

$$f(n) = 3n^2 - 100n + 6 \tag{9.16}$$
$$g(n) = n^2 \tag{9.17}$$
$$\Rightarrow 3n^2 - 100n + 6 \geq c \cdot n^2 \text{ for some } c \tag{9.18}$$
$$\text{If } c = 2: \quad 3n^2 - 100n + 6 \leq 2n^2 \tag{9.19}$$

We show Big-Omega the same way we show Big-O.
These statements are all true:

$$3n^2 - 100n + 6 = \Omega(n^2) \tag{9.20}$$
$$3n^2 - 100n + 6 \neq \Omega(n^3) \tag{9.21}$$
$$3n^2 - 100n + 6 = \Omega(n) \tag{9.22}$$

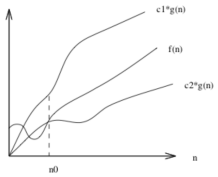
Proving 9.21:

$$f(n) = 3n^2 - 100n + 6 \tag{9.23}$$
$$g(n) = n^3 \tag{9.24}$$
$$\Rightarrow 3n^2 - 100n + 6 \geq c \cdot n^3 \quad (\text{for some } c) \tag{9.25}$$
$$\text{If } c = 1: \quad 3n^2 - 100n + 6 \geq n^3 \quad (\text{when } n > 3) \tag{9.26}$$

Proving 9.22:

$$f(n) = 3n^2 - 100n + 6 \tag{9.27}$$
$$g(n) = n \tag{9.28}$$
$$\text{For any } c: \quad cn < 3n^2 \quad (\text{when } n > 100c) \tag{9.29}$$

9.2.3 Big-Theta: "Tight" Bound



Definition 9.3 (Big-Theta: "Tight" Bound) *$f(n) = \Theta(g(n))$ means there exists some constants $c1$ and $c2$ such that $f(n) \leq c_1 g(n)$ and $f(n) \geq c_2 g(n)$.*
We say $f(n) = \Theta(g(n))$ or "f of n is Big-Theta of g of n".

Definition 9.4 (Theta and "order of") *When $f(x) = \Theta(g(x))$, it is the same as saying $f(x)$ is the order of $g(x)$, or that $f(x)$ and $g(x)$ are the same order.*

$3n^2 - 100n + 6 = \Theta(n^2)$ Both O and Ω apply

$3n^2 - 100n + 6 \neq \Theta(n^3)$ Only O applies
$3n^2 - 100n + 6 \neq \Theta(n)$ Only Ω applies

Interesting Aside Donald Knuth popularized the use of Big-O notation. It was originally inspired by the use of "ell" numbers, written as $L(5)$, which indicates a number that we don't know the exact value of, but is less than 5. That allows us to reason about the value without knowing the exact value: we know $L(5) < 100$, for example.

Theorem 9.5 *If* $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, *then* $f(x) = O(^n)$

a) $f(x) = 17x + 11$ b) $f(x) = x^2 + 1000$
c) $f(x) = x \log x$ d) $f(x) = x^4/2$
e) $f(x) = 2^x$ f) $f(x) = \lfloor x \rfloor \cdot \lceil x \rceil$

9.2.4 Logs, Powers, Exponents

We've seen $f(n) = O(n^d)$. If $d > c > 1$, then $n^c = O(n^c)$. n^c is $O\left(n^d\right)$, but n^d is not $O\left(n^c\right)$.

$\log_b n$ is $O(n)$ whenever $b > 1$. Whenever $b > 1$, c and d are positive:

$$(\log_b n)^c \text{ is } O\left(n^d\right), \text{ but } n^d \text{ is not } \left(O\left(\log_b n\right)^c\right) \tag{9.30}$$

This tells us that every positive power of the logarithm of n to the base b, where b ¿ 1, is big-O of every positive power of n, but the reverse relationship never holds. In Example 7, we also showed that n is O(2n). More generally, whenever d is positive and b ¿ 1, we have

$$n^d \text{ is } O\left(b^n\right), \text{ but } b^n \text{ is not } O\left(n^d\right) \tag{9.31}$$

This tells us that every power of n is big-O of every exponential function of n with a base that is greater than one, but the reverse relationship never holds. Furthermore, we have when c ¿ b ¿ 1,

$$b^n \text{ is } O\left(c^n\right) \text{ but } c^n \text{ is not } O\left(b^n\right) \tag{9.32}$$

This tells us that if we have two exponential functions with different bases greater than one, one of these functions is big-O of the other if and only if its base is smaller or equal.

9.2.5 Adding Functions

There are a set of rules that govern combining functions together.

$$O(f(n)) + O(g(n)) \to O(max(f(n), g(n))) \tag{9.33}$$
$$\Omega(f(n)) + \Omega(g(n)) \to \Omega(max(f(n), g(n))) \tag{9.34}$$
$$\Theta(f(n)) + \Theta(g(n)) \to \Theta(max(f(n), g(n)) \tag{9.35}$$

These statements express the notion that the largest term of the statement is the dominant one. For example, $n^3 + 2n^2 + 3 = O(n^3)$.

Example: Prove that $n^2 = O(2^n)$.

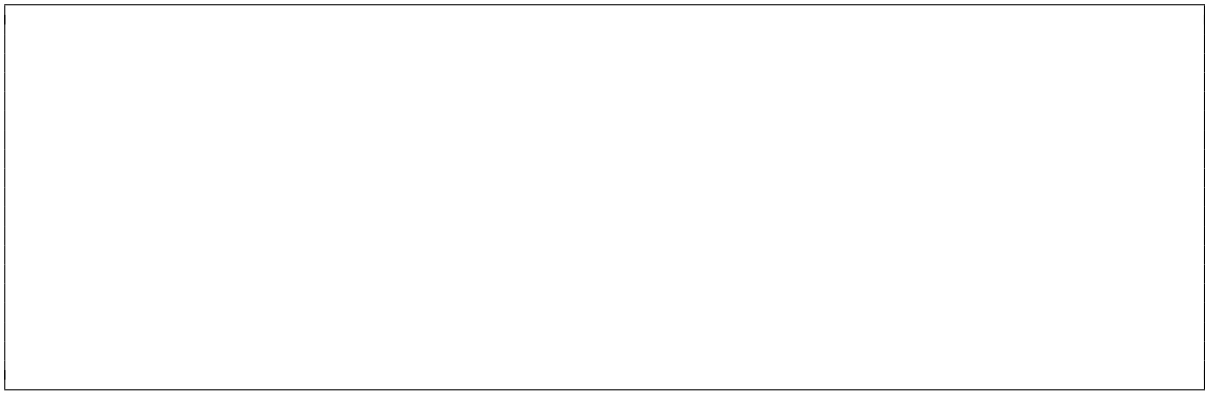
Example: Prove that if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_(n) = O(g_1(n) + g_2(n))$.

Example:

$$f(n) = n + \log n \tag{9.36}$$
$$g(n) = \sqrt(n) \tag{9.37}$$

Is $f(n) = O(g(n))$, $g(n) = O(f(n))$, or both?

Example: If $f(n) = n + \log n + \sqrt{n}$, find a simple function g such that $f(n) = \Theta(g(n))$.



Summary

- $f(n) = O(g(n))$ means $c \cdot g(n)$ is an upper bound on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\leq c \cdot g(n)$, for large enough n (i.e., $n \geq n_0$ for some constant n_0).
- $f(n) = \Omega(g(n))$ means $c \cdot g(n)$ is a lower bound on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\geq c \cdot g(n)$, for all $n \geq n_0$.
- $f(n) = \Theta(g(n))$ means $c1 \cdot g(n)$ is an upper bound on $f(n)$ and $c_2 \cdot g(n)$ is a lower bound on $f(n)$, for all $n \geq n0$. Thus there exist constants $c1$ and $c2$ such that $f(n) \leq c_1 \cdot g(n)$ and $f(n) \geq c_2 \cdot g(n)$. This means that $g(n)$ provides a nice, tight bound on $f(n)$.

9.2.6 Introduction to Algorithms

- An ***algorithm*** is a set of instructions for accomplishing a task.
- Technically, any program is an algorithm
- We talk about algorithms as general approaches to specific problems
- An algorithm is *general*, but is ***implemented*** in code to make it specific

Algorithms are like Recipes

- If I were to use a simile, I'd say algorithms are like recipes.
- People have been cooking and baking for a looong time
  - Let's take advantage of solved problems and use them as starting blocks
- There are general approaches to different kinds of foods
- Each recipe for a chocolate chip cookie is a little different, but follows the same general structure.
- I can adapt a recipe for chocolate chip cookies to a different kind of cookie if I want.
- I might modify my recipe depending on the context I'm cooking in: cooking for a 200 person formal dinner versus playing around on a Saturday afternoon.

What is an algorithm?

- An algorithm is the part of the "recipe" that stays the same no matter what it's implemented in or what hardware it's running on.
- An algorithm solves a general, specified problem
- An ***algorithmic problem*** is specified by describing:
  - The set of instances it works on
  - Desired properties of the output

Example: Sorting

Input: A sequence of N numbers: $n_1, n_2, n_3, \ldots, n_n$

Output: The permutation of the input sequence such as $n_1 \leq n_2 \leq n_3 \ldots \leq n_n$

We look to ensure that an algorithm is:

- Correct
- Efficient in time
- Efficient in space

The rest of today:

- Example algorithms
  - Binary Search
  - Selection Sort
- Algorithm Analysis
  - Proving Correctness (briefly)
  - ***Run time***: How long does it take for an algorithm to run?
  - ***Run space***: How much extra memory/storage does an algorithm require?
- Asymptotic Analysis and Growth of Functions

9.3 Some Algorithms

9.3.1 Expressing Algorithms

Expressing Algorithms

We need some way to express the sequence of steps in an algorithm.

In order of increasing precision:

- English
- Graphically
- Pseudocode
- real programming languages (C, Java, Python, etc)

Unfortunately, ease of expression moves in the reverse order.

An algorithm is an *idea*. If the idea is not clear when you express the algorithm, then you are using a too low-level way to express it.

9.3.2 Binary search

Searching

Input: A set of N values: $n_1, n_2, n_3, \ldots, n_n$ and a target value t

Output: Whether the set contains t

Imagine...

A (sub) roster of athletes on the USA Olympic Ski & Snowboard team for 2018:

1	Andy Newell
2	Bryan Fletcher
3	Chloe Kim
4	Jessie Diggins
5	Lindsey Vonn
6	Sadie Bjornsen
7	Sophie Caldwell
8	Taylor Fletcher

Is Chloe Kim on the US Ski & Snowboard team for 2018?

Is Chloe Kim on the US Ski-Snowboard team for 2018?

Let's make this a little more complicated...

Assume I have 1,000,000 athletes. How do I answer this question?

OR: Maybe I can't actually *see* the list here in its entirety.

How do I search?

A slight aside...

Consider a dictionary (the book kind!) You want to look up a word. First, you open up to the middle. If you've gone too far, you split the first half of the dictionary; if you haven't gone far enough, you split the second half of the dictionary.

Binary Search

A: Array to search I: Item to find min: starting point max: end point

BINARY-SEARCH(A, I, min, max)

```
1  if (min == max)
2        return false
3  mid = min + ⌊(max − min)/2⌋
4  if (A[mid] == I)
5        return true
6  if (A[mid] < I)
7        BINARY-SEARCH(A, I, mid, max)
8  else
9        BINARY-SEARCH(A, I, min, mid)
```

```
1  BINARY-SEARCH(A, I, 1, numElems)
```

What's interesting about Binary Search?

- It's ***recursive***
  - It's defined in terms of itself
- Each time we call BINARY-SEARCH, we are searching on only half the size of the input
  - Since we know what the middle element is, we know whether our final element is before or after that one, so can discard half the array with each comparison!

Quick Review

1. I stated the problem
2. I described a solution in English (using a metaphor)
3. I described the solution with psuedocode
4. I provided a graphical solution

9.3.3 Selection sort

Selection Sort

Input: A sequence A of n numbers: $n_1, n_2, n_3, \ldots, n_n$, and empty sequence B

Output: The input sequence such that B contains the elements of A ordered such that $n_1 \leq n_2 \leq n_3 \ldots \leq n_n$

SELECTION-SORT(A, B)

```
1  for i = 1 to A.length
2        min_ind = 0
3        for j = 1 to A.length
4              if A[j] < A[min_ind]
5                    min_ind = j
6        B[i] = A[min_ind]
7        A[min_ind] = Inf
```

The C code

```c
void selection_sort(int a[], int b[], int len){

  for (int i=0; i<len; i++){
    int min_ind = 0;
    for (int j=0; j<len; j++){
      if (a[j] < a[min_ind]){
        min_ind = j;
      }
    }
    b[i] = a[min_ind];
    a[min_ind] = 10000; //sentinel val
  }
}
```

whew What did we just do?

- Expressing Algorithms
  - English
  - Psuedocode
  - Programming Language (C)
  - Graphically!
- Binary Search– a first ***search*** algorithm
- Selection sort– a first ***sort*** algorithm
- Next up: ***Analyzing*** the algorithms

9.4 Analysis

What is Algorithm Analysis?

When we analyze algorithms, we are analyzing for 3 things:

1. Correctness
2. Run time
3. Run space

9.4.1 Correctness

Proving Correctness

How to prove that an algorithm is correct?

For any algorithm, we must prove that it always returns the desired output for all legal instances of the problem.

For sorting, this means even if the input is already sorted or it contains repeated elements.

Proof by:

- Induction
- Counterexample
- Loop Invariant

Proof by Counterexample Searching for counterexamples is the best way to disprove the correctness of a heuristic.

- Think about small examples
- Think about examples on or around your decision points
- Think about extreme examples (big or small)

Proof by Induction

Failure to find a counterexample to a given algorithm does not mean "it is obvious" that the algorithm is correct.

Mathematical induction is a very useful method for proving the correctness of recursive algorithms.

1. Prove base case
2. Assume true for arbitrary value n
3. Prove true for case $n + 1$

Proof by Loop Invariant

Built off proof by induction.

Useful for algorithms that loop.

1. Find p, a loop invariant
2. Show the base case for p
3. Use induction to show the rest.

9.4.2 Run time

What is run time?

The amount of time it takes for an algorithm to run, in terms of the size of the input n.

A faster algorithm running on a slower computer will always win for sufficiently large instances.

Usually, problems don't have to get that large before the faster algorithm wins.

This is where Big-O comes in.

Essentially, the number of lines of code that are run.

What is run time?

- Best Case
  - Given an optimal input and all the best decisions that can be made, how many steps until the algorithm terminates?
  - In a sort problem, it's usually a sorted input.
- Worst Case
  - Given the worst possible input and all the worst decisions that can be made, how many steps until the algorithm terminates?
  - In a search problem, it's usually the last item looked at.
- Average Case
  - Somewhere between the two; frequently an averaging of best & worst.

Run Time of Selection Sort

SELECTION-SORT(A, B)

```
1  for i = 1 to A.length.                    // n
2  min_ind = −1                              // 1
3      for j = 1 to A.length                 // n
4          if A[j] < A[min_ind]              // 1
5              min_ind = j                   // 1
6      B[i] = A[min_ind]                     // 1
7      A[min_ind] = Inf                      // 1
```

TODO: Show run time in terms of a sum?

$\sum_{i=1}^{n}(3 + \sum_{j=1}^{n} 2)$

Run time: $n \cdot (n + 1 + 1 + 1 + 1 + 1) \Rightarrow O(n^2)$

Is this best, worst or average?

Run Time of Selection Sort

Best case: {1, 2, 3, 4, 5, 6}

(It's already sorted)

Worst case: {6, 5, 4, 3, 2, 1 }

(It's reverse sorted)

Average case: {1, 6, 4, 5, 2, 3}

(It's a little of this and that)

Actually, for Selection Sort, there's no difference in run time for Best/Worst/Average case.

In all cases, we still iterate through all the elements.

$\Rightarrow O(n^2)$

Run Time of Binary Search

```
BINARY-SEARCH(A, I, min, max)
1  if (min == max)
2      return false
3  mid = min + ⌊(max − min)/2⌋
4  if (A[mid] == I)
5      return true
6  if (A[mid] < I)
7      BINARY-SEARCH(A, I, mid, max)
8  else
9      BINARY-SEARCH(A, I, min, mid)
```

Best case:

BINARY-SEARCH({1, 2, 3}, 2, 1, 3)

Worst case:

BINARY-SEARCH({1, 2, 3, 4, 5, 6, 7, 9, 10}, 2, 1, 10)

Run Time of Binary Search

```
BINARY-SEARCH(A, I, min, max)
1  if (min == max)
2      return false                                  // 1
3  mid = min + ⌊(max − min)/2⌋                       // 1
4  if (A[mid] == I)
5      return true                                   // 1
6  if (A[mid] < I)
7      BINARY-SEARCH(A, I, mid, max)                 // R(n/2)
8  else
9      BINARY-SEARCH(A, I, min, mid)                 // R(n/2)
```

$$R(n) = 1 + 1 + 1 + R(n/2)$$
$$R(n) = O(\lg n)$$

Run Time of Binary Search

Best Case: $O(1)$

Worst Case: $O(\lg n)$

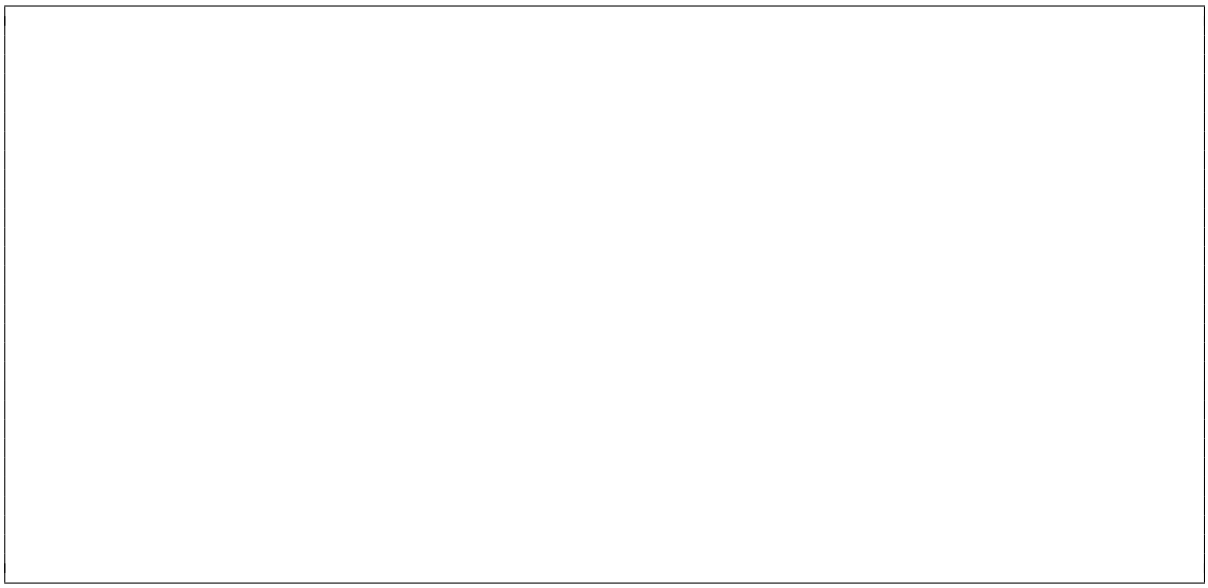
Average Case: $O(\lg n)$

Example: Given the following algorithm:

```
PRINTFOOBAR(n)
1  for i = 1 to n/2
2      for j = i to n − 1
3          for k = 1 to j
4              PRINT('FOOBAR')
```

Assume n is even. Let $T(n)$ denote the number of times 'foobar' is printed as a function of n.

- Express $T(n)$ as three nested summations.
- Simplify the summation.



Run time as Clock time

So far we've focused on counting "number of instructions" as a proxy for measuring the "clock time" (that is, number of seconds) that an algorithm will run. However, we can use the number of instructions as a tool to help us figure out clock time, when we have some specifics.

$$\frac{\text{Number of Instructions}}{\text{Instructions Per Second}} = \text{Number of Seconds} \tag{9.38}$$

The number of instructions is measured in terms of n, the size of the input to the algorithm. While (for the most part) the number of instructions is about the same from machine to machine, what varies is the number of instructions per second that are run. This gives us 3 variables: number of instructions, instructions per second and number of seconds to run the algorithm. Therefore, if we know 2 we can calculate the third.

Here's an example: Let's say I've implemented Selection-Sort on my MacBook Pro. I know Selection-Sort takes n^2 instructions to run. I choose to run it on an input length of 10,000 items. It takes 2 clock seconds to run (this is a number I'm choosing for illustration purposes; that's way too long!).

$$\frac{\text{Number of Instructions}}{\text{Instructions per Second}} = \text{Number of Seconds} \tag{9.39}$$

$$\frac{n^2}{\text{Instructions Per Second}} = 2 \text{ seconds} \tag{9.40}$$

$$n = 10,000: \quad \frac{10,000^2}{x \text{ Instructions per Second}} = 2 \text{ seconds} \tag{9.41}$$

$$\frac{10,000^2 \text{ instructions}}{2 \text{ seconds}} = x = 50 \text{ MIPS}^2 \tag{9.42}$$

[2] Millions of Instructions per Second. For reference, the iPhone 6 was probably around 25,000 MIPS (in 2014).

Thus, if we know our algorithm has n^2 instructions, and we measure that it takes 2 seconds to run on our machine with an input of 10,000 items, then our machine runs at about 50 MIPS.

Further, now that we know our machine runs at 50 MIPS, we can use that to estimate how long it will take to run a different algorithm (that is, different run time) or different input size.

Let's say we have one million items as input to the same algorithm:

$$n = 1,000,000: \quad \frac{(1,000,000)^2}{50 \text{ MIPS}} = ? \tag{9.43}$$

$$\frac{(1,000,000)^2}{50,000,000} = 20,000 \text{ seconds} \tag{9.44}$$

9.4.2.1 Runtime, Clocktime, and Efficiency

Let's take two relatively recent machines. One is powered by the Intel Core i7 500U which runs at 49,360 (roughly 50K MIPS), and the other is an Intel Core i7 2600K at 117,160 MIPS.

Runtime	Size of Input	Intel i7 A	Intel i7 B	
n	1,000	$\frac{1000}{50{,}000 \text{ MIPS}} = 0.02\mu\text{sec}$[3]	$\frac{1000}{117{,}000 \text{ MIPS}} = 0.009\mu\text{sec}$	linear
	10,000	$\frac{10000}{50{,}000 \text{ MIPS}} = 0.2\mu\text{secs}$		
	1,000,000	$\frac{1{,}000{,}000}{50{,}000 \text{ MIPS}} = 20\mu\text{secs}$	$\frac{1{,}000{,}000}{117{,}000 \text{ MIPS}} = 8.5\mu\text{secs}$	
$n \log n$	1000	$\frac{1000 \log 1000}{50{,}000 \text{ MIPS}} = \frac{3000}{50{,}000 \text{ MIPS}} = 0.06\mu\text{sec}$	$\frac{3000}{117{,}000 \text{ MIPS}} = 0.03\mu\text{sec}$	logarithmi
	10000			
	1,000,000	$\frac{1000000 \log 1000000}{50{,}000 \text{ MIPS}} = \frac{6{,}000{,}000}{50{,}000 \text{ MIPS}} = 120\mu\text{sec}$	$\frac{6{,}000{,}000}{117{,}000 \text{ MIPS}} = 51.3\mu\text{sec}$	
n^2	1000	$\frac{1000^2}{50{,}000 \text{ MIPS}} = 20\mu\text{secs}$	$\frac{1000^2}{117{,}000 \text{ MIPS}} = 8.5\mu\text{secs}$	quadratic/
	10000			
	1,000,000			
c^n	50	$\frac{2^{50}}{50{,}000 \text{ MIPS}} = \frac{1{,}125{,}899{,}906{,}842{,}624}{50{,}000{,}000{,}000} = 22518\mu\text{secs}$	$\frac{2^{50}}{117{,}000 \text{ MIPS}} = 9623\mu\text{secs}$[4]	exponentia
	10000			
	1,000,000			

Run Time, Summary

- We count up the number of statements that are run.
- Consider whether there's a difference in how long it takes a function to run given different inputs.
- Selection sort is $O(n^2)$, pretty much all the time.
- Binary search can be either $O(1)$ or $O(\lg n)$, depending on what the input looks like.

9.4.3 Memory Use

How much memory?

Another resource that we sometimes care about is the amount of memory it takes to run an algorithm.

Sometimes this is total, sometimes it's just the amount in addition to the input.

Binary Search:

Needs nothing, so memory is $O(1)$ (constant— nothing other than the input.

[4] $1\mu\text{sec} = 10^{-6}$ seconds

[4] $= \frac{1{,}125{,}899{,}906{,}842{,}624}{117{,}000{,}000{,}000}$

Selection sort:

Needs a whole other array! $O(n)$

Note: this is just how it was implemented here, in this discussion. If we chose not to use that other array, it wouldn't be $O(n)$.

9.5 Representative Problems

Stable Matching

- Gale-Shapley
- The problem of: matching residents with med schools
- Each resident has a prioritized list of schools they want to go to
- Each school has a prioritized list of students they want to accept
- A ***stable match*** is one where if either the school or the student is offered another match, they won't change.

Interval Scheduling

The Problem: We have a resource r, such as a classroom, and a bunch of requests $q : \{start, finish\}$. How can we schedule the requests to use the resource?

- We want to identify a set S of requests such that no requests overlap.
- Ideally, the S that we find contains the maximum number of requests.

In this diagram, we see three sets of requests.

Which set of requests is the preferred choice for the interval scheduling problem as defined?

Solution: A simple heuristic that is an example of a ***greedy algorithm***.

Weighted Interval Scheduling

The Problem: Same as interval scheduling, but this time, the request has a weight.

- The weight may be how much we'll earn by satisfying this request
- Find a subset that maximizes the weights
- This is very similar to the problem we just saw, but these weights cause a problem.
- Consider: If all requests except one have weight = 1, and one has weight greater than the sum of all the others.

Solution: An approach called ***dynamic programming***, where we calculate the weight of each subset and use that to find the best set overall.

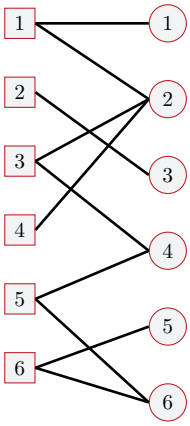
Bipartite Matching

The Problem: We have two groups of objects that we need to match, or assign to another object.

- An example: matching residents with med schools
- Each resident has a prioritized list of schools they want to go to
- Each school has a prioritized list of students they want to accept
- A ***stable match*** is one where if either the school or the student is offered another match, they won't change.

An Aside: Bipartite Graph

- A graph $G = (V, E)$ is ***bipartite*** if the nodes V can be partitioned into sets X and Y in such a way that every edge has one end in X and the other in Y.
- It's just a graph, but we tend to depict bipartite graphs in two columns to emphasize the bipartiteness.



Bipartite Matching

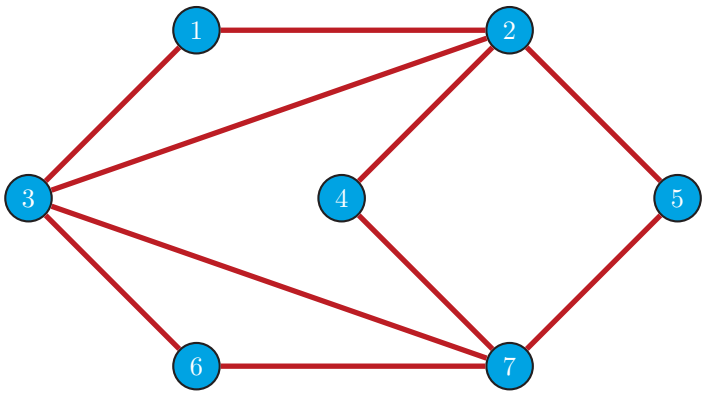
- Bipartite Matching is relevant when we want to match one set of things to another set of things.
  - Nodes could be Jobs and Machines; Edges indicate that a given machine can do the job.
  - Nodes could be men and women; Edges indicate that a given man is married to a given woman. (Okay, in the real world it's more complex, but this is a classic "problem" I feel required to present...)

Solution: use ***backtracking*** and ***augmentation*** to solve the problem, which contributes to ***network flow problems***

Independent Set

Independent Set is a very general problem:

- Given a graph $G = (V, E)$, a set of nodes $S \subseteq V$ is ***independent*** if no two nodes in S are joined by an edge.
- Goal: Find an independent set that is as large as possible.
- Applicable to any problem where you are choosing a collection of objects and there are pairwise conflicts.



In this graph, the largest independent set is $\{1, 4, 5, 6\}$

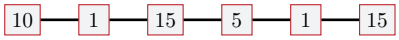
Independent Set

- Example: Each node is a friend, and each edge indicates a conflict between those two friends. Use Independent Set to find the largest group of people you can invite to a party with no conflicts.
- Interval Scheduling is a special case of Independent Set:
  - Define graph $G = (V, E)$ where V is the set of requests or intervals, and E is the set of edges that indicate conflicts between two requests.
- Bipartite Matching is a special case of Independent Set:
  - A little more complex than I want to explain in class; see the book and we'll cover it later.
- Solution: No efficient algorithm is known to solve this problem.
- However: If we're given an independent set for a given graph G, we can easily **check** that it's a correct answer.

Competitive Facility Location

This time, we have a two-player game.

- Dunkin Donuts puts a cafè at one location.
- Then Starbucks does.
- BUT! Cafès can't be too close (zoning requirement)
- Goal: Make your shops in the most convenient locations as possible.
- Model the problem:
  - Consider each location as a zone (rather than a point) that has an estimated value or revenue.
  - Model the problem as a graph: $G = (V, E)$ where V is the set zones as noted above, and E represents whether two zones are adjacent.
  - The zoning requirement says that the set of cafès is an independent set in G.



- Can't put a cafè in adjacent zones
- Can't put two cafès in one zone
- Edges indicate two zones are adjacent
- The set of cafès opened must be an independent set.

Competitive Facility Location

- Another question: Can we find a strategy such that Starbucks, no matter where Dunkin Donuts opens a cafè, can open cafès in locations with a total value of at least B?
- Even if I could give you a strategy, you'd have a hard time believing that the strategy is correct.
- This is in contrast to Independent Set!
- This problem is what we call a ***PSPACE-complete problem***
- Independent Set is a ***NP complete*** problem

Representative Problems: Summary

Interval Scheduling Solved easily with a greedy algorithm
Weighted Interval Scheduling Solved with dynamic programming
Bipartite Matching Solved with backtracking and augmentation
Independent Set No efficient approach to generate a solution, but it's easy to check a given solution
Competitive Facility Location No easy way to generate a solution, and NO EASY WAY to check a given solution

What is *efficient*?

- In general, it's pretty easy to come up with a ***brute force*** solution to a problem.
- For example, we can generate all possible solutions for a problem, and then check which one is correct (or acceptable).
- Definition attempt 1: An algorithm is efficient if it achieves qualitatively better worst-case performance, at an analytical level, than brute-force search.

- What's *"qualitatively better"*?
- Final definition: An algorithm is efficient if it has a polynomial running time.

Why does this matter?

- Algorithms are important
  - Many performance gains outstrip Moore's law: We can't always just throw hardware at the problem.
- Simple problems can be hard
  - Factoring, TSP
- Simple ideas don't always work
  - Nearest neighbor, closest pair heuristics
- Simple algorithms can be very slow
  - Brute-force factoring, TSP
- Changing your objective can be good
  - Guaranteed approximation for TSP
- And: for some problems, even the best algorithms are slow

Readings for NEXT week:

Rosen, Chapter 4.1, 4.2, 4.3, 4.4

Divisibility and Modular Arithmetic, Integer Representations and Algorithms,

Primes and Greatest Common Divisors, Solving Congruences Solving Congruences

9.6 Appendix: Logarithms

9.7 Logarithms

9.7.1 Definition

What is a logarithm?

$$\begin{aligned} b^x &= y \\ &\Updownarrow \\ \log_b(y) &= x \end{aligned}$$

We say "log base b of y equals x"

Once again,

b is called the *base*

x is called the *exponent*

Some Practice

$$\begin{aligned} \log_7(49) &= ? \\ &\Rightarrow 7^? = 49 \\ &\Rightarrow 7^2 = 49 \\ \Rightarrow \log_7(49) &= 2 \end{aligned}$$

Let's re-write this using the formula we have.

That let's us change the question to "49 is the 7 raised to what power?"

Or, "What is the exponent?"

9.7.2 Properties

Special Logs

- Base $b = 2$: **binary logarithm**, also referred to as $\lg x$
- Base $b = e$: **natural logarithm**, also referred to $\ln x$, where $e = 2.718.....$
  - The inverse of $\ln x$ is $\exp(x) = e^x$
    $\Rightarrow \exp(\ln x) = x$
- Base $b = 10$: The **common logarithm**, also referred to as $\log x$.
- If it's not one of these, the base is specified.

Restrictions

$\log_b(a)$ is only defined when $b > 1$ and $a > 0$.

Practice: Use what you know about exponents to convince yourself why this is true.

The Product Rule

$\log_a(xy) = \log_a(x) + \log_a(y)$

The logarithm of a product is the sum of the logs of its factors.

The Quotient Rule

$\log_a\left(\frac{x}{y}\right) = \log_a(x) - \log_a(y)$

The logarithm of a quotient is the difference of the logs of its factors.

The Power Rule

$\log_a(x^y) = y\log_a(x)$

When the term of a logarithm has an exponent, it can be pulled out in front of the log.

Change of Base Rule

$\log_a b = \frac{\log_c b}{\log_c a}$

9.7.3 Logarithm Exercises

2. solve these.

(a) $\frac{3+\log_7 x}{2-\log_7 x} = 4 \quad x > 0$

$$\begin{aligned} 3 + \log_7 x &= 8 - 4\log_7 x \\ 5\log_7 x &= 5 \\ \log_7 x &= 1 \\ x &= 71 = 7 \\ k &= \{7\} \end{aligned} \tag{9.45}$$

(b)

$$\frac{5+\log x}{3-\log x} \quad x > 0$$

$$2 = 8$$

$$x > 0$$

$$\begin{aligned} \tfrac{5+\log x}{3-\log x} &= 3 \\ 5 + \log x &= 9 - 3\log x \\ 4\log x &= 4 \\ \log x &= 1 \\ x &= 10^1 = 10 \\ K &= \{10\} \end{aligned} \tag{9.46}$$

(c) $\log_4\left(x^2-9\right)-\log_4(x+3)=3$

$$
\begin{aligned}
l\log_4\left(x^2-9\right)-\log_4(x+3)=3 \quad x>3\wedge x>-3\Rightarrow x\in(3;00)\\
\log_4\left(x^2-9\right)-\log_4(x+3)&=\log_4 64\\
\log_4\frac{x^2-9}{x+3}&=\log_4 64\\
\frac{x^2-9}{x+3}&=64\\
\frac{(x-3)(x+3)}{x+3}&=64\\
x&=67\in(3,00)\\
x&=\{67\}
\end{aligned}
$$

Summary

The Product Rule	$\log_a(xy)=\log_a(x)+\log_a(y)$
The Quotient Rule	$\log_a\left(\frac{x}{y}\right)=\log_a(x)-\log_a(y)$
The Power Rule	$\log_a(x^y)=y\log_a(x)$
The Change of Base Rule	$\log_a b=\frac{\log_c b}{\log_c a}$

9.7.4 Relevance of Logs

9.7.4.1 Logs and Binary Search

Logs and Binary Search

TODO: Something about in asymptotic analysis log base 10 is equivalent to log base 3 or whatever.

Binary search is $O(\log n)$.

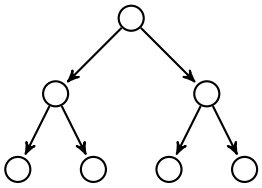
Given a telephone book of n names:

- Looking for person p
- Compare p to the person in the middle, or the $\left(\frac{n}{2}\right)^{nd}$ name
- After one comparison, you discard $\frac{1}{2}$ of the names in this book.
- The number of steps the algorithm takes = the number of times we can halve n until only one name is left.
  $\Longrightarrow \log_2 n$ comparisons
- In this case, $x=?$, $y=n$, and $b=2$

9.7.4.2 Logs and Trees

Logs and Trees

A binary tree of height 2 can have up to 4 leaves:



What is the height h of a binary tree with n leaf nodes?

For n leaves, $n = 2^h$

$\Rightarrow h = \log_2 n$

9.7.4.3 Logs and Bits

Logs and Bits

Let's say we have 2 bit patterns of length 1 (0 and 1),
and 4 bit patterns of length 2 (00, 01, 10, 11).

How many bits w do we need to represent any one of n different possibilities, either one of n items, or integers from 1 to n?

- There are at least n different bit patterns of length w
- We need at least w bits where $2^w = n$
  $\Rightarrow w = \log_2 n$ bits

Takeaway

Logs arise whenever things are repeatedly halved or doubled.

9.8 Appendix: Selection Sort, graphically