CMSC351 - Section 0201

Ash Dorsey
ash@ash.lgbt

Contents

Dynamic Programming

- Not dynamic
- Useful for interviews because it's a popular form of question
- Useful concept

Fibonacci sequence

$F(n) = F(n-1) + F(n-2)$

Sadly, this algorithm is terrible algorithmly ($O(2^n)$). Fix it with dynamic programming!

Use caching to resolve this, store previous results in an array, and reuse them.

Coin Changing

Given a set of types of coins, how many of each coin does it take to make change for some value n.

$c = [1, 5, 10], n = 27 = 2 \times 10 + 1 \times 5 + 2 \times 1$

a. **How can we do this if we don't care about how many coins?**

   Just use a bunch of pennies. $n \Rightarrow [n, 0, 0]$

b. **How do we do this to minimize coins?**

c. **How many different ways can be made change?**

Aside: what coins work? You can only do so if they are relatively prime. $\underbrace{[5, 10]}_{\text{doesn't work}}, [2, 3], [3, 7]$

a. **Greedy Algorithm**

Greedy algorithms exploit a simple heuristic to basically brute force the answer.

In the context of coins, you subtract out the largest coin until you can't and then repeat for the next coin.

$$\begin{aligned} c = [1, 5, 10], n = 27 \\ 27 \to 17 \to 7 \to 2 \to 1 \to 0 \end{aligned}$$

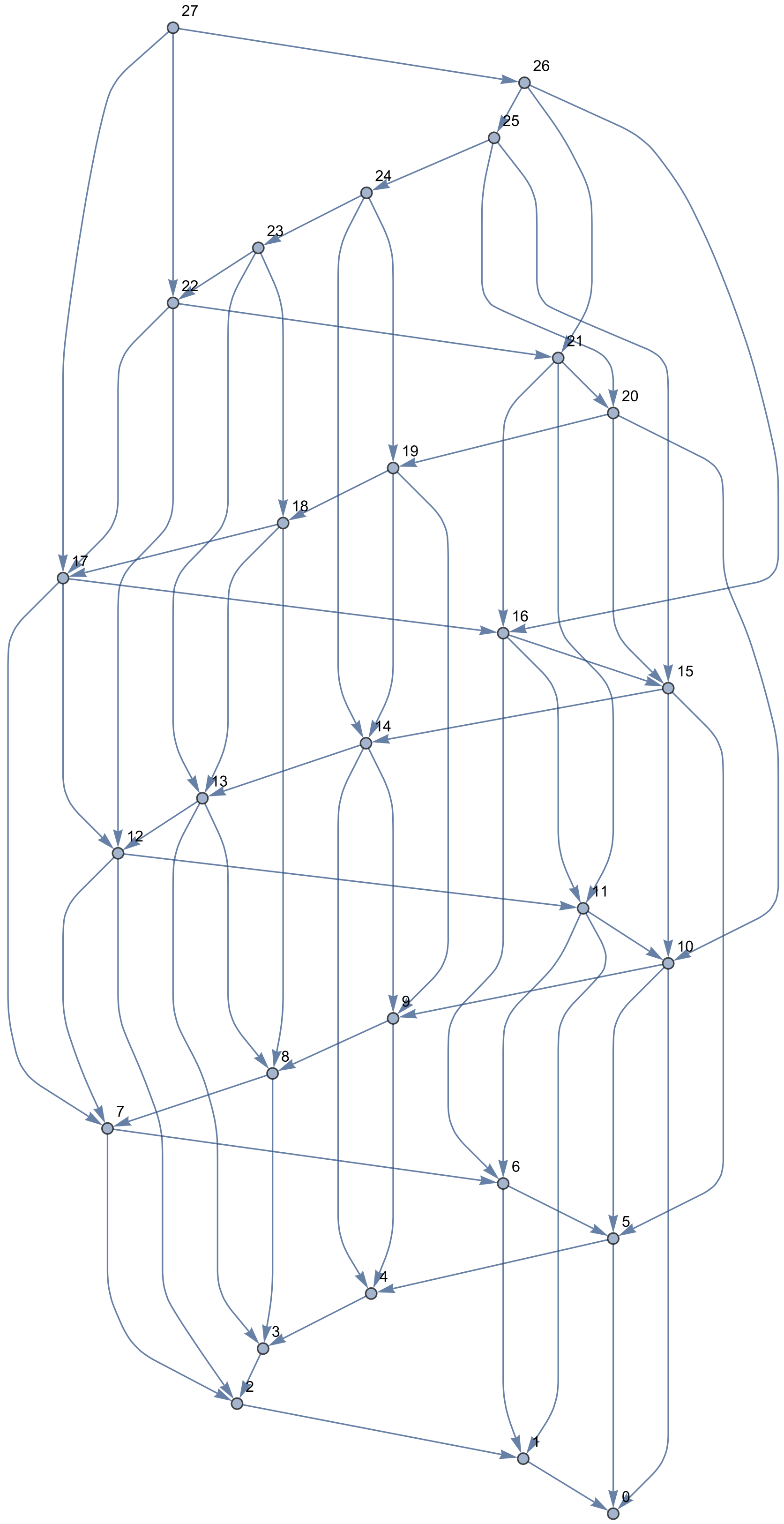
$$\begin{aligned} c = [1, 10, 25], n = 30 & \\ 30 \to 5 \to 4 \to 3 \to 2 \to 1 \to 0 & \qquad \text{6 coins with the greedy algorithm} \\ 30 \to 20 \to 10 \to 0 & \qquad \text{but you could use 3 coins with dimes} \end{aligned}$$

When you are thinking about the answer to a problem, don't start with a preconception of the problem, just start playing with the answers.

Example:

$$c = [1, 5, 10], n = 27$$
$$27 \to 17 \to 7 \to 2 \to 1 \to 0$$

Graph of options available:



So this is an optimal algorithm:

```
def coin_change(n: int, c: list[int]) -> list[int]:
  if n == 0:
    return []
  else:
    results = {}
    for coin in c:
      if n - coin >= 0:
        results[coin] = coin_change(n - coin, c)
    optimal = min(results.items(), key=lambda x: len(x[1]))
    optimal[1].append(optimal[0])
    return optimal[1]
```

But this is a terrible algorithm for complexity! Use dynamic programming and cache it:

```
A = [] # cache
A.append(0)
C = [1,5,10] # coin denominations
for n in range(1, x):
  howmanycoins = float("inf")
  for coin in C:
    if n - coin >= 0:
      howmanycoins = min(howmanycoins,1+A[n-coin])
  A.append(howmanycoins)
```

Big O Notation

Imagine you are manning a desk, checking people in. If one more person shows up, you have to do one more thing. If you required everyone to introduce themselves to everyone else, every additional person requires there to be far more work done.

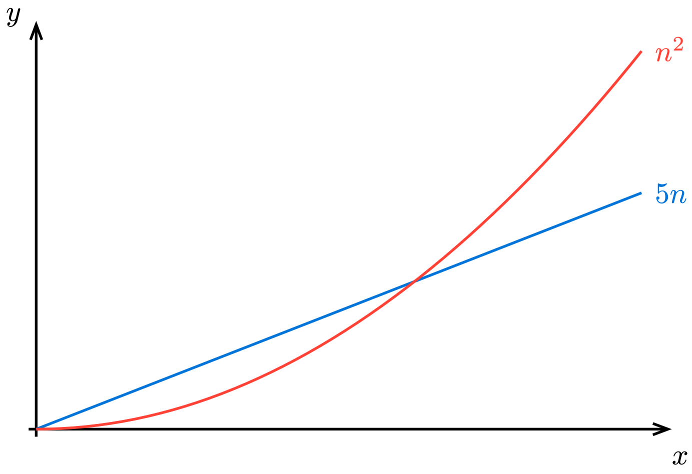
Suppose you have an algorithm that processes a list of n elements. We might want to know:

- How does the time of this algorithm change as we add elements?
  - It probably goes up
- How does memory usage change with more elements?
  - It probably also goes up

You can often get quicker algorithms with more memory.

Suppose we have two algorithms:

- Algorithm 1 takes $5n$ time
- Algorithm 2 takes n^2 time



Big O

As n gets larger, is there some function that's always above it (after a cutoff)?

Definition

$$f(x) = O(g(x)) \text{ if } \exists x_0, c > 0 \text{ such that } \forall x \geq x_0, f(x) \leq cg(x)$$

Examples

You can just say a bigger function, but it's probably unhelpful:

$$x = O(x^2) \to x \in O(x^2), x \in O(x^3), O(x^2) \subset O(x^3)$$

Show $x^2 - 1 = O(x^2)$:

$$\begin{gathered} x^2 - 1 \leq cx^2 \\ x^2 - 1 \leq 10x^2 \qquad \text{Let } c = 10 \\ -1 \leq 9x^2 \\ -\frac{1}{9} \leq x^2 \therefore \text{true} \because x^2 \geq 0, x \in \mathbb{R} \end{gathered}$$

Show that $2x^2 + 1 = O(x^2)$:

$$\begin{gathered} 2x^2 + 1 \leq cx^2 \\ 2x^2 + 1 \leq 11x^2 \qquad \text{Let } c = 11 \\ 1 \leq 9x^2 \\ \frac{1}{9} \leq x^2 \\ -\frac{1}{3} \leq x \leq \frac{1}{3} \\ \therefore \text{ for } x > \frac{1}{3}, \exists c > 0, 2x^2 \leq cx^2 \therefore 2x^2 + 1 = O(x^2) \end{gathered}$$

Show that $4x^2 + x\lg(x) - 1 = O(x^2)$:

$$
\begin{gathered}
4x^2 + x\lg(x) - 1 \leq cx^2 \\
4x^2 + x\lg(x) - 1 \leq 5x^2 \qquad \text{Let } c = 5 \\
x\lg(x) - 1 \leq x\lg(x) \leq x^2 \\
\lg(x) \leq x \qquad \text{Positive } x \\
\therefore 4x^2 + x\lg(x) - 1 = O(x)
\end{gathered}
$$

Show that $x^2\lg(x) + \frac{100}{x^2} = O(x^3)$:

$$
\begin{gathered}
x^2\lg(x) + \frac{100}{x^2} \leq cx^3 \\
x^2\lg(x) + \frac{100}{x^2} \leq x^3 + \frac{100}{x^2} \leq cx^3 \\
\frac{100}{x^2} \leq (c-1)x^3 \\
100 \leq (c-1)x^5 \\
\sqrt[5]{\frac{100}{c-1}} \leq x \\
\sqrt[5]{\frac{100}{\frac{1}{1000}}} \leq x \qquad \text{Let } c = \frac{1}{1000} + 1 \\
\sqrt[5]{100000} \leq x \\
10 \leq x \\
\therefore \text{ for } x > 10, \exists c > 0, x^2\lg(x) + \frac{100}{x^2} \leq cx^3 \therefore x^2\lg(x) + \frac{100}{x^2} = O(x^3)
\end{gathered}
$$

$O(\text{whatever})$ always has the largest term and no constants (by convention).

For example, $O(n), O(nc), O(1), O(\lg(n)), O(n), O(n\lg(n)), O(n^2), O(n^x)$,$O(2^n), O(n!)$

Question

If my function is monotonically increasing, do I still need x_0? What about if it is monotonic and passes through the origin?

For the first one, yes, because $x^2 + 1$ is increasing but no factor of c can make $(0)^2 + 1 < c(0)^2 \Rightarrow 1 < 0 \therefore$ false.

For the second one, I believe no. (he says yes, seemingly without example).

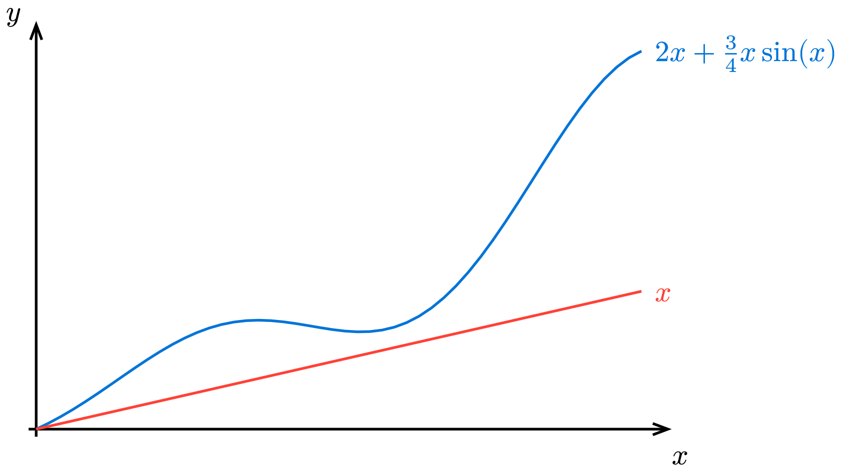
Big Ω

Bounding from below! Entirely the opposite of Big O.

Definition

$$f(x) = \Omega(g(x)) \text{ if } \exists x_0, B > 0 \text{ such that } \forall x \geq x_0, f(x) \geq Bg(x)$$

Examples



Show $3n\lg n - n + 1 = \Omega(n\lg n)$**:**

$$
\begin{gathered}
3n\lg n - n + 1 \geq B(n\lg n) \\
3n\lg n - n + 1 \geq 3n\lg n - n \geq B(n\lg n) \\
n(3\lg n - 1) \geq B(n\lg n) \\
3\lg n - 1 \geq B(\lg n) \\
-1 \geq (B-3)(\lg n) \\
1 \leq (3-B)(\lg n) \\
\frac{1}{3-B} \leq \lg n \\
2^{\frac{1}{3-B}} \leq 2^{\lg n} \\
2^{\frac{1}{3-B}} \leq n \qquad (B = 2) \\
2^{\frac{1}{1}} \leq n \\
2 \leq n \\
\therefore x_0 = 2, B = 2, 3n\lg n - n + 1 = \Omega(n\lg n)
\end{gathered}
$$

Show $n^2 - n = \Omega(n^2)$**:**

$$
\begin{gathered}
n^2 - n \geq Bn^2 \\
n - 1 \geq Bn \\
-1 \geq (B-1)n \\
1 \leq (1-B)n \\
\frac{1}{1-B} \leq n \qquad \left(B = \frac{1}{2}\right) \\
2 \leq n \\
\therefore x_0 = 2, B = \frac{1}{2}, n^2 - n = \Omega(n^2)
\end{gathered}
$$

Big Θ

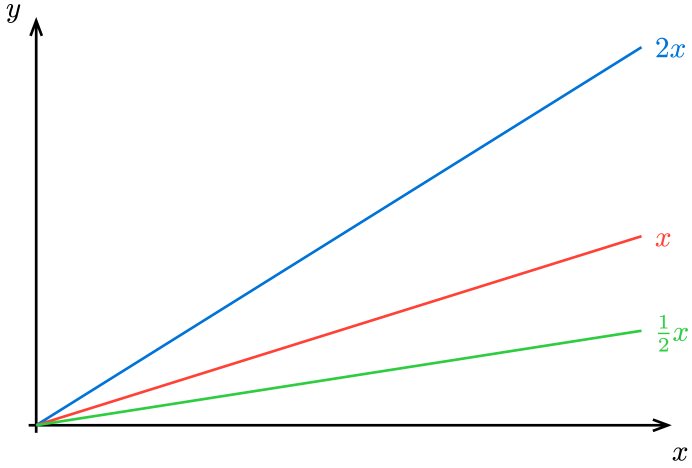
Defintion

$$f(x) = \Theta(g(x)) \text{ if } \exists x_0, b > 0, c > 0, \text{such that } \forall x \geq x_0, bg(x) \leq f(x) \leq cg(x)$$

Examples

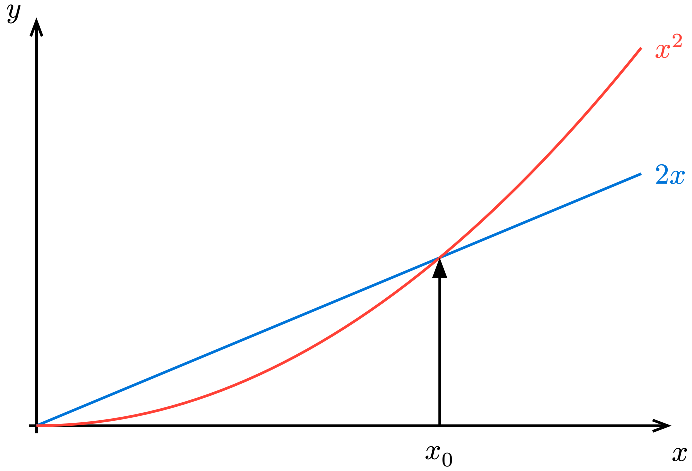
Is $f(x) = x \in \Theta(x)$

$b = \frac{1}{2} \land c = 2 \rightarrow \frac{1}{2}x \leq x \leq 2x \therefore f(x) \in \Theta(x)$



Jumping Back

What is x_0, and why is it considered irrelevant?



x_0 typically happens way earlier than is a useful, so algorithms with better complexity are usually just better because you run them on large amounts of data.

Using Limits to Calculate Big - _ notation

Theorem

$$\text{If } \lim_{x\to\infty} \frac{f(x)}{g(x)} \neq \infty, g(x) \text{ must grow faster or equally as fast.} \therefore f(x) = O(g(x))$$

$$\text{If } \lim_{x\to\infty} \frac{f(x)}{g(x)} \neq 0, g(x) \text{ must grow slower or equally as fast} \therefore f(x) = \Omega(g(x))$$

$$\text{If } \lim_{x\to\infty} \frac{f(x)}{g(x)} \neq 0 \land \lim_{x\to\infty} \frac{f(x)}{g(x)} \neq \infty, g(x) \text{ must be the same order as } f(x) \therefore f(x) = \Theta(g(x))$$

Examples

$$f(x) = x, g(x) = x^2$$

$$\lim_{x\to\infty} \frac{x}{x^2} = \lim_{x\to\infty} \frac{1}{x} = 0 \therefore f(x) = O(g(x))$$

$$f(x) = 3x, g(x) = 2x$$

$$\lim_{x\to\infty} \frac{3x}{2x} = \lim_{x\to\infty} \frac{3}{2} = \frac{3}{2} \therefore f(x) = \Theta(g(x))$$

Intution

If you have a function that is a combination of “nice” functions, the biggest term will be the big Θ

Nice Functions In Increasing Order

$$1, \lg(n), n, n\lg(n), n^2, n^2\lg(n), ..., 2^n, 3^n, ..., n!$$

Examples

$$3n^2 + 7n^2\lg(n) + 10\lg(n) - 100n = \Theta(n^2\lg(n))$$

$$3^n + 5\lg(n) + 100n^{100} = \Theta(3^n)$$

But if you don’t have nice functions, things could go wrong.

$$2 + \sin(x) = \Theta(1) \qquad 1 \leq 2 + \sin(x) \leq 3$$

$$x(2 + \sin(x)) = \Theta(x) \qquad 1 \leq 2 + \sin(x) \leq 3 \Rightarrow x \leq x(2 + \sin(x)) \leq 3x$$

Big Theta Assumptions

$$f(x) = \Theta(g(x)) \iff f(x) = O(g(x)) \land f(x) = \Omega(g(x))$$

False statements:

$$f(x) = O(g(x)) \Rightarrow f(x) = \Theta(g(x))$$

$$f(x) \neq O(g(x)) \Rightarrow f(x) = \Omega(g(x))$$

Analysing the Time Complexity of Pseudocode

What should you care about when analyzing code?

Program	Time

`let mut sum = 0;`
`for i in 1..=n {`
	`c1`
	`c2 per iteration`

`  sum = sum + i;`
`}`
	`c3`
	`(part of c2)`

If we care about the **exact** time this program takes, then we need to consider *every* part:

$$T(n) = c_1 + c_n n + c_3 n$$

Some things you can ignore

But what if we only care about Θ, O, or Ω? You can ignore some things.

1. If the body of the loop takes any time, you can ignore the maintenance lines (iteration):

$$T'(n) = c_1 + c_3 n = \Theta(n)$$

But you *do* have to care if there is nothing there:

```
for i in 0..n {}
```

Still $O(n)$ time.

2. Constant time operations that are adjacent to things that take any time can be ignored.

$$T'(n) = c_3 n = \Theta(n)$$

Conditionals

```
if n > 10 {
  println!("Hi!")
}
```

$\rightarrow \Theta(1)$ But be careful, conditionals can split the upper and lower bounds:

```
if x < y{
  for i in 1..=n {
    println!("Friday!")
  }
}
```

If $x < y \rightarrow \Theta(n)$, if $x \geq y \rightarrow \Theta(n)$. Then, for the entire thing, $T(n) = O(n)$, $T(n) = \Omega(1)$, $T(n) = \Theta(\text{doesn't exist})$

Best, worst and average case

You cannot control n. Given an n

- Best case: what is the fastest it can run?
- Worst case: what is the slowest it can run?

Example

```
fn example<T, const N: usize>(A: [T; N]) {
  let mut i = 0;
  while i < N && A[i] != 0 {
    println!("35!");
    i += 1;
  }
}
```

The best case is when $A[0] = 0 \Rightarrow \Theta(1)$ The worst case is when $A[i] \neq 0 \forall i \Rightarrow \Theta(n)$ Average case: Difficult to define what average is.

(I believe with the assumption that it is randomly picking from {0, 1} for each element, $\Theta(1)$, but I have not really proved it.)

Review the Limit Theorem

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} \neq \infty \Rightarrow f(x) = O(g(x))$$

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} \neq 0 \Rightarrow f(x) = \Omega(g(x))$$

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} \neq 0 \land \lim_{x \to \infty} \frac{f(x)}{g(x)} \neq \infty \Rightarrow f(x) = \Theta(g(x))$$

Example

Show $x^2 - 1 = \Theta(x^2)$

$$\lim_{x \to \infty} \frac{x^2 - 1}{x^2} = \lim_{x \to \infty} 1 - \frac{1}{x^2} = 1 - \lim_{x \to \infty} \frac{1}{x^2} = 1 - 0 = 1 \therefore x^2 - 1 = \Theta(x^2)$$

Show $n^{10} = O(2^n)$

$$\lim_{n \to \infty} \frac{n^{10}}{2^n} \underbrace{=}_{\text{L'Hôpital's rule}} \lim_{n \to \infty} \frac{10n^9}{\ln 2(2^n)} = \lim_{n \to \infty} \frac{10 \cdot 9 \cdot n^8}{(\ln 2)^2 (2^n)} = \cdots = \frac{10!}{(\ln 2)^{10} 2^n} = 0$$

$$\lim_{n \to \infty} \frac{n^{10}}{2^n} \neq \infty \therefore n^{10} = O(2^n)$$

Misconceptions

Best, average, and worst cases are unrelated to O, Θ, Ω.

$$\begin{cases} x^2 = O(x^4) \\ x^2 = O(x^3) \\ x^2 = O(x^2) \\ x^2 = \Theta(x^2) \\ x^2 = \Omega(x^2) \\ x^2 = \Omega(x) \\ x^2 = \Omega(1) \end{cases}$$

Maximum Contiguous Sum

Given a list A of numbers, find the maximum contiguous sum. This is the maximum sum of a sublist of A, or $\sum_{j=0}^{j_f} = A[k+j]$, where $1 \leq j_f \leq n, 0 \leq k + j_f < n$. For this example, you *must* take at least 1 element.

Solutions

Bruteforce

The easiest solution is to try every single sublist.

$$\begin{bmatrix} -3 & 6 \end{bmatrix}$$

Python

```python
# A is a list of length n
max = A[0]
for i in range(0, n):
  sum = 0
  for i in range(1, n):
    sum = sum + A[j]
    if sum > max:
      max = sum
# max is the maximum continuous sum
```

Take this subpart as taking constant time c:

```python
sum = sum + A[j]
if sum > max:
  max = sum
```

Time complexity:

$$\sum_{j=1}^{n} \mathbb{c} = n\mathbb{c}, \sum_{j=0}^{n} \mathbb{c} = (n+1)\mathbb{c}, \qquad \underbrace{\sum_{i=0}^{n} = \frac{n(n+1)}{2}}_{\text{apparently an easy sum to derive}}$$

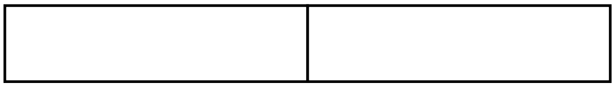
$$T(n) = \sum_{i=0}^{n-1}\sum_{j=i}^{n-1} \mathbb{c}$$

$$\begin{aligned}
T(n) &= \sum_{i=0}^{n-1}\left(\mathbb{c}\left(\underbrace{(n-1)}_{\text{top of sum}} - \underbrace{(i)}_{\text{bottom of sum}} + \underbrace{1}_{\text{sums are inclusive and "horrible"}}\right)\right) \\
&= \sum_{i=0}^{n-1}(\mathbb{c}(n-i)) = \mathbb{c}\sum_{i=0}^{n-1}(n-i) = \mathbb{c}\left(\sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i\right) \\
&= \mathbb{c}\left(n\sum_{i=0}^{n-1} 1 - \sum_{i=0}^{n-1} i\right) \\
&= \mathbb{c}\left(n((n-1)-(0)+1) - \sum_{i=0}^{n-1} i\right) \\
&= \mathbb{c}\left(n \cdot n - \sum_{i=0}^{n-1} i\right) \\
&= \mathbb{c}\left(n^2 - \sum_{i=0}^{n-1} i\right) \\
&= \mathbb{c}\left(n^2 - \sum_{i=0}^{n-1} i\right) \\
&= \mathbb{c}\left(n^2 - \frac{(n-1)n}{2}\right) \\
&= \mathbb{c}\left(n^2 - \left(\frac{n^2}{2} - \frac{n}{2}\right)\right) \\
&= \mathbb{c}\left(\frac{n^2}{2} + \frac{n}{2}\right) \\
&\vdots \\
&= \Theta(n^2)
\end{aligned}$$

In the end, $\Theta(n^2)$ isn't great, but also isn't awful. Let's seek a better algorithm though.

Divide and Conquer

Break the list down into two sublists and recurse.



You must also check the middle, though, because the answer might straddle it.

Base case: with a list of length 1, the maximum sum is just that element.

The $\text{maximum sum} = \max(\text{sum from left}, \text{sum from right}, \text{middle sublist})$

```
def mcm(A, L, R):
  if L == R:
    return A[L]
  else:
    C = (L+R) // 2
    Lmax = mcm(A, L, C)
    Rmax = mcm(A, C+1, R)
    // straddle max calculation
    Lhmax = float("-inf")
    Lhsum = 0
    for i in range(C, L + 1, -1):
      Lhsum = Lhsum + A[i]
      if Lhsum > Lhmax:
        Lhmax = Lhsum

    Rhmax = float("-inf")
    Rhsum = 0
    for i in range(C+1, R+1):
      Rhsum = Rhsum + A[i]
      if Rhsum > Rhmax:
        Rhmax = Rhsum
    Smax = Lhmax + Rhmax
    return max([Lmax, Rmax, Smax])
```

At level 0, I do $n\mathbb{c}$ work.

When I recurse, I do $\frac{n\mathbb{c}}{2} + \frac{n\mathbb{c}}{2}$ work.

When I recurse, I do $\frac{n\mathbb{c}}{4} + \frac{n\mathbb{c}}{4} + \frac{n\mathbb{c}}{4} + \frac{n\mathbb{c}}{4}$ work.

…

So, at each level, you do $n\mathbb{c}$ work.

Let $k =$ be the level. $\frac{n\mathbb{c}}{2^k}$ is the amount of work per level. Then, to find the max level, $\frac{n\mathbb{c}}{2^k} = \mathbb{c} \Rightarrow \lg(n) = k$.

$n\mathbb{c} = \lg(n) = \Theta(n\lg(n))$

Questions

What happens to the divide and conquer algorithm if the list is positive?

The straddle sum of the top level will “win” in the end.

What if the list is all negative?

The singular minimum element will win.

Maximum Contiguous Sum

We’ve seen brute force: $\Theta(n^2)$ and divide and conquer: $\Theta(n\lg n)$

Dynamic Programming

$$\begin{bmatrix}5 & 7 & -10 & 3 & 1 & 2 & 4\end{bmatrix}$$

If we're next to a positive number, you always want to take it $\therefore$. If there are no negative numbers, it's the whole list; positive numbers are grouped.

You want to take the highest (least negative) number if everything is negative.

$$[5 \;\; 10 \;\; -100 \;\; 10 \;\; 7] \qquad \underbrace{\Rightarrow}_{\text{max sum so far (left to right)}} \qquad [5 \;\; 15 \;\; 15 \;\; 15 \;\; 17]$$

Kadane's Algorithm

For a list A and an index i, define maximum contiguous sum ending at i that goes up to and **includes** i**.**

$$A = [4 \;\; -8 \;\; 6 \;\; -1 \;\; 3 \;\; 1 \;\; 5 \;\; 9]$$
$$B_{\text{Max Sum}} = [4 \;\; -4 \;\; 6 \;\; 5 \;\; 8 \;\; 9 \;\; 14 \;\; 23]$$

The MCS (Maximum Contiguous Sum) ending at index i is $B_{\text{Max Sum}}[i] = \max(A[i], A[i] + B_{\text{Max Sum}}[i - 1])$

Canonical Form

(remember that I am translating the pseudocode into python because I like it more, but that means $n \notin \text{range}(1, n)$)

```
max_overall = A[0]
max_ending_at_i = A[0]
for i in range(1, n):
  max_ending_at_i = max(max_ending_at_i + A[i], A[i])
  max_overall = max(max_ending_at_i, max_overall)
```

More Intuitive (but less efficient)

This solution uses additional memory to store the B list, but it matches better with the definition.

```
def mcs(A: list):
  n = len(A)
  B = [None] * n
  B[0] = A[0]
  for i in range(1, n):
    B[i] = max(A[i], A[i] + B[i-1])
  return max(B)
```

128/3

General Types of Algorithms

Greedy Algorithms: Generally grabs the most obvious good-looking solution. Usually not optimal.

Example: A chicken trying to get somewhere will not backtrack and, therefore, can't figure out how to go around a fence.

Dynamic: A technique where we cache results as we go.

Divide and conquer: Divide a list in two and recurse.

Sorting Algorithms

Properties of such

1. In place (boolean): If it is, the sorts the values within in the original array
   - Shifting around elements within the original array is in place
   - If you create a new array to manipulate, that is in place
2. Auxiliary space: Other than the list itself, how much extra space do we use?
3. Stability: is the order of identical elements preserved:
   - [3, 2, 1, 2] → {[1, 2, 2, 3], [1, 2, 2, 3]} (the first example is stable)
   - useful when sorting by one thing, then another

Bubble sort

One of the simplest sorting algorithms, but it's pretty terrible. Be aware, there are many different implementations. We will talk about the worst one.

Pass through the list from left to right, comparing adjacent elements. If they are out of order, swap the elements. Repeat the process until everything is sorted.

Example

$$\begin{gathered}
[[2, 11], 5, 8, 0] \\
[2, [11, 5], 8, 0] \\
[2, 5, [11, 8], 0] \\
[2, 5, 8, [11, 0]] \\
[[2, 5], 8, 0, 11] \\
[2, [5, 8], 0, 11] \\
[2, 5, [8, 0], 11] \\
[[2, 5], 0, 8, 11] \\
[2, [5, 0], 8, 11] \\
[2, 0, 5, [8, 11]] \\
[[2, 0], 5, 8, 11] \\
[0, 2, 5, 8, 11] \\
\text{sorted}
\end{gathered}$$

Pseudocode

```
for i in range(n-1):
  for j in range(n-i-1):
    if A[j] > A[j+1]:
      # below, swap A[j] and A[j + 1]
      (A[j], A[j+1]) = (A[j+1], A[j])
```

Time Complexity

$$T(n) = \sum_{i=0}^{n-2} \sum_{i=0}^{n-i-2} c = \Theta(n^2)$$

Best, worst, and average case: $\Theta(n^2)$

If you make a slightly smarter algorithm and have a flag where it detects the list is sorted by checking if everything has been correct in an iteration, then the best case is $\Theta(n)$ (it's still terrible).

Properties

In place, auxiliary memory of $\Theta(1)$, stable.

Selection sort

Idea: look through the list, find the index of the smallest item, and then swap $A[\text{that index}]$ with $A[\text{first unsorted index}]$. Repeat.

Example

$$
\begin{gathered}
[[5,2,1,3,4]] \\
[1,[2,5,3,4]] \\
[1,2,[5,3,4]] \\
[1,2,3,[5,4]] \\
[1,2,3,4,5] \\
\text{sorted: this may look linear, but finding the minimum is } O(n) \text{ work}
\end{gathered}
$$

Pseduocode

```
for i in range(n - 1):
  minindex = i
  for j in range(i+1, n):
    if A[j] < A[minindex]:
      minindex = j
  (A[i], A[minindex]) = (A[minindex], A[i]) # swap
# A is sorted
```

Time Complexity

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} c = \sum_{i=0}^{n-2} (c((n-1)-(i+1)+1)) = \sum_{i=0}^{n-2} (c(n-i-1) = c\sum_{i=0}^{n-2}(n-i-1)$$

$$= c\left(\sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i\right) = c\left((n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-3)(n-2)}{2}\right)$$

$$= c\left((n-1)(n-2+1) - \frac{(n-3)(n-2)}{2}\right) = c\left((n-1)(n-1) - \frac{(n-2)(n-1)}{2}\right) = \cdots = \Theta(n^2)$$

Properties

In place, uses auxiliary memory of $\Theta(1)$, is not stable.

Insertion Sort

Maintain a sorted area in the front of the list and add new elements to it.

Example

$$[3, 4, 1, 3, 2]$$

$$\text{the underline indicates the sorted section:}$$

$$[\underline{3}, 4, 1, 3, 2]$$

$$[\underline{3,\ 4}, 1, 3, 2]$$

$$[\underline{1,\ 3,\ 4}, 3, 2]$$

$$[\underline{1,\ 3,\ 3,\ 4}, 2]$$

$$[\underline{1,\ 2,\ 3,\ 3,\ 4}]$$

Pseduocode

```
for i in range(1, n):
  key = A[i]
  j = i - 1
  while j >= 0 and key < A[j]:
    A[j+1] = A[j]
    j = j - 1
  A[j + 1] = key
```

Time Complexity

Best case: $\Theta(n)$, already sorted list.

$$T(n) = \sum_{i=1}^{n-1}(c_1 + c_2 + c_4) = \Theta(n)$$

Worst case: $\Theta(n^2)$, if the list is reverse sorted.

$$T(n) = \sum_{i=1}^{n-1}(c_1 + i(c_2 + c_3) + c_2 + c_4) = \Theta(n^2)$$

Average case:

What does average mean?

Colloquially, shuffle the list a bunch and see how long it takes.

For a list A, a pair of indicies $(i, j), i < j$ but $A[i] > A[j]$ is called an inversion.

Example:

$$[\underline{4}, 3, 1, 2, 5]$$

$$\text{Inversions: } (0, 1), (0, 2), (0, 3), (1, 2), (1, 3)$$

How many possible inversions are there: $\frac{n(n-1)}{2}$

$T(n) = (n-1)(c_1 + c_4) + \underbrace{I}_{\text{inversions}}(c_2 + c_3) + (n-1)(c_2)$

Best case: $I = 0 \Rightarrow T(n) = \Theta(n)$

Average case: $I = \frac{n(n-1)}{4} \Rightarrow T(n) = \Theta(n^2)$

Worst case: $I = \frac{n(n-1)}{2} \Rightarrow T(n) = \Theta(n^2)$

Properties

Stable sort, $\Theta(n^2)$ average complexity, auxiliary memory of $O(1)$, in place.

Overall comments

- get some sleep (staying up all night is unhelpful)
- review homework, in-class problems, and the sample exams
- look through notes
- review 250 material (bayes theorem apparently?)
  - Sums are important and relevant (you only need to know like 4)
- budget time while taking the exam (if you're not confident about something or think it will take too long, just skip it and come back to maximize your score)
- understand
  - psuedocode
  - how algorithms work at a high level
  - run time of algorithms
- this exam is apparently "uninteresting"

Coin-changing

Greedy approach

Not always optimal for the number of coins

Smarter algorithm

Gives an ideal solution by using dynamic programming. Apparently, the hardest question will use this as a stepping stone, so probably study up

Big - $\{\Omega.O, \Theta\}$

Know the definition of the Big - _'s

$$\exists x_0, c > 0, \text{such that if } x \geq x_0 \text{ then } f(x) \leq cg(x)$$

- Proof using definition
  - You'll need to provide the c and x_0

Limit theorems

$$\text{If} \lim_{x\to\infty} \frac{f(x)}{g(x)} \neq \infty, g(x) \text{ must grow faster or equally as fast.} \therefore f(x) = O(g(x))$$

$$\text{If} \lim_{x\to\infty} \frac{f(x)}{g(x)} \neq 0, g(x) \text{ must grow slower or equally as fast} \therefore f(x) = \Omega(g(x))$$

$$\text{If} \lim_{x\to\infty} \frac{f(x)}{g(x)} \neq 0 \land \lim_{x\to\infty} \frac{f(x)}{g(x)} \neq \infty, g(x) \text{ must be the same order as } f(x) \therefore f(x) = \Theta(g(x))$$

Therefore, you will need to know your (basic) derivatives for L'hopital's rule.

Remember, things can have multiple big O's or big Ω's.

Make pseudocode into exact runtimes.

Maximum contiguous sum

- Understand problem
- Bruteforce solution ($\Theta(n^2)$)
- Divide and conquer ($\Theta(n \lg n)$)
- Kadane's Algorithm ($\Theta(n)$)

Sorts

Bubble sort

- intuition
- Algorithm $\Theta(n^2)$ in all cases

Selection sort

- Same as bubble sort, but not stable

Design of exam:

- True/False questions
- procedural question
- "thinky" problem
- procedural question
- design your own algorithm

median will be ~83%, apparently

Example problem

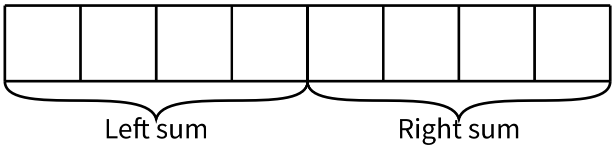
Prove from definition:

$$x^2 + x = O(x^2)$$

First step: write it out:

$$\begin{gathered} x^2 + x \leq cx^2 \\ x \leq (c-1)x^2 \\ 1 \leq (c-1)x \\ \frac{1}{c-1} \leq x \\ \text{Let } c = 2: \\ \frac{1}{2-1} \leq x \\ 1 \leq x \\ \therefore x_0 = 1, c = 2 \end{gathered}$$

Divide and conquer



Then, the total MCS = max(left, straddle, right), where straddle is all the sums that include the center element.

There is a great diagram in Max's notes explaining one array $[5, -2, 7, 9, -1, -3, 2]$. Perhaps try manually doing all the algorithms on this array and then check with his work.

silence for the first 7 minutes of class and then! Movement! The projector turns on, and written on the piece of paper displayed is the word "today"...

Exam Stuff

Pretty normal exam, but the odd one out was the bubble sort question.

Weird Bubble Sort

You can swap by pairs, and it is always faster because it only has to touch half the elements.

The Last Question

It was secretly Fibonacci in disguise. (I didn't notice this somehow but I think I got it right regardless)

```
def stairs(n):
  if n == 1:
    return 1
  if n == 2:
    return 2
  else:
    return stairs(n-1) + stairs(n-2)
```

```
cache = {}
def stairs(n):
  if n in cache:
    return cache[n]
  if n == 1:
    return 1
  if n == 2:
    return 2
  cache[n] = stairs(n-1) + stairs(n-2)
  return cache[n]
```

Binary Search

We are given a list of size n and want to know if a given element is in the list.

Bruteforce

Look at all the elements. $\Theta(n)$

Binary Search

If the list is sorted, we can take advantage of the structure of the list.

If you're doing a bunch of lookups, sorting is a one-time cost. You can also insert elements into a sorted list, maintaining the sorted order.

Look at the center: If we find it, great! Else, if it's less than the target, go up, and if it's more than the target, go down.

$$\text{Finding 21}$$
$$[1\ \ 3\ \ 9\ \ \underline{17}\ \ 21\ \ 100]$$
$$[\underline{21}\ \ 100]$$
$$\text{Done.}$$

Time

Best case: it’s directly in the center: $\Theta(1)$
Worst case: doesn’t exist: $\Theta(\lg(n))$

$$\text{Searching for} - 1$$
$$[1\ \ 2\ \ 5\ \ \underline{7}\ \ 9\ \ 100]$$
$$[1\ \ \underline{2}\ \ 5]$$
$$[\underline{1}]$$
$$1 \neq -1 \therefore \text{done, doesn't exist}$$

Average case

Assume average means a random element in the list.

Chance of a best case (first pass): $\frac{1}{n}$, 1 step
Chance of finding it on the second pass: $\frac{2}{n}$, 2 steps
Chance of finding it on the third pass: $\frac{4}{n}$, 3 steps
Chance of finding it on the ith pass: $\frac{2^{i-1}}{n}$, i steps

Find the expected value of the number of steps:

$$T(n) = \sum_{i=1}^{\lg n} \left(\frac{2^{i-1}}{n}\right)(\mathbb{c}_1(i)) = \mathbb{c}_1\left(\frac{1}{n} - 1 + \lg(n)\right) = \Theta(\lg n)$$
$$\underbrace{T(n) = T\left(\frac{n}{2}\right) + \mathbb{c}_1 \land T(1) = \mathbb{c}_1}_{\text{recurrence}}$$
$$\Rightarrow T(n) = \mathbb{c}_1 + \mathbb{c}_1 \lg n$$

Recurrence

Maximum Contiguous Sum, Divide and Conquer

$$T(1) = 1$$
$$T(n) = 2T\left(\frac{n}{2}\right) + n$$
$$\Rightarrow T(n) = n(1 + \lg n) = \Theta(n \lg n)$$

Recurrance Relations

Here is a normal function:

$$f(x) = x + 1$$

But you can also define functions recursively:

$$f(x) = \frac{1}{2}f(x-1) + 1 \leftarrow \text{recurse}$$
$$f(1) = 1 \leftarrow \text{equivalent to "base case"}$$

Example: Binary Search

$$T(n) = 1 + T\left(\frac{n}{2}\right)$$

Example: Maximum Contiguous Sum, Divide and Conquer

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Defintion of Recurrance

A recurrance defines a function $T(n)$ in terms of itself and a set value for a small n.

In programming, we only care about integer n, so we may need to add some floors or other methods of ensuring integers remain integers.

Computing Recurrances

How to compute:

$$T(n) = 3T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + 5n, T(0) = T(1) = 2$$
$$\text{Find } T(5)$$
$$T(5) = 3T(2) + 5 \cdot 5 = 3(3T(1) + 5 \cdot 2) + 25 = 3(3 \cdot 3 + 10) + 25 = 3 \cdot 19 + 25 = 82$$

Our goal is to find a closed form version of this formula. If not, at least find the Θ complexity of it.

Defintion of Closed Form

Non-self-referential answer to a question (typically formed of a finite set of basic functions). No dependencies.

Methods of finding the exact time

Drilling Down

1. Start with the recurrence
2. Plug in for increasing depths of recursions ($T\left(\frac{n}{2}\right), T\left(\frac{n}{4}\right)$, ...)
3. Notice a pattern and then create a general formula for an arbitary level of recursion (k).
4. Find the formula to create the base case (2^k), then make it a log ($\log_2(n) = k$)
5. Plug in and get a solution.

Examples

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{1}{2}n \text{ with } T(1) = 3$$

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{1}{2}n$$

$$= 2\left(2T\left(\frac{n}{4}\right) + \frac{1}{4}n\right) + \frac{1}{2}n$$

$$= 2^2T\left(\frac{n}{2^2}\right) + 2 \cdot \frac{1}{2}n$$

$$= 2^2\left(2T\left(\frac{n}{2^3}\right) + \frac{1}{2}\frac{1}{2^2}n\right) + 2 \cdot \frac{1}{2}n$$

$$= 2^3T\left(\frac{n}{2^3}\right) + 3 \cdot \frac{1}{2}n$$

$$\text{there appears to be a pattern!}$$

$$= 2^kT\left(\frac{n}{2^k}\right) + \frac{nk}{2}$$

$$\text{This ends when } n = 2^k, \text{because our base case is } T(1)$$

$$\text{Assume that } n \text{ is an integer power of 2, because we're CS students and like integers.}$$

$$n = 2^k \Rightarrow \log_2(n) = \log_2\left(2^k\right) \Rightarrow \log_2(n) = k \Rightarrow \text{plug in the} \log$$

$$= 2^{\log_2(n)}T\left(\frac{n}{2^{\log_2(n)}}\right) + \frac{n\log_2(n)}{2}$$

$$= nT\left(\frac{n}{n}\right) + \frac{n\log_2(n)}{2}$$

$$= nT(1) + \frac{n\log_2(n)}{2}$$

$$= \boxed{3n + \frac{n\log_2(n)}{2}}$$

Practice

$$T(n) = 2T\left(\frac{n}{2}\right) + n, T(1) = 4$$

$$T(n) = 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$$

$$= 2^2 T\left(\frac{n}{2^2}\right) + 2n$$

$$= 2^2\left(2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right) + 2n = 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

$$= 2^k T\left(\frac{n}{2^k}\right) + kn$$

$$\text{Assume that } n \text{ is an integer power of 2}$$

$$n = 2^k \Rightarrow \log_2(n) = \log_2\left(2^k\right) \Rightarrow \log_2(n) = k$$

$$= 2^{\log_2(n)} T\left(\frac{n}{2^{\log_2(n)}}\right) + (\log_2(n))n$$

$$= nT\left(\frac{n}{n}\right) + n\log_2(n)$$

$$= nT(1) + n\log_2(n)$$

$$= 4n + n\log_2(n)$$

$$= n(4 + \log_2(n))$$

$$= \Theta(n \lg n)$$

Another Example of Recurrence

$$T(n) = 3T\left(\frac{n}{2}\right) + 4, T(1) = 5$$

$$T(n) = 3\left(3T\left(\frac{n}{2^2}\right) + 4\right) + 4$$

$$= 3^2 T\left(\frac{n}{2^2}\right) + 3^1 \cdot 4 + 3^0 4$$

$$= 3^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 3^i \cdot 4$$

$$\vdots$$

$$T(n) = 3^k T\left(\frac{n}{2^k}\right) + 4\left(\frac{3^k - 1}{2}\right)$$

$$n = 2^k \Rightarrow \log_2(n) = k$$

$$3^{\log_2(n)} T\left(\frac{n}{n}\right) + 4\left(\frac{3^{\log_2(n)} - 1}{2}\right)$$

$$3^{\frac{\log_3(n)}{\log_3(2)}} T\left(\frac{n}{n}\right) + 4\left(\frac{3^{\frac{\log_3(n)}{\log_3(2)}} - 1}{2}\right)$$

$$n^{\frac{1}{\log_3(2)}} T(1) + 4\left(n^{\frac{1}{\log_3(2)}} - 1\right)$$

$$n^{\frac{1}{\log_3(2)}} 5 + 4\left(n^{\frac{1}{\log_3(2)}} - 1\right)$$

$$n^{\log_2(3)} 5 + 4\left(n^{\log_2(3)} - 1\right)$$

Trees

Number of branches

$T(n) = 2T\left(\frac{n}{2}\right) + n$ implies the fact that the algorithm uses 2 branches because it doubles the effort of the lower part $\left(T\left(\frac{n}{2}\right)\right)$; therefore, it runs it twice.

Then, $T(n) = 1T\left(\frac{n}{2}\right) + n \Rightarrow$ one branches, $T(n) = 3T\left(\frac{n}{2}\right) + n \Rightarrow$ three branches

Depth

$T(n) = T\left(\frac{n}{2}\right) + n \Rightarrow$ depth $\log_2(n)$ because you divide by two.

$T(n) = T(n - 1) + n \Rightarrow$ depth n because you subtract by one ($1 \times n = n$).

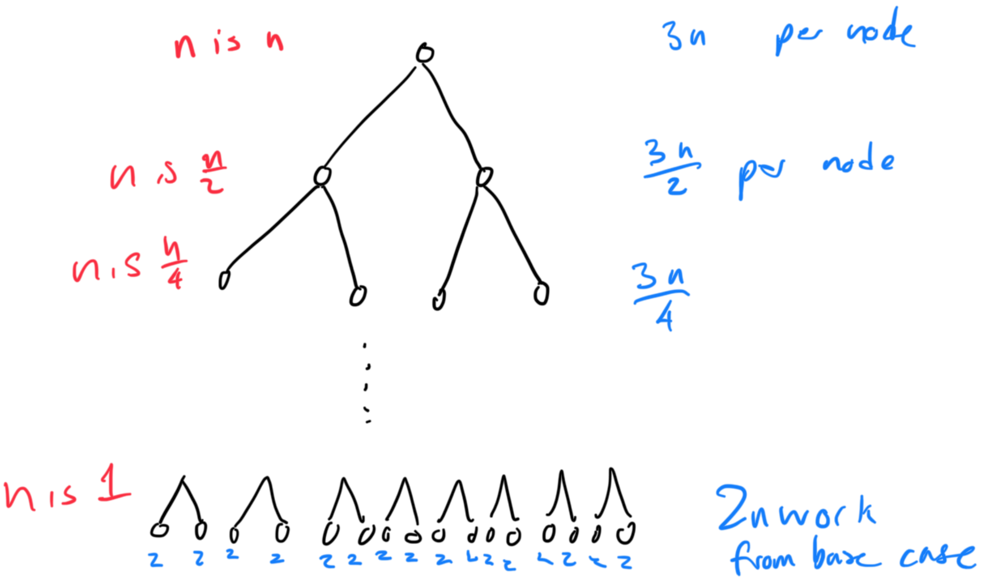
Factorial

```
def fac(n):
  if n == 0: return 1
  return n * fac(n - 1)
```

$$\therefore T(n) = n \times T(n - 1) + \mathbb{c}_2, T(0) = \mathbb{c}_1$$

Tree

$$\underbrace{2}_{\text{binary tree}} \underbrace{T\left(\frac{n}{2}\right)}_{\Rightarrow \text{depth} \log_2(n)} + \underbrace{3n}_{\text{work per node}}, T(1) = 2$$



k	Total
0	3n
1	3n
2	3n
3	3n
⋮	
$\log_2(n)$	2n

$$\therefore T(n) = \sum_{i=1}^{\lg n - 1} (3n) + 2n$$

Justin's way

$$T(n) = 2T\left(\frac{n}{3}\right) + 5n + 1$$

Level	# of nodes	Value of node at each list	Total value of level
0			

… procceeding along because Max skipped it.

Another Drilling Down Example

$$T(n) = 2T(n-1) + 1, T(0) = 3$$

$$
\begin{aligned}
T(n) &= 2(2T(n-2)+1)+1 \\
&= 2^2T(n-2)+2+1 \\
&= 2^2(2T(n-3)+1)+2+1 \\
&= 2^3T(n-3)+2^2+2+1 \\
&= 2^kT(n-k)+\sum_{i=0}^{k-1}2^i \\
&= 2^kT(n-k)+\frac{1-2^k}{1-2} \\
&= 2^kT(n-k)+2^k-1 \\
&\text{Let } k=n \\
&= 2^nT(n-n)+2^n-1 \\
&= 2^nT(0)+2^n-1 \\
&= 2^n\cdot 3+2^n-1 \\
&= 4\cdot 2^n-1 \\
&= 2^{n+2}-1 \\
&= \Theta(2^n)
\end{aligned}
$$

Recurrance from table

Create this table:

Level	Count	Time per Node	Time for Level
$i=0$	1	$2n+1$	$1(2n+1)$
$i=1$	3	$2\left(\frac{n}{5}\right)+1$	$3\left(2\left(\frac{n}{5}\right)+1\right)$
$i=2$	9	$2\left(\frac{n}{25}\right)+1$	$9\left(2\left(\frac{n}{5^2}\right)+1\right)$
$i=k$	3^k	4	$3^k\cdot 4$

Then, just sum the time for each level across all levels, and you will have the total time:

$$
\begin{gathered}
T(n)=4\left(3^k\right)+\sum_{i=0}^{k-1}\left(3^i\left(2\left(\frac{n}{5^i}\right)+1\right)\right) \\
T(n)=4\left(3^k\right)+2n\sum_{i=0}^{k-1}\left(\frac{3}{5}\right)^i+\sum_{i=0}^{k-1}\left(3^i\right) \\
\vdots \\
T(n)=5n-\frac{1}{2}\left(3^{\log_5(n)}+1\right)=\Theta\left(3^{\log_5(n)}\right)
\end{gathered}
$$

In the end your sum will be in final form. Figure out k in terms of n for your base case, and then plug in.

Master Theorem

Take this as an example:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

When going through a tree, $\log_b(n)$ will tell you how deep your tree will be, and $f(n)$ is the amount of work per level.

1. If $f(n) = O(n^c)$ and $\log_b(a) > c$, then $T(n) = \Theta\left(n^{\log_b(a)}\right)$
   a. If $f(n) = \Theta(n^c)$ and $\log_b(a) = c$, then $T(n) = \Theta\left(n^{\log_b(a)} \lg n\right)$
2. If $f(n) = \Theta\left(n^c \lg^k(n)\right)$, and $\log_b(a) = c$, then $T(n) = \Theta\left(n^{\log_b(a)} \lg^{k+1} n\right)$
3. If $f(n) = \Omega(n^c)$ and $\log_b(a) < c$ then $T(n) = \Theta(f(n))$

Steps:
1. Write out $\Theta(f(n))$
2. Write out $\log_b(a)$
3. Patten match

Example

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$c = 1, \log_2(2) = 1 \Rightarrow T(n) = \Theta\left(n^{\lg(2)} \lg(n)\right)$$

Example

$$T(n) = 8T\left(\frac{n}{2}\right) + n^2 + 2n + 1$$

$$f(n) = \Theta(n^2) \Rightarrow c = 2$$

$$\log_b(a) = \log_2(8) = 3$$

$$3 > 2 \Rightarrow T(n) = \Theta(n^3) \qquad \text{(Case 1)}$$

Example

$$T(n) = 9T\left(\frac{n}{3}\right) + n^2 + \lg(n)$$

$$f(n) = \Theta(n^2) \Rightarrow c = 2$$

$$\log_b(a) = \log_3(9) = 2$$

$$2 = 2 \Rightarrow T(n) = \Theta(n^2 \lg n)$$

Example

$$T(n) = 9T\left(\frac{n}{3}\right) + n^2 \lg n + \lg n$$

$$f(n) = \Theta(n^2 \lg n) \Rightarrow c = 2, k = 1$$

$$\log_2 9 = 2$$

$$2 = 2$$

$$\Rightarrow T(n) = \Theta(n^2 \lg^2 n)$$

Example

$$T(n) = 5T\left(\frac{n}{25}\right) + n + 1$$

$$f(n) = \Theta(n) \Rightarrow c = 1$$

$$\log_5(25) = 2$$

$$1 < 2 \Rightarrow T(n) = \Theta(f(n)) = \Theta(n)$$

Example

$$T(n) = 3T\left(\frac{n}{2}\right) + \lg n$$

$$f(n) = \Theta(\lg n) = O(n) \Rightarrow c = 1$$

$$\log_2(3) \Rightarrow 1 < \log_2(3) < 2$$

$$\log_2(3) > 1 \Rightarrow T(n) = \Theta\left(n^{\log_2(3)}\right)$$

$$T(n) = 16T\left(\frac{n}{2}\right) + n^3 \lg(n)$$

$$f(x) = O(n^4) \Rightarrow \log_2(16) = 4 \not> 4$$

$$\text{But!}$$

$$f(x) = O(n^{3.5}) \Rightarrow \log_2(16) = 4 > 3.5 \Rightarrow T(n) = \Theta\left(n^{\log_2(16)}\right) = \Theta(n^4)$$

Merge Sort

Simple Function: Merge!

How would you combine two lists, both in sorted order, to one list?

$$a = [5 \;\; 9 \;\; 11 \;\; 15 \;\; 23 \;\; 27 \;\; 29]$$

$$b = [8 \;\; 12 \;\; 17 \;\; 19]$$

$$\Rightarrow [5 \;\; 8 \;\; 9 \;\; 11 \;\; 12 \;\; 15 \;\; 17 \;\; 19 \;\; 23 \;\; 27 \;\; 29]$$

Compare the first two elements, remove the smallest one, and add it to your new list.

(this is broken in some fashion, and I don't feel like figuring out why)

```python
def merge(a: list, b: list) -> list:
  a_i = 0
  b_i = 0
  output = []

  while a_i < len(a) and b_i < len(b):
    if a[a_i] > b[a_i]:
      output.append(a[a_i])
      a_i += 1
    else:
      output.append(b[a_i])
      b_i += 1

  if a_i >= len(a):
    output.extend(b[b_i + 1:])
  else:
    output.extend(a[a_i + 1:])

  return output
```

Merge sort!

```
def merge_sort(a: list) -> list:
  if len(a) == 1:
    return a

  return merge(merge_sort(a[: len(a) // 2]), merge_sort(a[len(a) // 2:]))
```

$$[9\ \ 17\ \ 8\ \ 3\ \ 2\ \ 1\ \ 6\ \ 10]$$
$$[9\ \ 17\ \ 8\ \ 3] \qquad [2\ \ 1\ \ 6\ \ 10]$$
$$[9\ \ 17]\ [8\ \ 3]\ [2\ \ 1]\ [6\ \ 10]$$
$$[9]\ [17]\ [8]\ [3]\ [2]\ [1]\ [6]\ [10]$$
$$[9\ \ 17]\ [3\ \ 8]\ [1\ \ 2]\ [6\ \ 10]$$
$$[3\ \ 8\ \ 9\ \ 17]\ [1\ \ 2\ \ 6\ \ 10]$$
$$[1\ \ 2\ \ 3\ \ 6\ \ 8\ \ 9\ \ 10\ \ 17]$$

Properties
Max space: $S(n) = S\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow \Theta(n)$

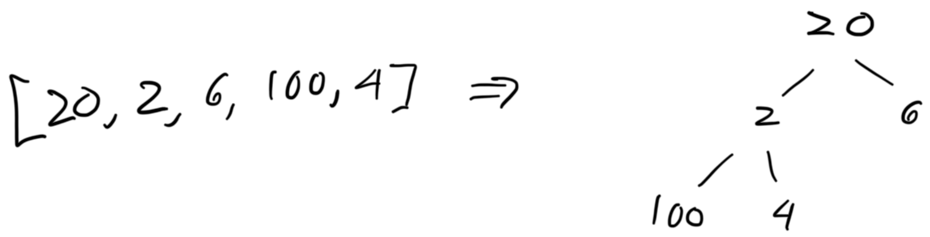
Stable: yes

In place: no

Binary Trees

On storing binary trees

Instead of using an actual tree data structure, we simply use an array to store them:



If our index is i, our children have indices $2i$ and $2i + 1$. If our index is i, our parent is at index $\lfloor \frac{i}{2} \rfloor$. (This is one indexed)

This has the advantage that most things are faster, but on the other hand, if you want to insert an element or rotate a tree, or other things that may only modify a subset of the whole structure, it may be slower.

Level notes
Level 0 is the root node.

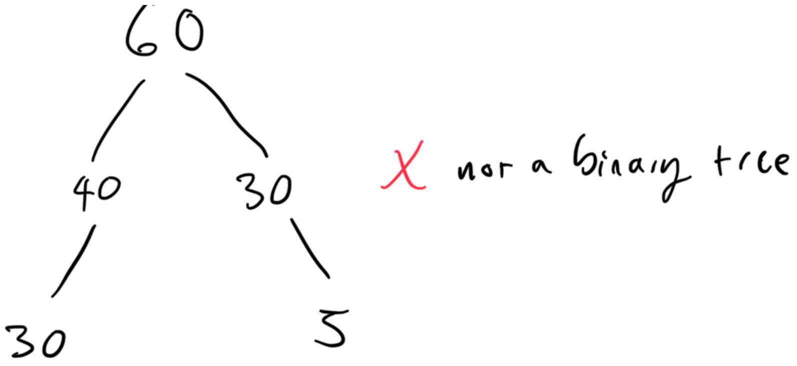
The leftmost node has the index 2^k.

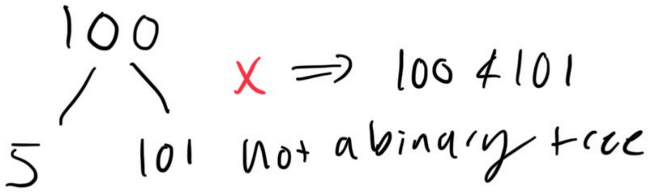
A node with index i is in level $\lfloor \lg i \rfloor$

If you have n nodes, the max level is $\lg n$.

Max Heap

Definition
A max heap is a complete binary tree in which each node is larger than its children. (Forming a partial ordering)





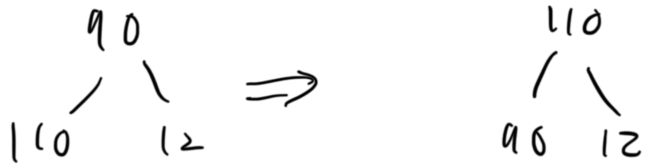
Why is this useful?
You can get the maximum very easily; it's always at the top! This can be used to create priority queues, something that's often used in OS schedulers.

Funny Properties
Every subtree of a max heap is a max heap (Seems useful! May lead to recursion).

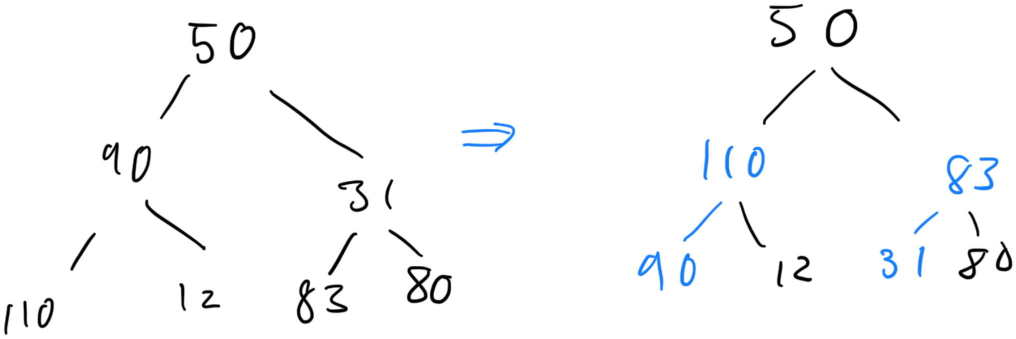
`maxheapify`
Given a list, how do we convert it to a max heap? (assuming you do convert the list to a binary tree via the above process).

After that conversion, how can we now impose the order enforced by the max heap on the binary tree?

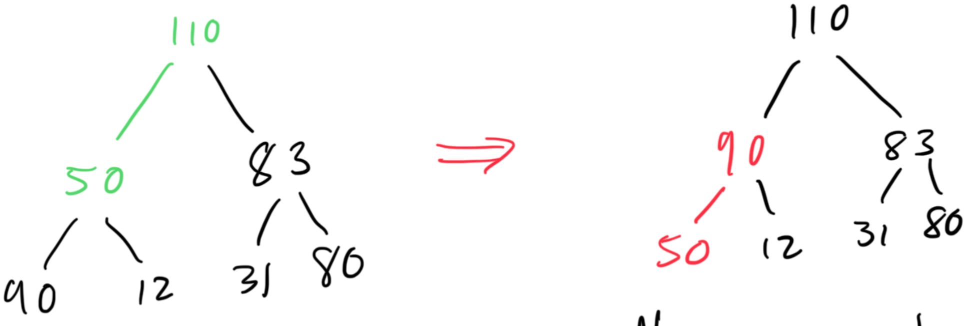
An observation: If we start at the bottom and work up, we can easily make each subtree into a max heap.



make the biggest child go up



Now let 50 fall:



Now a max heap!

110, by itself, is already a max heap

12 is a max heap

83 is a max heap

80 is a max heap

∴ every node alone is a max heap

Basically, have a function `float_key_down` that assumes all children are maxheaps and progressively swaps down an element.

The best case of this is $O(n)$ when all elements are already in the correct order. The worst case of this is $O(n \lg n)$ if you have to travel the maximum distance for each float down.

To use this for sorting, build the max-heap and then pull off the top element repeatedly. This is heap sort! 🎉

Internships

Why are there no internships now? Well, previously, tech companies had a lot of capital due to the low cost of getting more money via loans. But since COVID ended, to reduce inflation, it was made much more expensive to get more money, so tech companies had to become profitable, and to do so, they fired lots of people. And so now, nobody would want to hire an intern because they have a much better alternative than just hiring someone who had just been at Google. So basically, just try to do something, and don't just waste your time over the summer, even if doing something is just doing projects, teaching a bootcamp, or being a TA, just do something related to CS. ("real" jobs are pointless.)

To get the maximum of a max heap off, you remove the max by swapping it to the end and then float that key down.

To use this to sort

1. Build the heap $O(n \lg n)$
2. Remove max n times $O(n \lg n)$

therefore, it is a $O(n \lg n)$ sort.

It is

1. In place
2. $\Theta(1)$ auxillary space
3. But not stable.

Exam

Certainly there

There will be a recurrence using digging down, where you need to show steps and the final sum. (recommend no more than 5 minutes spent on this question). To practice, use Justin's exams, online examples, etc.

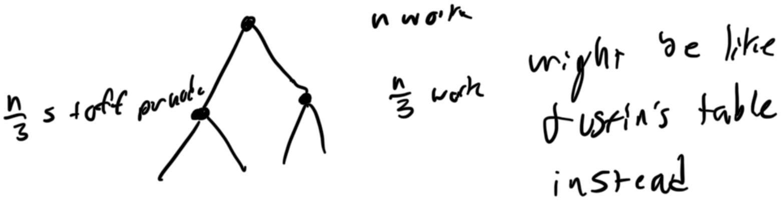
Master Theorem: memorize it 😕.

There will be about 5 problems, including ones where you can't use it. One will be somewhat of a trick question. It's recommended that you take 1 minute per problem.

Maybe there

Drawing out a tree for a recurrence, labeling work done at each level. (less than five minutes)

$$T(n) = 2T\left(\frac{n}{3}\right) + n$$



Sort something. Draw out merge sort, heap sort, quick sort, insertion sort, bubble sort, or selection sort. Practice exists on Justin’s exams

The other half

Conception stuff: how does a sort change if you guarantee a certain input property? What sort would you choose to use with a specific input property?

If I made a small change, how might the sort behave?

Look at a new algorithm and write a recurrence.

There will be no pseudocode creation.

Terms you should know

- Inversion: an element is out of order compared to another: $\binom{n}{2}$ opportunities
- Stable: elements that compare equal will remain in the same order after sorting
- In place: $O(1)$ auxiliary space.

Practice: Max will publish samples. Justin’s notes have questions like this at the end. Understand why a sort is the way it is.

Quick Sort

Another fast sort! Also divide and conquer.

Idea

Pick an element on the far right called the pivot, and move it to the correct place. (Move everything less than the pivot to the left of it. Move everything greater than it to the right.)

Recurse on the left and right halves.

7 10 5 3 2 6

pivot

$\Rightarrow \underbrace{5\ \ 3\ \ 2}\ \ 6\ \ \underbrace{7\ \ 10}$

new lists to sort

$T(n) = T(x) + T(y) + n \Rightarrow x + y + 1 = n$

For example with

[9 10 5 6 3 1]

1 is a terrible pivot so

[1 9 10 5 6 3 1]

$\Rightarrow T(7) = T(0) + T(6) + 7$

Parition

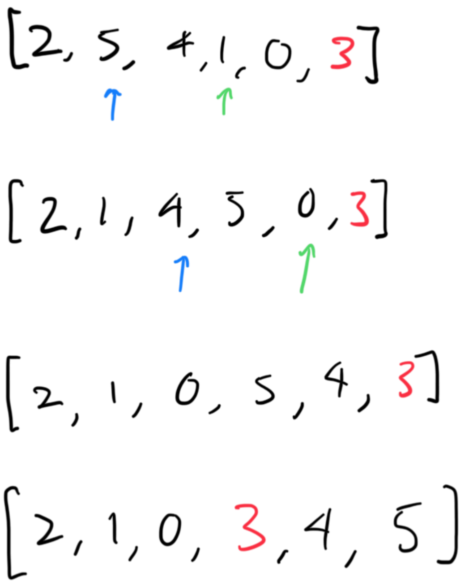
It puts the pivot in the right place, with elements less than it before it and those greater than it after it.

Procedure for pivot

The pivot is typically the last element of the list.

Procced as follows:

1. Find the leftmost key greater than the pivot, and the next key less than or equal to the pivot.
2. Swap them
3. Repeat until there is nothing left.

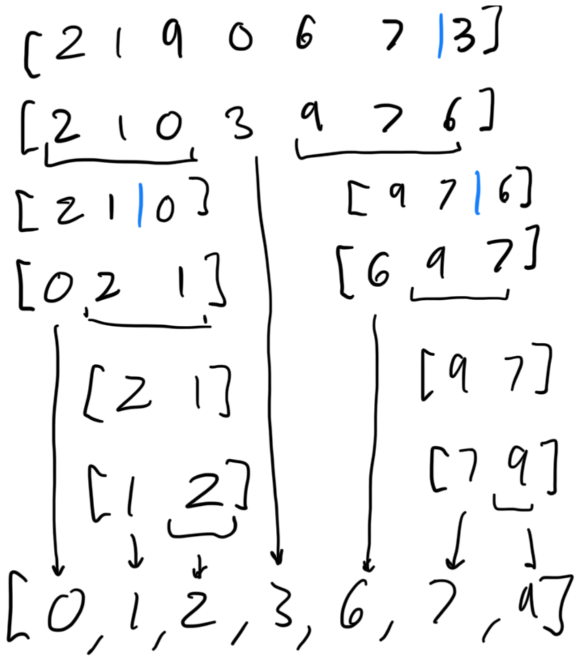


The blue starts when it finds something greater than the pivot. The green one then goes forward from there. Neither of them will go backwards, so this is a maximum of $2n-1$ runtime, which is $O(n)$. You can also be smart and do other stuff to just have n runtime.

Partition Pseudocode

```
def partition(A, L, R):
  pivot = A[R-1]
  t = L
  for i in range(L, R):
    if A[i] <= pivot:
      # swap A[t] and A[i]
      (A[t], A[i]) = (A[i], A[t])
      t += 1
  # swap pivot and A[t]
  (A[t], A[R-1]) = (A[R-1], A[t])
  return t
```

But what about quicksort!



```
def quicksort(A, L, R):
  if L < R:
    resulting_pivot_index = partition(A, L, R)
    quicksort(A, L, resulting_pivot_index - 1)
    quicksort(A, resulting_pivot_index + 1, R)
```

Time complexity

Best pivot would be the median. If we were to always get the median, what would be the recurrance for quicksort?

$$T(n) = 2T\left(\frac{n}{2}\right) + n \to T(n) = \Theta(n \lg n)$$

For the worst case, if we were to always get max,

$$T(n) = T(n-1) + n \to T(n) = \Theta(n^2)$$

Average case:

$$T(n) = \sum_{i=0}^{n-1} \frac{1}{n}(T(i) + T(n-i-1) + \Theta(n)) \to T(n) = \Theta(n \lg n)$$

Sort Properties

- Unstable
- In-place
- $O(1)$ memory

Comparision based sorting

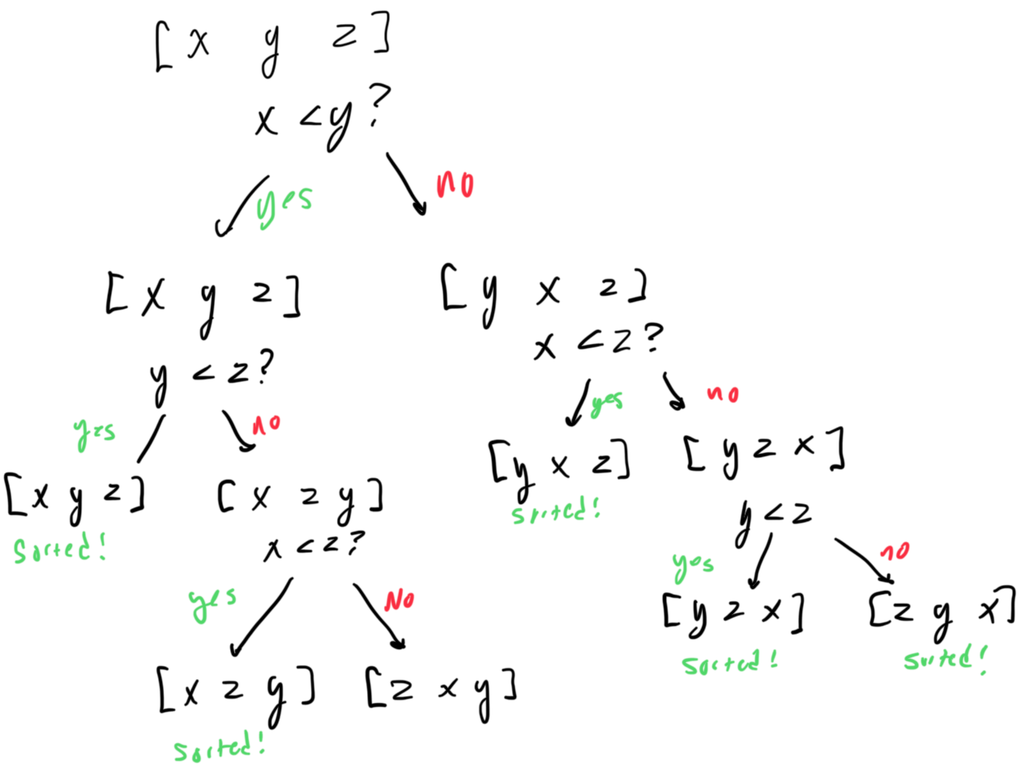
This is what all our sorts have been like up to now: they compare elements to each other and sort accordingly.

Upper Limit on Speed

$O(n \lg n)$

The reason is

Imagine we're sorting a list of size 3:



Because you must make this many comparisons to eliminate all possible initial relative positions, you can't do better than this, which is $O(n \lg n)$.

Review for Exam 2

Reminder on what will be on the exam

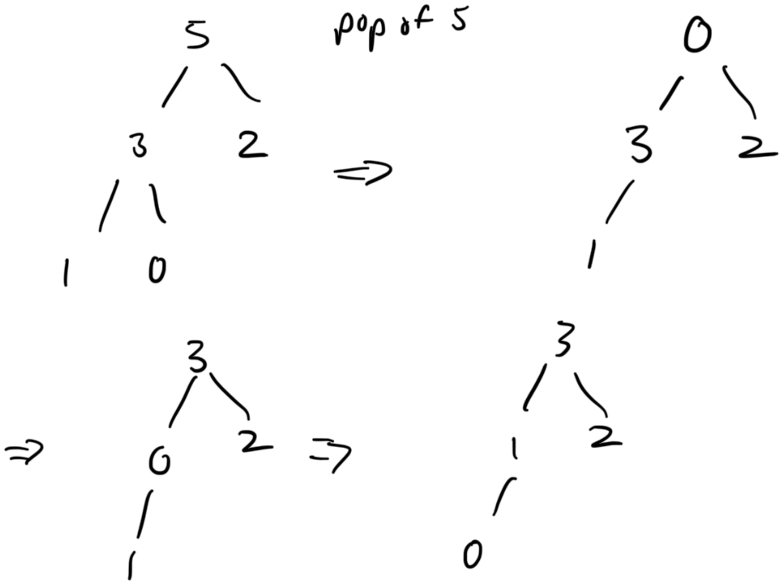
- Definitely on the exam
  - Drilling down
  - Master Theorem (memorize!)
- Definitely on the exam
  - Manually sorting a list using a specific sort
    - Bubble Sort

    - Swap elements one by one, keep repeating (gradually sort the right-hand side of the list)
    - Worst case: $O(n^2)$
    - Best case: $O(n^2)$
    - Average case: $O(n^2)$
    - Auxiliary Memory: $O(1)$
    - Stable
    - In-place
  - Selection Sort
    - Iteratively find the min, then shrink what you're searching
    - Worst case: $O(n^2)$
    - Best case: $O(n^2)$
    - Average case: $O(n^2)$
    - Auxiliary Memory: $O(1)$
    - Unstable
    - In-place
  - Insertion Sort
    - Maintain a sorted sublist and keep finding stuff to put on the right place in the list using binary search
    - Worst case: $O(n^2)$
    - Best case: $O(n)$
    - Average case: $O(n^2)$
    - auxiliary Memory: $O(1)$
    - Stable
    - In-place
  - Merge Sort
    - Split the list in half until you have lists of size 1, then merge them back together.
    - $T(n) = 2T\left(\frac{n}{2}\right) + n$
    - Worst case: $\Theta(n \lg n)$
    - Best case: $\Theta(n \lg n)$
    - Average case: $\Theta(n \lg n)$
    - Auxiliary Memory: $\Theta(n)$
    - Stable
    - Not in-place
  - Heap sort
    - Create a heap, repeatedly take off the top, and then float the key down with the swapped element from the bottom
    - Worst case: $\Theta(n \lg n)$
    - Best case: $\Theta(n \lg n)$
    - Average case: $\Theta(n \lg n)$
    - Auxiliary Memory: $O(1)$
    - Unstable
    - In-place
  - Quicksort
    - Partition the list by the last element and then split into sublists and quicksort the lower parts.

        - Worst case: $\Theta(n^2)$
        - Best case: $\Theta(n \lg n)$
        - Worst case: $\Theta(n \lg n)$
        - Auxiliary Memory: $O(1)$
        - Unstable
        - In-place
    - Building a tree for time complexity calculation
    - Binary search tree (there *will* be a question)
        - Find an element in $O(\lg n)$ time
- Surprise
    - Conceptial Questions

You just need to write summations; you don't need to solve them.

Heap sort single iteration example:



Quick hack: if it's not this exact format: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, it is not the Master Theorem.

Constrainted sorts

Counting sort

Before, with comparison-based sorts, we could not beat $n \lg n$. But now, we are unshackled from those constraints and can use the data better.

For counting sort, we impose the constraint that we are sorting integers between 0 and k.

Example

1. Find k.

2. Declare an array of size k called B.
3. Loop over A. For each element in A, add one to $B[A_i]$.
4. exp

$$\begin{array}{rl}
A = & [9\ 7\ 6\ 3\ 7\ 9\ 6\ 3\ 3\ 1] \\
B = & [0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0] \\
 & \ \ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9 \\
 & \qquad\quad \Longrightarrow \\
B = & [0\ 1\ 0\ 3\ 0\ 0\ 2\ 2\ 0\ 1] \\
 & \ \ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9 \\
 & \qquad\quad \Longrightarrow \\
A = & [1\ 3\ 3\ 3\ 6\ 6\ 7\ 7\ 9\ 9]
\end{array}$$

In place: No

Auxiliary space: $\Theta(k)$

Runtime: $\Theta(n + k)$

Stable: 😔, no, it loses tons of information, too.

Making this sort stable.

1. Find k
2. Declare arrays of size k called B and C.
3. Loop over A. For each element in A, add one to $B[A_i]$.
4. Iterate over B_i and set $C[i] = B[i] + C[i - 1]$ with $C[-1] = 0$
5. Make a new array D of size n.
6. Loop over A in reverse order. Set $D[C[A[i]] - 1] = A[i]$. Decrement C[A[i]].

$$\begin{array}{rl}
A = & [1\ 3\ 9\ 2\ 1\ 3\ 8\ 7\ 1] \\
B = & [0\ 3\ 1\ 2\ 0\ 0\ 0\ 1\ 1\ 1] \\
 & \ \ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9 \\
C = & [0\ 3\ 4\ 6\ 6\ 6\ 6\ 7\ 8\ 9] \\
 & \ \ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9
\end{array}$$

$$\begin{array}{rl}
D = & [\qquad\qquad\qquad\qquad\quad] \\
 & \ \ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8
\end{array}$$

$$A[8] = 1 \Rightarrow C[1] = 3 \Rightarrow \text{insert to } 3 - 1 = 2, C[1] = 2$$

$$\begin{array}{rl}
D = & [\qquad\ 1 \qquad\qquad\qquad] \\
 & \ \ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8 \\
C = & [0\ 2\ 4\ 6\ 6\ 6\ 6\ 7\ 8\ 9] \\
 & \ \ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9
\end{array}$$

$$A[7] = 7 \Rightarrow C[7] = 7 \Rightarrow 7 - 1 = 6$$

$$\begin{array}{rccccccccccl} D = [& & & 1 & & & & 7 & & &] \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \end{array}$$

$$\begin{array}{rcccccccccccl} C = [& 0 & 2 & 4 & 6 & 6 & 6 & 6 & 7 & 8 & 9 &] \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \end{array}$$

$$A[6] = 8 \Rightarrow C[8] = 8 \Rightarrow 8 - 1 = 7$$

$$\begin{array}{rccccccccccl} D = [& & & 1 & & & & 7 & 8 & &] \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \end{array}$$

$$\begin{array}{rcccccccccccl} C = [& 0 & 2 & 4 & 6 & 6 & 6 & 6 & 6 & 7 & 9 &] \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \end{array}$$

$$A[5] = 3 \Rightarrow C[3] = 6 \Rightarrow 6 - 1 = 5$$

$$\begin{array}{rccccccccccl} D = [& & & 1 & & & 3 & 7 & 8 & &] \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \end{array}$$

$$\begin{array}{rcccccccccccl} C = [& 0 & 2 & 4 & 5 & 6 & 6 & 6 & 6 & 7 & 9 &] \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \end{array}$$

$$A[4] = 1 \Rightarrow C[1] = 2 \Rightarrow 2 - 1 = 1$$

$$\begin{array}{rccccccccccl} D = [& & 1 & 1 & & & 3 & 7 & 8 & &] \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \end{array}$$

$$\begin{array}{rcccccccccccl} C = [& 0 & 1 & 4 & 5 & 6 & 6 & 6 & 6 & 7 & 9 &] \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \end{array}$$

$$A[3] = 2 \Rightarrow C[2] = 4 \Rightarrow 4 - 1 = 3$$

$$\begin{array}{rccccccccccl} D = [& & 1 & 1 & 2 & & 3 & 7 & 8 & &] \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \end{array}$$

$$\begin{array}{rcccccccccccl} C = [& 0 & 1 & 3 & 5 & 6 & 6 & 6 & 6 & 7 & 9 &] \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \end{array}$$

$$A[2] = 9 \Rightarrow C[9] = 9 \Rightarrow 9 - 1 = 8$$

$$\begin{array}{rccccccccccl} D = [& & 1 & 1 & 2 & & 3 & 7 & 8 & 9 &] \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \end{array}$$

$$\begin{array}{rcccccccccccl} C = [& 0 & 1 & 3 & 5 & 6 & 6 & 6 & 6 & 7 & 8 &] \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \end{array}$$

$$A[1] = 3 \Rightarrow C[3] = 5 \Rightarrow 5 - 1 = 4$$

$$\begin{array}{rccccccccccl} D = [& & 1 & 1 & 2 & 3 & 3 & 7 & 8 & 9 &] \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \end{array}$$

$$\begin{array}{rcccccccccccl} C = [& 0 & 1 & 3 & 4 & 6 & 6 & 6 & 6 & 7 & 8 &] \\ & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \end{array}$$

$$A[0] = 1 \Rightarrow C[1] = 1 \Rightarrow 1 - 1 = 0$$

$$D = [1\ \ 1\ \ 1\ \ 2\ \ 3\ \ 3\ \ 7\ \ 8\ \ 9]$$

$$0\ \ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8$$

$$C = [0\ \ 0\ \ 3\ \ 4\ \ 6\ \ 6\ \ 6\ \ 6\ \ 7\ \ 8]$$

$$0\ \ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8\ \ 9$$

$$D \text{ is the sorted array, but stablely sorted this time}$$

```python
def counting_sort(A):
  assert min(A) >= 0, "the minimum in the list is less than zero"
  k = max(A)
  n = len(A)
  pos = [0] * (k+1)
  anew = [0] * n
  for x in A:
    pos[x] += 1

  for i in range(1, k + 1):
    pos[i] += pos[i - 1]

  for x in reversed(A):
    anew[pos[x] - 1] = x
    pos[x] -= 1

  return anew
```

Bucket Sort

1. Create n buckets
2. Assume the input is uniformly distributed. Then, use uniform bucket sizes and put elements into the bucket they correspond to (index into the bucket is $\left\lfloor \frac{A[i]}{k} \right\rfloor$)
3. Sort the subarrays of the buckets. This can be any sort.
4. Insert the elements back in sorted order by looping over the buckets and then the subarrays.

A=[56 79 83 15 16 17 21 33]

Ten Buckets:

[

0	1	2	3	4	5	6	7	8	9
	15	21	33		56		79	83	
	16								
	17								

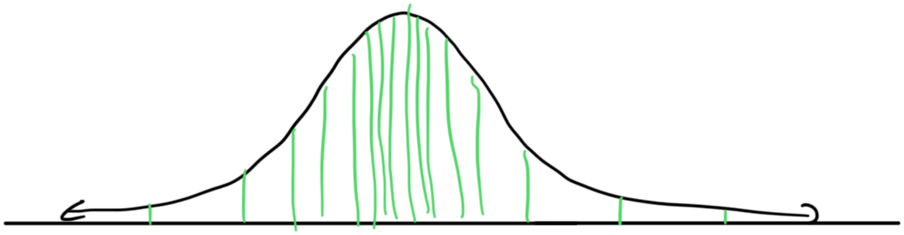
]

Sort Subarrays, then form new, sorted array:

⇒ [15 16 17 21 33 56 79 83]

Alternative indexing into the bucket also exists: change the bucket sizes to make the buckets have an expected value of 1 element.

For example, with a normal distribution, the buckets will be bigger towards the centers:



Properties
Auxiliary space: $\Theta(n)$
Worst case: $\Theta(n^2)$
Average case: $\Theta(n)$
Best case: $\Theta(n)$

In-place: no
Stable: yes

Radix sort

Designed to handle things that can be written as digits with a particular base.

For example, decimal numbers (523, 716), binary numbers (1001001, 1011010), and strings (“Hi”, “string”)

Stable sort each number based on just the least significant digit.

[123 311 213 141 222 321]

Sort least significant digit

$[\underbrace{311\quad 141\quad 321}_{1}\quad \underbrace{222}_{2}\quad \underbrace{123\quad 213}_{3}]$

Sort by 2nd digit

$[\underbrace{311\quad 213}_{1}\quad \underbrace{321\quad 222\quad 123}_{2}\quad \underbrace{141}_{4}]$

Sort by 3rd digit:

$[\underbrace{123\quad 141}_{1}\quad \underbrace{213\quad 222}_{2}\quad \underbrace{311\quad 321}_{3}]$

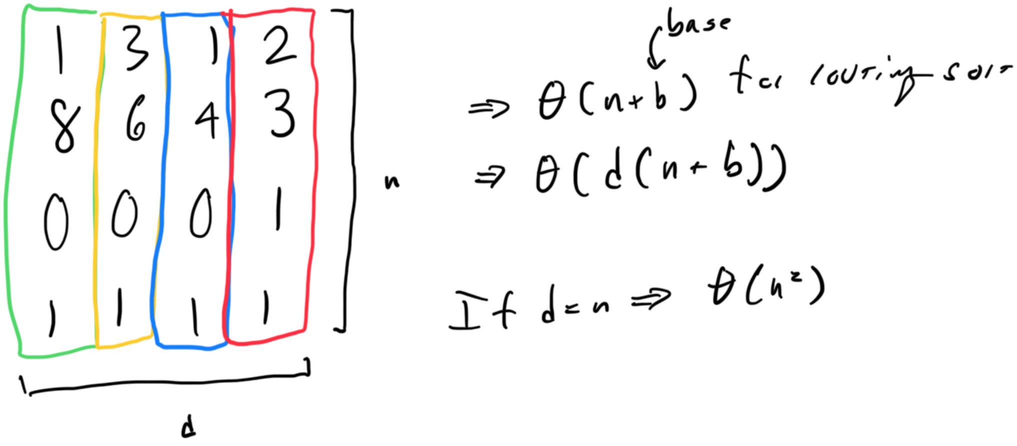
∴ Fin.

Notes for radix sort:

1. You need to pad numbers to be all the same length
2. The underlying sort needs to be stable

With an underlying sort of $\Theta(f(n))$, the runtime is $\Theta(df(n))$.

Example:



Usually, you use counting sort because it has the same constraints, and radix sort avoids the downside of using an excessive amount of memory.

Properties

Aux space: If the underlying sort uses $\Theta(g(n))$, aux space is $\Theta(g(n))$

In-place: depends on the underlying sort

Stable: Yes

Faster Multiplications

If we're multiplying two 2-digit numbers, we're actually doing:

$$AB$$
$$A = 10a_1 + a_0$$
$$B = 10b_1 + b_0$$

$$\begin{aligned} AB &= (10a_1 + a_0)(10b_1 + b_0) \\ &= 100a_1b_1 + 10(a_1b_0 + a_0b_1) + a_0b_0 \end{aligned}$$

$$\begin{aligned} &(a_1 + a_0)(b_1 + b_0) = a_1b_1 + a_1b_0 + a_0b_1 + a_0b_0 \\ \Rightarrow\ &a_1b_0 + a_0b_1 = (a_1 + a_0)(b_1 + b_0) - a_0b_0 - a_1b_1 \\ \Rightarrow\ &AB = 100a_1b_1 + 10((a_1 + a_0)(b_1 + b_0) - a_0b_0 - a_1b_1) + a_0b_0 \end{aligned}$$

This results in only 3 multiplications compared to the normal algorithm, which has 4. Multiplications are an order of magnitude slower than addition, so we can treat addition as free. The decimal shift (multiplying by 10) is just a decimal shift, which is also free.

Generalizing, for n digit numbers:

$$AB = 10^n a_1b_1 + 10^{\frac{n}{2}}((a_1 + a_0)(a_1 + a_0)) - a_0b_0 - a_1b_1) + a_0b_0$$

This has 3 $\frac{n}{2}$ digit multiplications.

To multiply a big number, use this recursively. First, apply it on n digits, then use this algorithm for the 3 $\frac{n}{2}$ digit multiplications. Single-digit multiplications are the base case.

$$T(n) = 3T\left(\frac{n}{2}\right) + n \Rightarrow T(n) = \Theta\left(n^{\log_2(3)}\right) = \Theta\left(n^{1.58}\right)$$

```python
import math
def karatsuba(A, B):
  # yes these logs are algorithmically complex, but you really wouldn't use base 10: base 2 is much nicer for computers, or at least one of the powers of two, so you can simply find the position of the highest bit, a much faster operation
  A_digits = math.log10(A) + 1
  B_digits = math.log10(B) + 1
  if A_digits == 1 or B_digits == 1:
    return A * B
  else:
    sp = math.floor(min(A_digits,B_digits) / 2)
    # split digits, sp digits from the right
    # if you were doing base 2, this would just be a bit shift
    A_0 = A % 10**sp
    A_1 = A // 10**sp
    B_0 = B % 10**sp
    B_1 = B // 10**sp
    k_1 = karatsuba(A_1, B_1)
    k_2 = karatsuba(A_1+A_0, B_1 + B_0)
    k_3 = karatsuba(A_0, B_0)
    return 10**(2 * sp) * k_1 + 10**(sp) * (k_2-k_3-k_1) + k_3
```

Graphs

Graphs are vertices/nodes connected by edges.

A loop is an edge joining a vertex to itself.

Multiple edges occur when vertices are connected by more than one edge.

A graph is simple if it has no loops and no multiple edges.

A graph is weighted if the edges have values assigned. These values are usually thought of as costs and are generally positive.

A graph is directed if the edges have directions.

Call V the number of vertices and E the number of edges.

A walk is a traversal of a graph. You can repeat the edges.

A trail is a walk with no repeated edges.

A path is a walk with no repeated edges or vertices.

A cycle is a path where the last edge points to the first vertex.

A graph without cycles is called acyclic.

A graph is connected if there is always a path between two vertices. Conversely, if a path cannot be formed between all pairs of vertices, it is disconnected.

Directed, acyclic, connected graphs are trees.

Disconnected and empty graphs are usually not valid/useful answers, so please don't use them on the homework.

Graph Storage

The Java Solution

This is bad but functional:

```
public class Node {
  private ArrayList<Nodes> nodes;
}
```

Adjacency Matrix:

If you have a symmetric graph, your graph is not directed.

The eigenvectors mean something; if you square the matrix, it tells you the number of connections.

Adjacency list

Define a list A such that $A[i]$ is the list of vertices i connects to.

Longest, Shortest Path

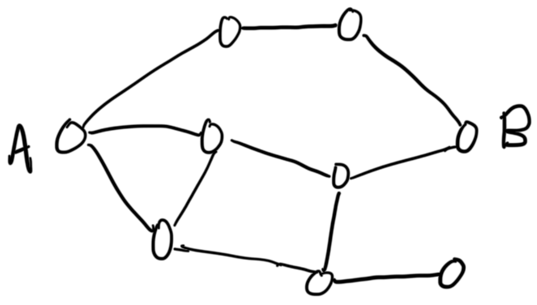
The longest, shortest path through a graph is called the diameter.

Every pair of nodes has a set of paths between them. The shortest path between nodes is the one with the least weight, or for an unweighted graph, the one where you traverse the fewest edges.

Take the set of all the shortest paths between nodes. The one of those that is the longest is the longest, shortest path.

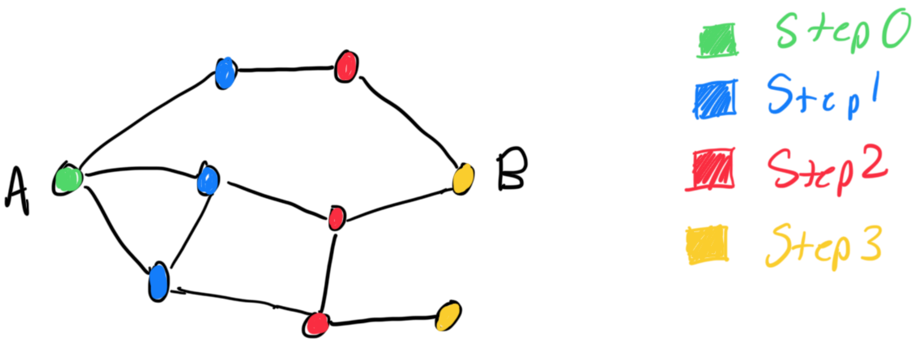
Shortest Path through a graph

Given a simple, connected, undirected, unweighted graph, how do we find the shortest past from note A to node B.



Breadth-First Search (BFS)

Start at A. Move outward to check if their neighbors, if none of those are node B, check that set of node's neighbors.



As an aside, starting at both node A and node N can significantly improve the graph's runtime speed because you have about half the number of steps for both sides, which is very helpful for an algorithm with exponential runtime.

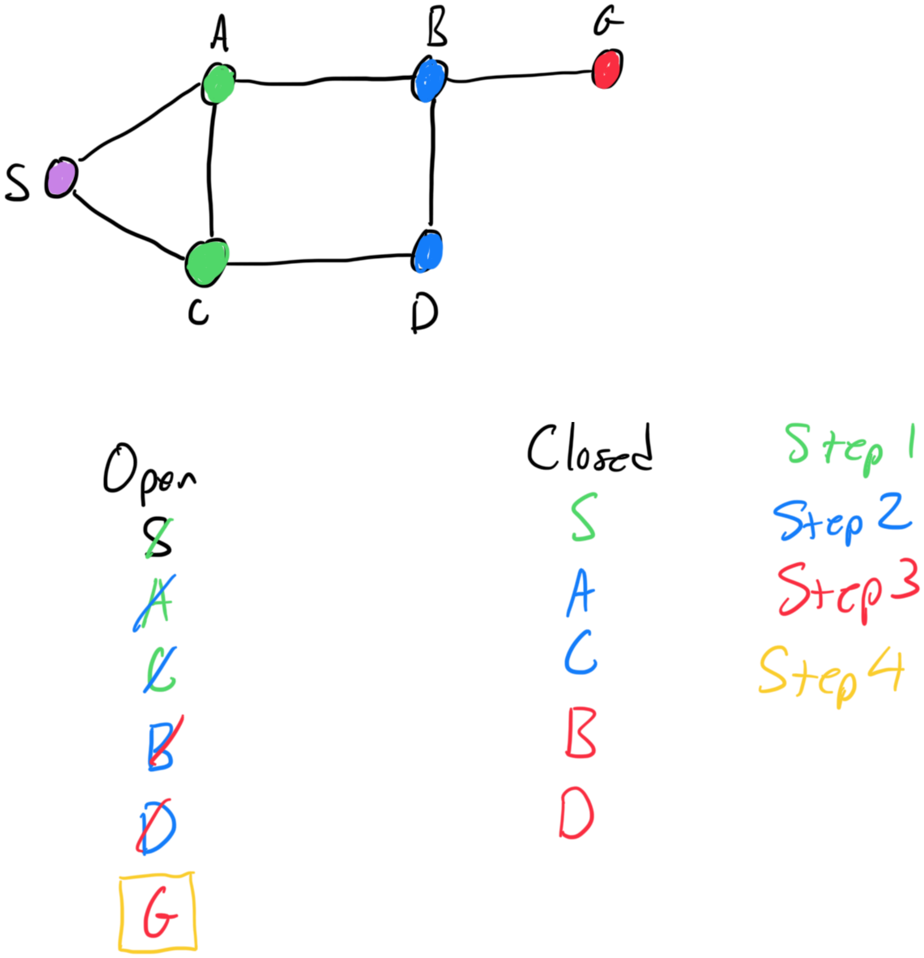
This can also be seen as building a tree showing how far everything is from A.

You have two lists, $O = \text{open}$ and $C = \text{closed}$. O is a queue.

Add S to the O list.

While O has an item: Take the top, and check if it's the goal. If it's not, add it to C and add all of its neighbors that are not in the closed list to O.

```
def bfs(start, end):
  O = [start]
  C = set()
  # while O has an item
  while O:
    # take the top
    item = O.pop()
    # check if it's the goal
    if item == end:
      return # done!
    else:
      # add it to C
      C.add(item)
      # add all neighbors that are not in the closed list to O.
      for neighbor in list(set(item.neighbors) - C):
        O.append(neighbor)
```



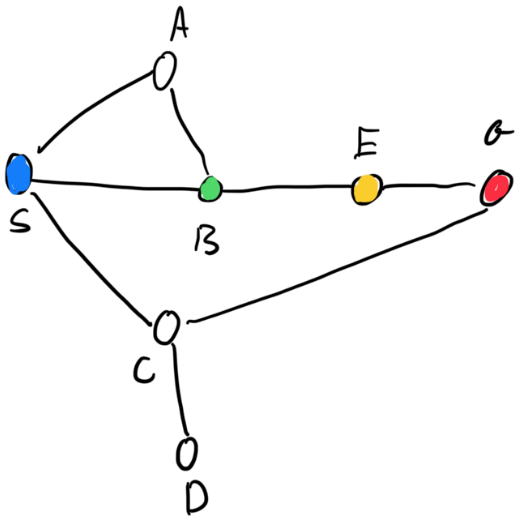
Time complexity
If we store this as a matrix, it will iterate V time over potential path length V, giving $\Theta(V^2)$.

If we store it as an adjacency list, we can find all of a node's neighbors in constant time, giving $\Theta(V + E)$.

This uses $\Theta(V)$ memory.

Depth-First Search (DFS)

Same thing as Breath-First Search, but you use a stack instead of a queue, and don't allow repeats in the Open list.



Open	Closed
~~S~~	S
A	B
C	E
~~B~~	
~~E~~	
$\boxed{G}$	

Step 1

Step2

Step 3

Step4

S - B - E - G is the result.

Dijkstra Algorithm

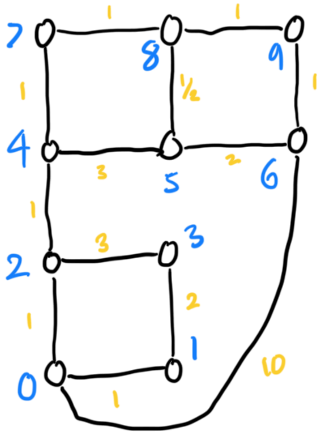
An algorithm for finding shortest paths in weighted graphs.

This is similar to a depth-first search, except you use a priority queue for the weight.

Track the optimal parents through a secondary list P and the optimal distance in a list D.

1. Add the start node to the open priority queue with weight 0.
2. Add the start node to D with a distance of 0 (do not give it parents).
3. Remove the node from the priority queue with the minimum weight.
4. If that node is the end node, you're done. Trace through the P list to find the path.
5. If not, find that node's neighbors.
5. For each neighbor, add the distance to get to it to the weight of the current node.
6. If the neighbor has not been found before, add it to the open queue with the weight of the distance.
7. Also, if that distance is lower than the distance formerly found, as stored in the D list, update the D list with the new distance and update the P list to refer to the current node. If the node is in the open list, update its priority, too.

Example



Priority	Open
0	~~0~~
1	~~1~~
1	~~2~~
~~10~~ 6.5	6
3	~~3~~
2	~~4~~
3	~~7~~
~~5~~ 4.5	~~5~~
4	~~8~~
5	9

Closed
0
1
2
4
3
7
8
9

	0	1	2	3	4	5	6	7	8	9
Dist = [		1	1	3	2	4.5	6.5	3	4	]
Parent = [		0	0	1	2	8	5	4	7	]

Depth-first traverse

(Yes, we have gone over this before, but here are more details)

Data structure details

We want to be able to check constantly to see if something is on the open list.

One solution is to have a separate hash set specifically to check if it is in the open list.

Or, have an array with a pointer for each node in a doubly linked list. Every time you want to know if a node is in the open list, check the pointer array. If there is a pointer there, remove whatever it points to.

These both have the same algorithmic complexity, but the second is faster in practice.

Pseduocode

```
# Let V be the number of vertices, a be the start node
from llist import dllist

vorder = []

stack = dllist()
x = stack.append(a)

visited = [False] * V
SP = [None] * V
SP[a.index] = x

while stack: # is not empty
  x = stack.pop()
  if not visited[x.index]:
    visited[x.index] = True
    vorder.append(x)
  for y in x.neighbors:
    if not visited[y.index]:
      if sp[y.index] != None:
        stack.remove(sp[y.index])
      stack.append(y)
      sp[y.index] = y
```

The problem with depth-first search is that it doesn't care about direction at all; it will just go until it either hits a dead-end or finds something. Dijkstra's algroithm gives it a direction.

- Comparison based sorting
- Counting/bucket/radix sort
- Karatsuba algorithm
- Graphs, BFS, DFS, Dijkstra's

You will need to perform:

- Counting sort
  - Create the pos array and walk through the steps.
  - pos should always be in ascending order
  - Practice Justin's Exams, make a list and do it

- Bucket sort
  - Make buckets, put stuff in them, sort
  - Practice, look online and at problems he will release.

- Radix sort
  - Check the digits are sorted correctly
  - See prior notes

- BFS and DFS: find a path

- Dijkstra's: find a path

Most likely, it will be focused on counting sort and Dijkstra's.

Exciting stuff

There will be

- Shorter conceptual questions
- Know all the runtimes and know why they are the runtimes.
  - Why are they the runtimes for sorts and pathfinding
- Master theorem might be on the exam

Comparision Based sorting

Sorts where you compare elements. The upper limit is $O(n \lg n)$.

The reason why our other sorts are better is because integer indexing is fast.

Noncomparision sorts

Another sorting things: The income distribution in the US is right-skewed (probably relevant for bucket sort)

Counting sort:

- Understand what pos is, the cumulative array

Example:

$$\begin{gathered} A = [3,4,7,2,1,6,1] \\ \text{counting} = [0,2,1,1,1,0,1,1] \\ \text{pos} = [0,2,3,4,5,5,6,7] \\ \text{A\_new} = [1,1,2,3,4,6,7] \end{gathered}$$

Longer example:

$$\begin{gathered} A = [3,4,7,2,1,6,1] \\ \text{pos} = [0,2,3,4,5,5,6,7] \\ \text{A\_new} = [\ ,\ ,\ ,\ ,\ ,\ ,\ ] \end{gathered}$$

The pos array stores the last index + 1 that a number is associated with. By going backward through the array and inserting it in that last place, you maintain stability.

Stable yes:
Runtime: $\Theta(n + k)$
In-place: no
Aux-space: $\Theta(n + k)$

Bucket Sort

We create n buckets evenly spaced out through our input range.

$[0 - 100], n = 5$

Buckets:

$$[[0 - 20], [20 - 40], [40 - 60], [60 - 80], [80 - 100]]$$

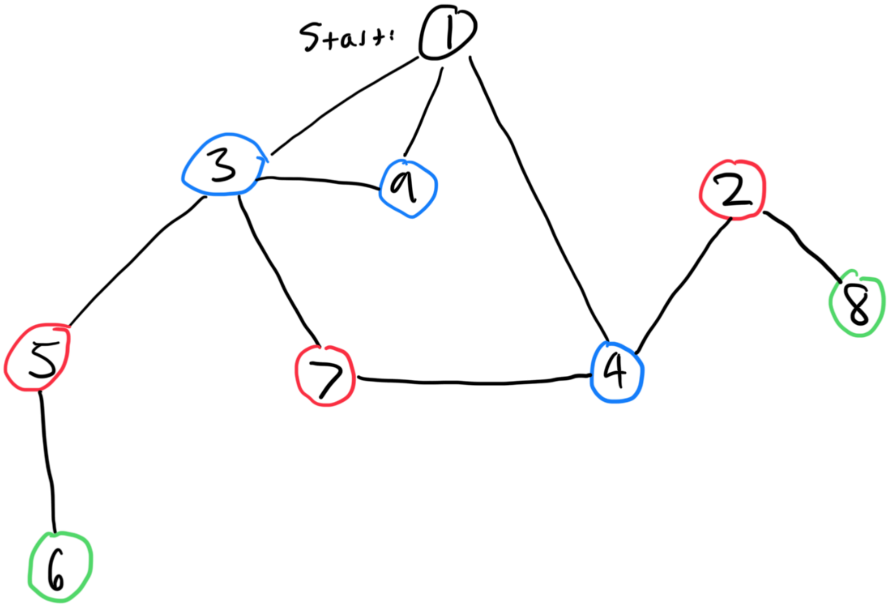
$$A = [16, 9, 7, 30, 98]$$
$$\text{buckets} = [[15, 9, 7], [30], [], [], [98]]$$
$$\text{sort the buckets with some other sort}$$
$$[[7, 9, 15], [30], [], [], [98]]$$
$$A = [7, 9, 15, 30, 98]$$

Runtime (average): $\Theta(n)$
Stable: With stable underlying sorts
In-place: ?
Auxiliary space: $\Theta(n)$

Graph stuff

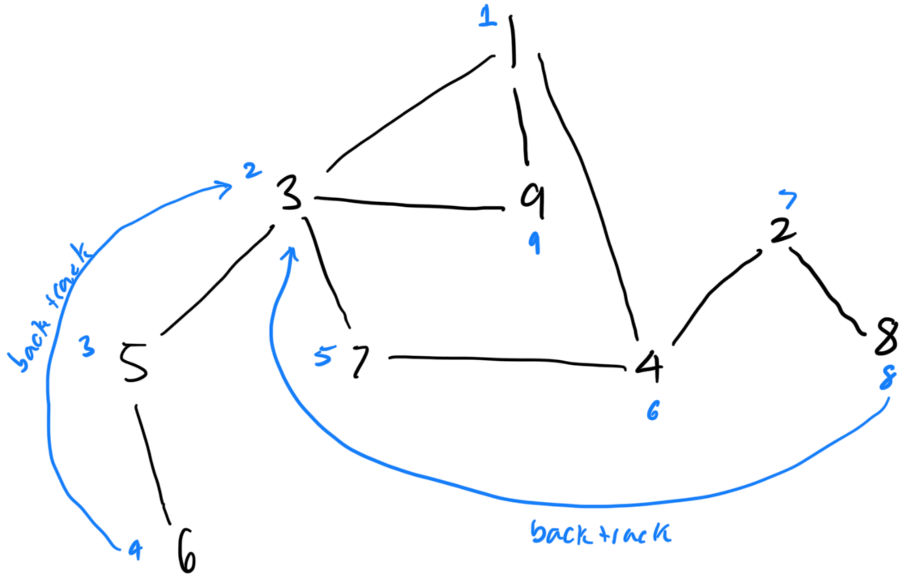
For next class!

BFT



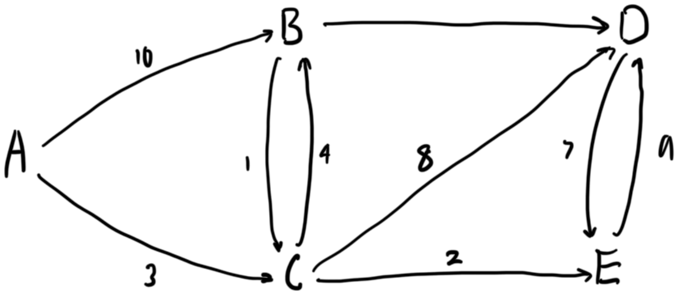
$N: [1, \underbrace{3, 9, 4,}\ \underbrace{5, 7, 2,}\ \underbrace{6, 8}]$

DFT



N= [1, 3, 5, 6, 7, 4, 2, 8, 9]

Dijkstra algorithm

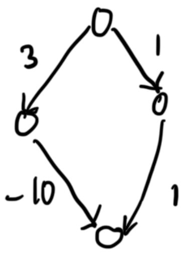


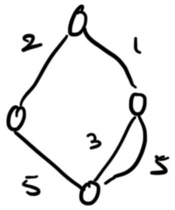
Iter	Queue	Distances					Parents				
1	{A}	0	10	3	∞	∞	N	A	A	N	N
2	{A, C}	0	7	3	11	5	N	C	A	C	C
3	{A, C, E}	0	7	3	11	5	N	C	A	C	C
4	{A, C, E, B}	0	7	3	9	5	N	C	A	B	C
5	{A, C, E, B, D}	0	7	3	9	5	N	C	A	B	C

You can reconstruct the path by tracing the parent array. For example, for $D \leftarrow A$, the shortest path is $D \leftarrow B \leftarrow C \leftarrow A$.

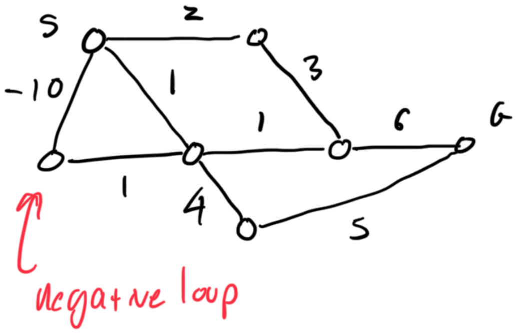
Floyd's algorithm

Sometimes, Dijkstra's algorithm fails us: you can't have multiple edges or negative weights:



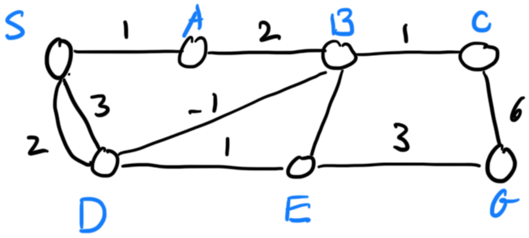


Note that you still have to assume graphs with negative loops don't exist, for example:



This is an algorithm for the shortest path from S to G but it can be modified to give the shortest path from S to anything.

Look at this graph:



This corresponds to the adjacency matrix:

$$\begin{array}{c} \begin{matrix} S & A & B & C & D & E & G \end{matrix} \\ \begin{pmatrix} 0 & 1 & \infty & \infty & 2 & \infty & \infty \\ 1 & 0 & 2 & \infty & \infty & \infty & \infty \\ \infty & 2 & 0 & 1 & -1 & 1 & \infty \\ \infty & \infty & 1 & 0 & \infty & \infty & 6 \\ 2 & \infty & -1 & \infty & 0 & 1 & \infty \\ \infty & \infty & 1 & \infty & 1 & 0 & 3 \\ \infty & \infty & \infty & 6 & \infty & 3 & 0 \end{pmatrix} \end{array}$$

We will say that if $d[i,j] > d[i,k] + d[k,j]$, that makes $d[i,j]$ *wrong*.

We start by picking an i and an j. For each node, check if $d[i,j] > d[i,k] + d[k,j]$. If it is, change $d[i,j]$.

For example, check $S \to A$. The only other possible path from S is $S \to D$, but $S \to D \to A$ is impossible.

$S \to B$ is ∞. Try $S \to A \to B = 1 + 2 = 3$ and $S \to D \to B = 2 + (-1) = 1$

$$\begin{array}{c} \begin{matrix} S & A & B & C & D & E & G \end{matrix} \\ \begin{pmatrix} 0 & 1 & 1 & \infty & 2 & \infty & \infty \\ 1 & 0 & 2 & \infty & \infty & \infty & \infty \\ 1 & 2 & 0 & 1 & -1 & 1 & \infty \\ \infty & \infty & 1 & 0 & \infty & \infty & 6 \\ 2 & \infty & -1 & \infty & 0 & 1 & \infty \\ \infty & \infty & 1 & \infty & 1 & 0 & 3 \\ \infty & \infty & \infty & 6 & \infty & 3 & 0 \end{pmatrix} \end{array}$$

$S \to C$ is ∞. Try other paths, but they all result in ∞ length as well.

Intuition:

Using what I know about edge weights already, can I pick an intermediate node between i and j that gets me from i to j.

Check AB then AC then … then ED for the shortest intermediate. This just works to find the shortest path.

This has a horrendous runtime of $\Theta(V^3)$, but it will get you the shortest path.

For path storage and then reconstruction, I believe you just store a parent array.

```python
# let d be an adjacency matrix, and n be the number of vertices
p = [[None] * n for _ in range(n)] # n x n array all null
```

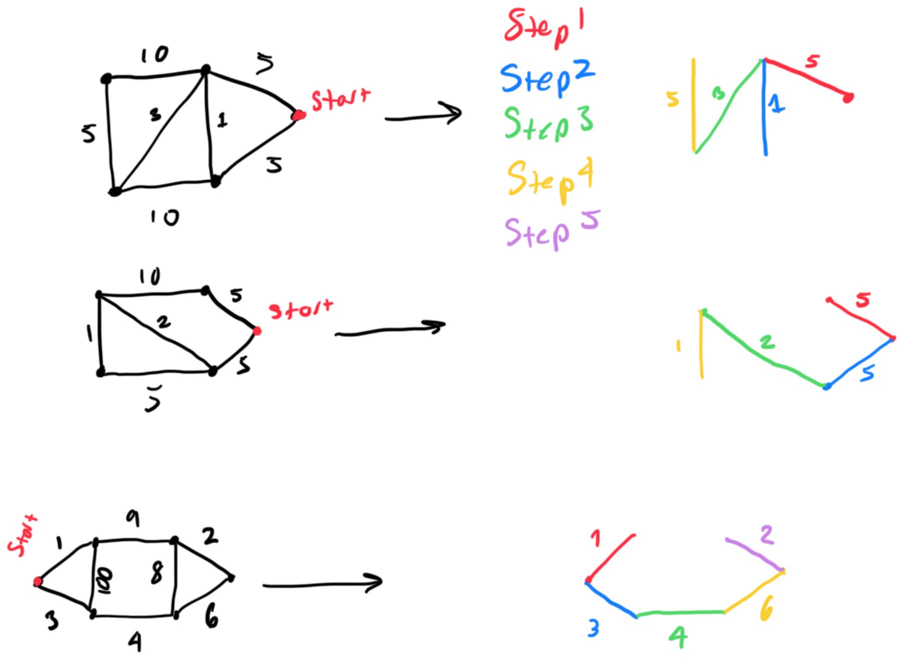
Prim's Algorithm

Spanning tree: a subgraph of a graph that contains every node and is a tree.

A minimum spanning tree is a spanning tree where the sum of the weights is less than the sum of the weights in every other spanning tree.

1. Pick a starting node that will be the head of the tree
2. Add the smallest weighted edge connected to any node in the tree as long as that does not form a cycle.

For example:



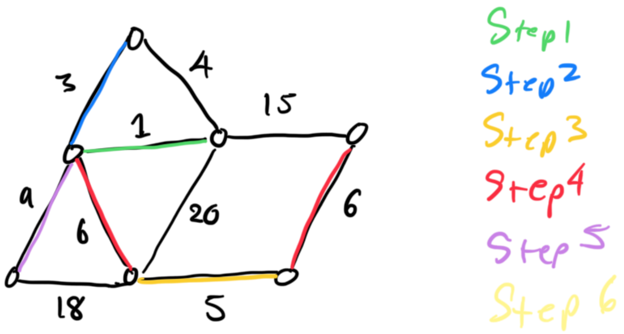
The time complexity for it when using an adjacency list and binary heap is $\Theta(E \log V)$.

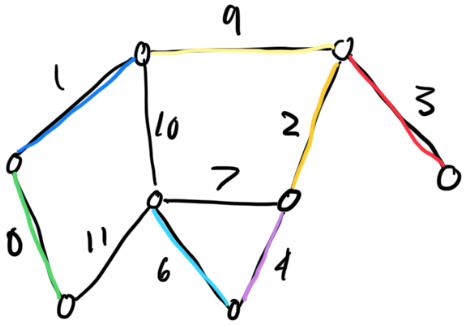
There is a very easy V^2 algorithm. If you are clever and use a Fibonacci heap, you can go down to $\Theta(E + V \log V)$.

Kruskal's Algorithm

Another greedy algorithm for finding minimum spanning trees.

Keep adding the lowest-weight edge that does not form a cycle.





How can we tell if we are forming a cycle?

We do this quickly by using a disjoint set data structure.

The time complexity of Kruskal's Algorithm is $O(E\alpha(E))$, where α is the "extremely slowly growing inverse Ackermann function."

Minimax algorithm

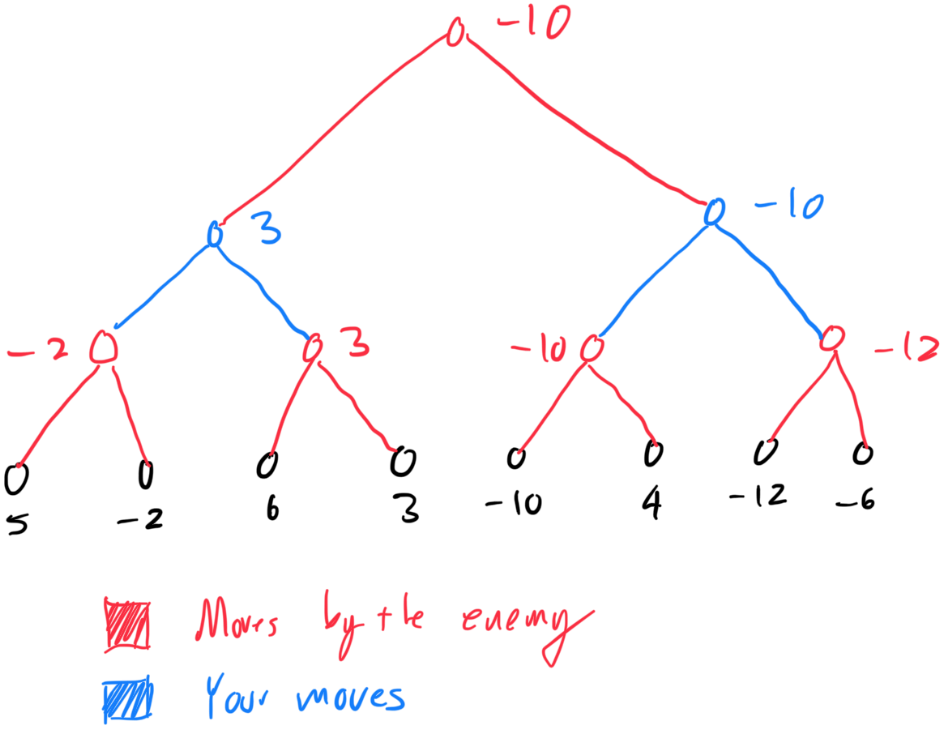
Historically, graph search used to be considered AI.

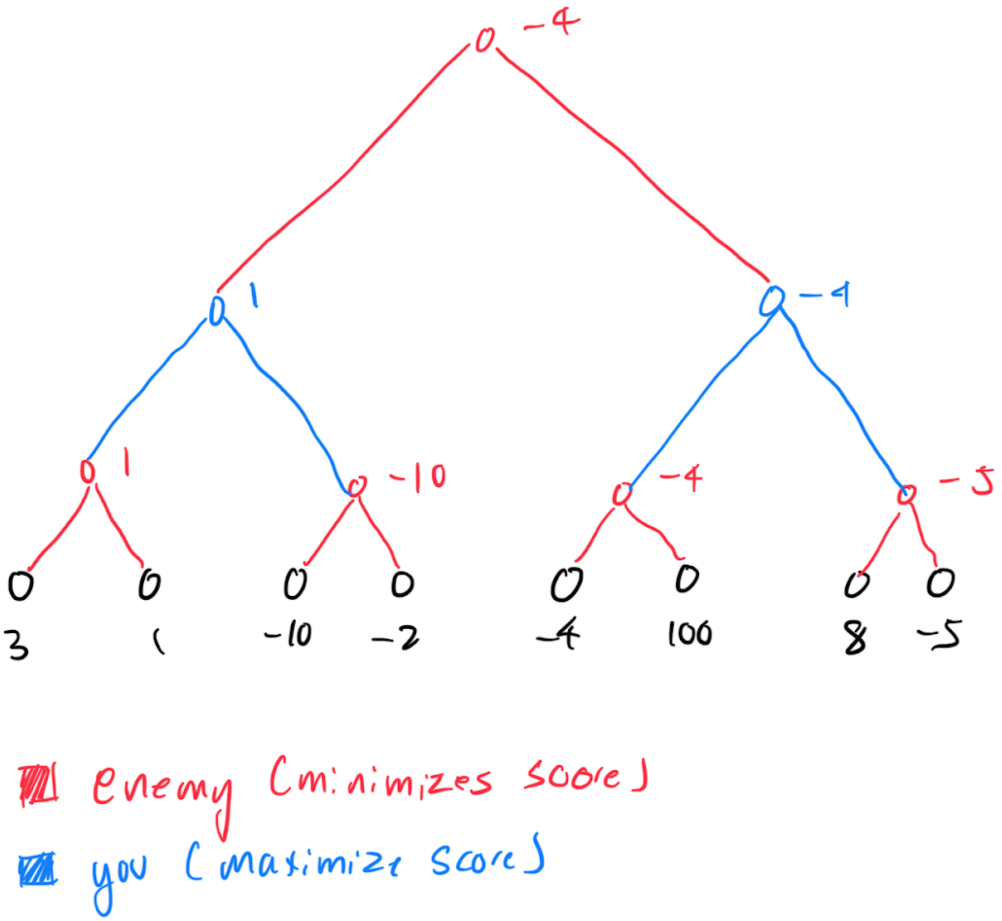
In the past, the 15 puzzle was difficult to solve. It can now be solved by converting it to a graph and then performing a breadth-first search. It is costly on memory, but it gives you the shortest path guaranteed.

Next comes Tic-tac-toe. You can't just use a normal graph search because the other player is trying to stop you.

To solve this, assume your opponent is good. Therefore, they will take the best possible move.

Starting from the terminal states, propagate the score up the tree. Because your enemy is good at the game, they will choose the score that will minimize your score. Because you are good, you will choose the maximum score possible.


-10
3
-10
-2
3
-10
-12
5
-2
6
3
-10
4
-12
-6
▨ Moves by the enemy
▨ Your moves



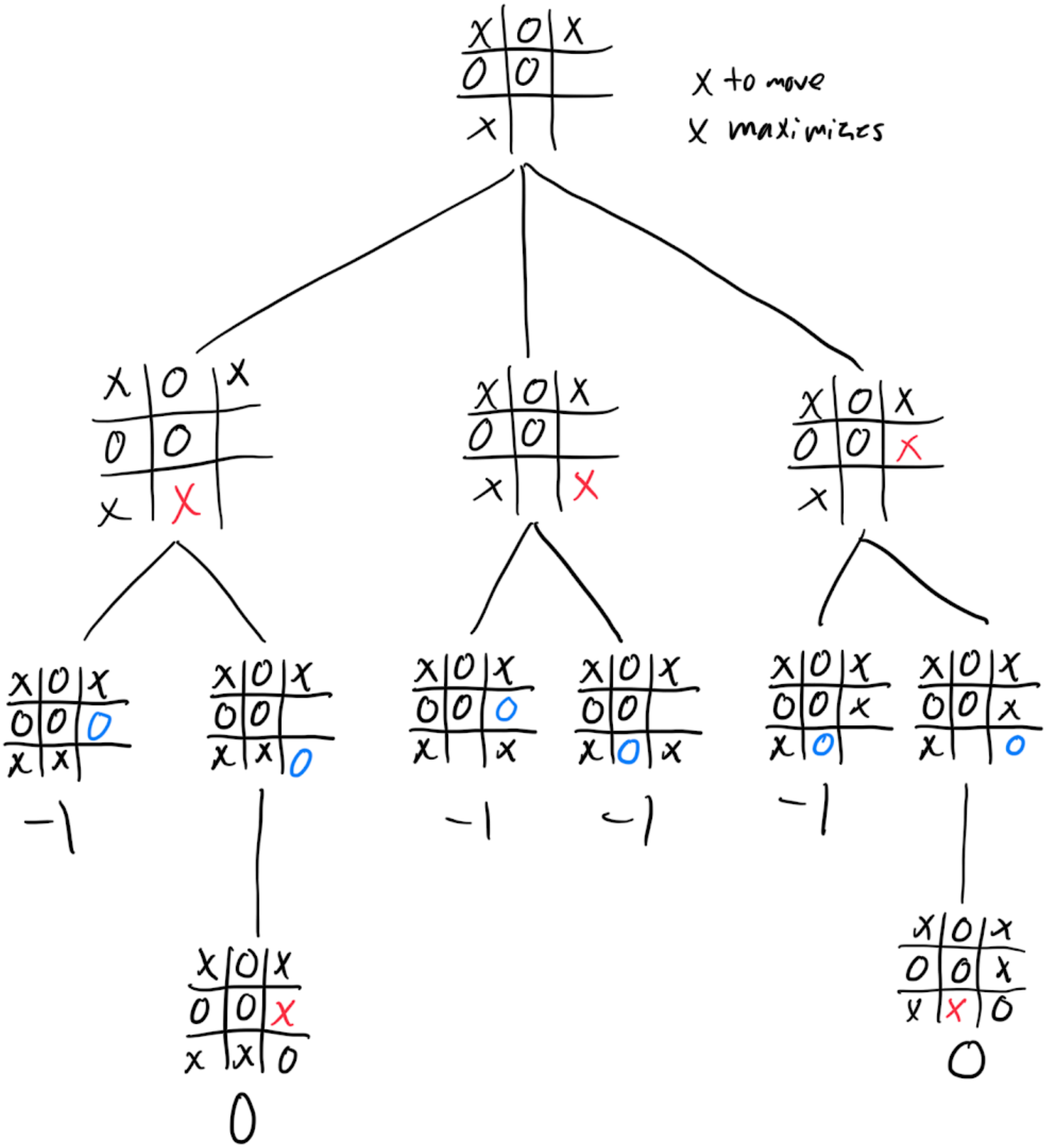
Alternatively, you can go from the top down and then end at some point when you have exhausted your time to search. Then, have a heuristic that you execute at the layer.

Heuristic-Based minimax

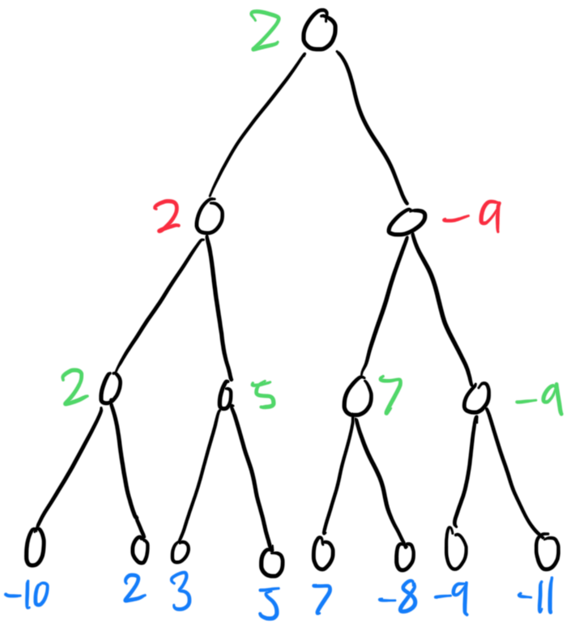
Explicitly, you build out the game tree a bunch as deeply as you can. For each leaf remaining in your tree, you compute something called a heuristic: a function that maps states to a real-valued number representing how good they are to a player.

If this function is called $h_1(s)$ for player 1 and $h_2(s)$ for player 2, because the game is zero-sum, you know that $h_1(s) - h_2(s) = 0$ and therefore $h_1(s) = h_2(s)$.

Alternate assigning nodes positive or negative multiples of the heuristic based on whether it is the maximizer's turn or the minimizer's turn.



All moves lose: -1 score



Assumptions: this is assuming that both both players have the same heuristic function.

Call the function that is the true heuristic $h^\star(t)$.

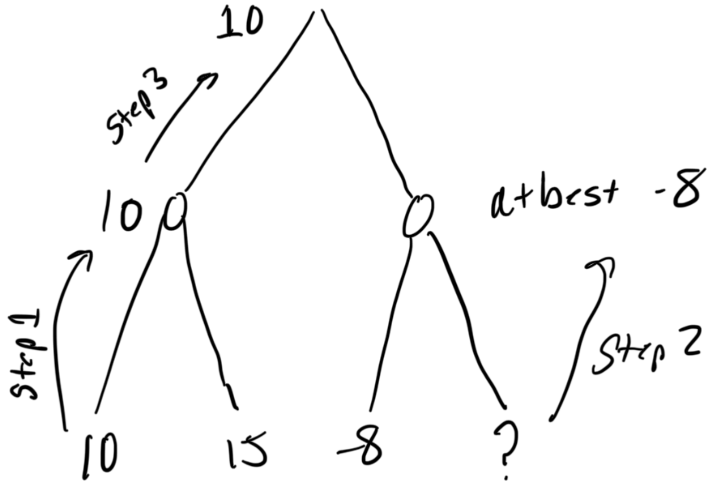
It is often very hard to get that value, so you can use an approximation instead. In chess, the number of pieces is a common heuristic. In tic-tac-toe, it is often the open lines of attack.

Runtime: We need to iterate over every node. The tree has a branching factor of b.

$1 + b + b^2 + b^3 + \cdots + b^d \to O\left(b^d\right)$

Alpha-beta pruning

It says that if you ever find a branch min can move down that has a node worse than your current best, you can stop exploring that branch:



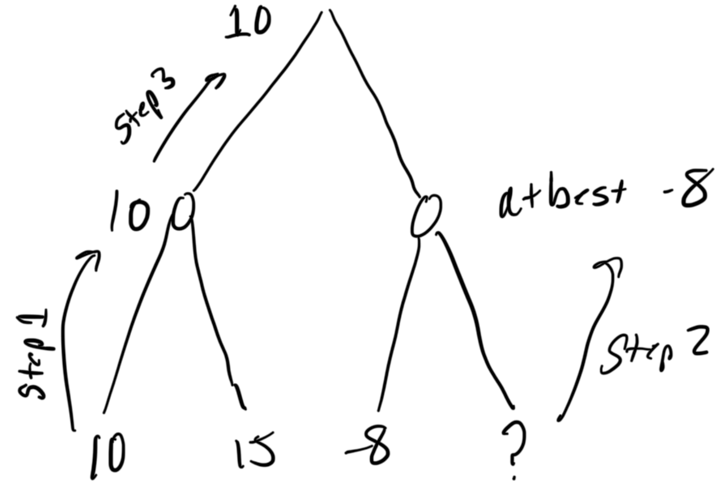
Even when we don't know the ?, we know the top of the tree will be 10.

Flaws with Minimax

If you have a suboptimal heuristic, that can be exploited. On the converse, you won't assume the other player will play poorly.

Stochastic games

The general strategy here is to compute the expected value of each choice.



Even when we don't know the ?, we know the top of the tree will be 10.

In this example, you calculate the expected value of each coin flip.

Compression

There are two types of compression:

1. Lossless compression, where you get exactly the original data back
2. Lossy compression, where you get the data back, and it could be slightly different

Example

Take the string “helloooo”. If you encode this in a fixed-length code:

```
h = 00
e = 01
l = 10
o = 11
```

This then makes it 4 times shorter.

You could also encode this with a prefix code:

```
h = 000
e = 001
l = 01
o = 1
```

```
helloooo -> 00000101011111
```

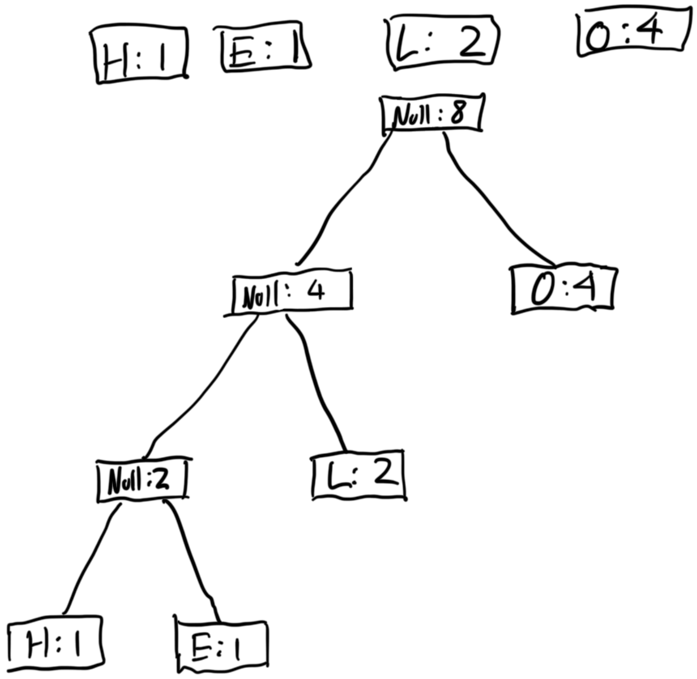
This is 4.57x smaller than the original encoding.

The algorithm for finding an optimal prefix code (Huffman encoding)

Example: "helloooo"

$$H:1, E:1, L:2, O:4$$

Put this in a max-heap:



You do this by starting with the lowest two nodes and then connecting them with their sum and a null character.

You then add the next node and connect them with their sum as well.

To do the encoding, choose a bit for left and right on the tree. Here we will choose $\text{left} = 0, \text{right} = 1$.

This results in:

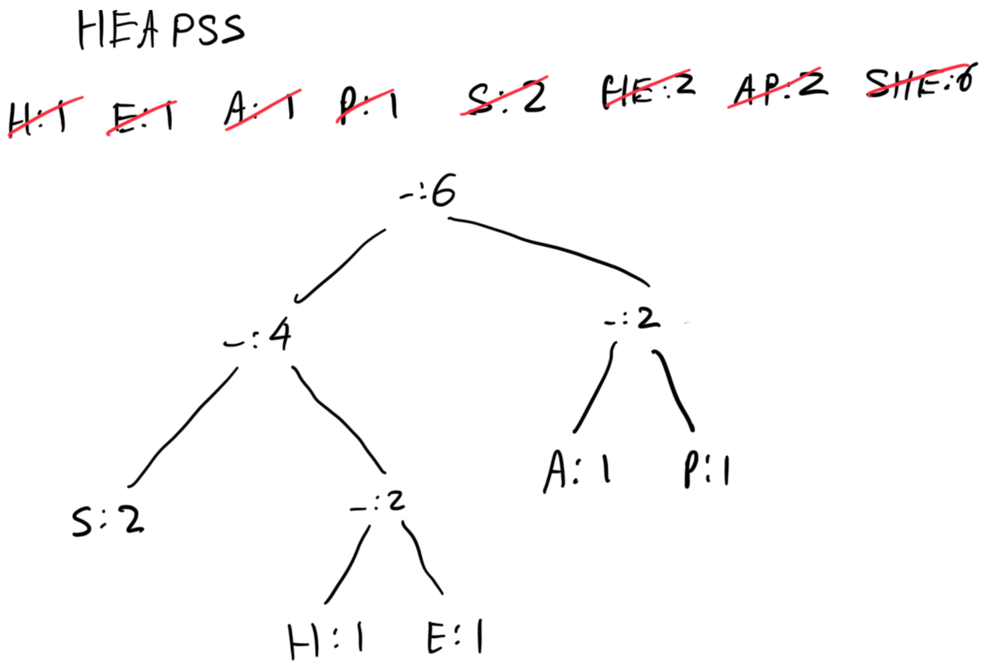
```
h = 000
e = 001
l = 01
o = 1
```

Look at that; it's exactly the same thing as we got before by eyeballing the perfect encoding.

This is mathematically proven to be the optimal encoding for character-level encoding, ignoring the storage of the tree.

This also will represent all possible minimal encodings for all possible binary trees.

Another example:


HEAPSS
~~H:1~~ ~~E:1~~ ~~A:1~~ ~~P:1~~ ~~S:2~~ ~~HE:2~~ ~~AP:2~~ ~~SHE:6~~
-:6
-:4
-:2
S:2
-:2
A:1
P:1
H:1
E:1

Pseudocode

```
for i in range(1, n):
  # extract the binary tree t1 with the smallest count from H
  # rebuild H
  # extract the binary tree t2 with the smallest count from H
  # rebuild H
  # combine t1 and t2 into t
  # insert t into H
```

Time complexity

Building this tree takes $\Theta(n \log n)$ time.

Checking vs Doing Something

Some examples of this:

- Sorting a list takes $\Theta(n \lg n)$ time. But checking that a list is sorted takes only $\Theta(n)$ time.
- Checking that a subset of a set sums to zero takes $\Theta(n)$ time. But finding that subset (with brute force) takes $\Theta(2^n)$ time.
- Check if a solution exists when given a $n^2 \times n^2$ Sudoku board.
  - How many squares exist: $n^2 \times n^2$
  - How many possibilities per square: n^2
  - Finding the solution (brute-force) takes $O\left(\left(n^2\right)^{n^4}\right)$
- Given a $n^2 \times n^2$ Sudoku board and a solution, how long does it take to verify the solution is correct: $O(n^4)$.

Decision Problems

A decision problem is a problem that takes an input, called an instance, and outputs yes or no.

Call the decision problem Q, and the instance I. Then $Q(I) = \text{yes} \vee Q(I) = \text{no}$.

Examples

Given 2 numbers, is their product greater than zero?

$$I = \{3, 4\} \Rightarrow Q(I) = \text{yes}$$
$$I = \{-2, 7\} \Rightarrow Q(I) = \text{no}$$

Given a list, is it sorted?

$$I = [1, -1, 5, 7] \Rightarrow Q(I) = \text{no} \; I = [3, 4, 9, 13] \Rightarrow Q(I) = \text{yes}$$

Given a graph, is there a cycle?

Proof/Witness

Given a decision problem and an instance with $Q(I) = \text{yes}$, then a proof or witness is evidence that $Q(I) = \text{yes}$ is true.

For example: Does the graph have a cycle:

Given a set, is there a subset that sums to 0?

Note that this does not mean that finding a witness is easy. For example, with sudoku, even if $Q(I) = \text{yes}$, finding the witness is difficult.

If you present a faulty witness, that doesn't mean that $Q(I) = \text{no}$.

Turing Machine

It is a math formalization of a computer. Non-deterministic Turing machines can explore all branches at once.

P

Things in P:

- Everything we've done so far in class basically

Things not in P:

- Finding all permutations of something ($n!$)
- Finding all subsets of a set (2^n)

The definition: The set of decision problems that a deterministic turning machine can solve in polynomial time.

NP

Definition: The set of decision problems such that a polynomial time algorithm on a deterministic Turing machine takes in the problem and a witness, and if the witness is valid, it will output true in polynomial time.

Basically, if you are given an answer, can you check that that answer is correct in polynomial time?

Alternatively, NP is when a nondeterministic Turing Machine can solve a problem in polynomial time.

NP-Complete

1. Decision problem: true/false answer
2. Yes answers can be demonstrated and then shown in polynomial time
3. A brute-force solution exists
4. It can be used to simulate every other NP-complete problem.

Traveling Salesmen Problem

Given a set of cities, can you construct a tour that has a path length of less than k?

To check, simply compute the length of the solution.

SAT Solving

Given a series of boolean expressions of length 3, can I find assignments of all variables that make the statement true? (Note that there can be more than 3 variables).

To check, simply evaluate at the variables given.

Graph Coloring

Given a graph, can you color the nodes so that no two identically colored nodes are next to each other, given k colors?

Note that $k < 4$, because of the four color theorem.

NP-Completeness

- A problem is in P if there exists a polynomial time algorithm to solve it.
- A problem is in NP if it is a decision problem that can be verified with a witness in polynomial time.
- NP-complete is a class of problems in NP where if we can prove that if there is a polynomial time algorithm for solving one, that algorithm can be used to solve all of the NP-complete problems.

Reductions

If somebody asks the question, "Is the answer to problem A yes for this input?" instead of solving problem A, transform it into a different problem and then evaluate the other problem with your transformed input.

Reductions are typically proofs that if you can solve problem A in $O\left(n^k\right)$, then you can solve problem B in $O\left(n^k\right)$.

Steps

1. Assume that problem A is solvable in $O\left(n^k\right)$ time.
2. Take an arbitrary instance of problem B and then apply some transformations to turn it into an instance of problem A.
3. Evaluate that transformed instance in problem A, and return the answer.

Note that the transformations themselves must be $O\left(n^k\right)$ as well.

Example

Problem 1: Independent set: Is there a set of nodes in a graph where no two of them are adjacent?

Problem 2: 3-SAT: given a conjunction of disjunctions of size 3, is there an assignment that makes the expression true?

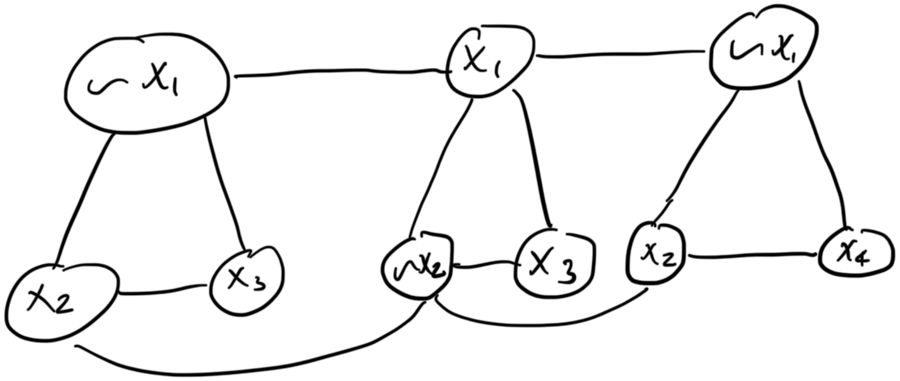
$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_4 \vee x_2 \vee x_1) \wedge (x_4 \vee x_5 \vee x_5) \Rightarrow \begin{cases} x_1 = \text{true} \\ x_2 = \text{false} \\ x_3 = \text{false} \\ x_4 = \text{true} \\ x_5 = \text{false} \end{cases}$$

We will show that if we can quickly solve the independent set problem, we can solve 3-SAT quickly.

Ways to solve 3-SAT:
1. Find a way to assign each variable such that the whole formula evaluates to true.
2. Pick a variable from each clause and see if you assign it without finding any conflicts.

Create a graph representing the problem, with nodes within a disjunction connected and negations across disjunctions connected.

$$(\sim x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \sim x_2 \vee x_3) \wedge (\sim x_1 \vee x_2 \vee x_4)$$



Take the graph and find the independent set. If there is one of the proper size (3 in this case), the problem is satisfiable.

Therefore, because we can do this for any 3-SAT, the 3-SAT problem is polynomial reducible to the independent set problem.

What you should know for the exam

- Know the basics:
  - Basic math (limits, sums (be careful; be verbose), log rules, etc)
- Big O
  - Definition of big O, limit theorems, master theorem, recurrence relations, digging down
- What are the prereqs for using an algorithm?
  - Dijkstra's requires positive weights
- Why do the algorithms exist
- Be able to recreate anything done in class
- Review the review guide

- Look at previous exams and study what you missed in particular
- Make your own exam questions
- Know the names of things in graphs
  - Walk, path, cycle, loop, simple, weighted, trail, etc

Below this, there are spoilers for the review guide. You should look at the review guide and try to answer the questions yourself first, and you should really try. The process of learning involves failure, and there are no grades here.

From Max:
This review guide is a non-exhaustive (but close to exhaustive!) list of all topics covered in class, as well as what you should know about them. It includes a number of potential questions to ask yourself, which are not practice exam questions. I know I say this constantly, but memorizing the answers to each of these won't really help you. Knowing the answers to these is what you're going for. Don't look them up. Sit down with a piece of paper and figure them out. If you can't, there is a gap in your knowledge that you will need to fill more thoroughly.

Covering the Review Guide and answering questions

Sorts

Know the best, average, and worst-case runtimes for sorts and understand why they are the way they are. Know why you would use some particular sort over others.

Bubble sort

1. Why would you use bubble sort? What advantages does it have over something like quicksort or merge sort?
   - Bubble sort goes through the list n times, comparing adjacent elements and swaps if they are in the wrong order
   - It is far simpler, so for small lists, it can be much faster
   - It is stable, unlike quicksort, and uses $O(1)$ auxiliary memory, unlike merge sort.
   - It swaps but only moves things over a short distance.
2. What is true after the first pass of bubble sort?
   - The last element would now be correct

Insertion sort

1. Do you actually know the difference between this one and selection sort?
   - Insertion sort will take the next element and inserts it into the sorted part of the array, vs. selection sort, which finds (selects) the correct element and puts it in the right place

Selection Sort

1. What is true after the first pass of selection sort?
   - The first element is in the correct place, but that is all that has happened.
2. Does selection sort use more or fewer operations than bubble sort? Why or why not?
   - It uses fewer because bubble sort uses n^2 operations, and selection sort modifies the lower bound, so at worst, it will use $\left(\frac{n}{2}\right)n$ operations. See this video.
3. Why use selection sort?
   - It puts the elements in the correct place directly. Therefore, it uses a minimal number of swaps.

Merge Sort

1. Be able not just to do merge sort but also to merge. You should know the recurrence for merge sort off the top of your head.
2. Do you actually know the difference between this one and selection sort?
    1. Merge sort splits the list in half repeatedly until you have a single element, and then merges those lists together. Because when merging, both lists are already sorted, so you can compare the first two elements in each sublist when forming a combined list.
3. Why is merge sort parallelizable?
    - Run each merge step on its own thread. This means merge sort takes 2n - 1 = O(n) time with O(n) threads. (@1597)

Quicksort

1. Know how partition works and how it interacts with the best and worst case. Make sure you remember what a pivot is.
    - The pivot splits the list in half.
2. What is the recurrence for Quicksort? Is it different in the best, worst, and average case?
    - $T(n) = T(k) + T(n - k) + \Theta(n), 0 \leq k < n$. On average, k varies continuously from the left to the right, and so sorting takes $\Theta(n \log n)$ on average, $\Theta(n \log n)$ best case $\left(k = \frac{n}{2}\right)$ and $\Theta(n^2)$ worst case ($k = n - 1$).
3. Why is having the partition be the median better?
    - The recurrence will be $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$, and so it will always be good and reduce the complexity of the problem.

Heap sort

Make sure you know what a heap is, how to build one, and how to use it as a sort. There are a number of different operations you might have to perform with this one (removing the min, building the heap, etc).

- What is a heap? How is it different from a binary tree?
    - A heap does not care about the order of the trees below, just that the top node is above the
- Why do we send something to the top of the heap and let it float down when we remove the min?
- Does heap sort have any worst cases or best cases?
    - It’s always $\Theta(n \lg n)$

See the shared google doc for further answers!