Contents

1 Algorithms of Arithmetic **2**

1.1 Lecture 1 . . . 2

1.1.1 Addition/Multiplication . . . 2

2 Divide and Conquer Algorithms **3**

2.1 Lecture 2 . . . 3

2.2 Lecture 3 . . . 4

2.2.1 Recurrences and Master Theorem . . . 4

2.2.2 Matrix Multiplication . . . 4

2.2.3 Sorting . . . 5

2.2.4 Selection/Medians . . . 7

2.3 Lecture 4 . . . 9

2.3.1 Polynomial Multiplication . . . 9

2.3.2 Cross Correlation . . . 12

3 Graph Algorithms **14**

3.1 Lecture 5 . . . 14

3.1.1 Graph Representation . . . 14

3.1.2 Depth First Search (DFS) . . . 14

3.2 Lecture 6 . . . 18

3.2.1 Topological Sort . . . 18

3.2.2 Strongly Connected Components . . . 19

3.3 Lecture 7 . . . 21

3.3.1 Single Source Shortest Paths . . . 21

3.3.2 Weighted Graphs . . . 23

1 Algorithms of Arithmetic

1.1 Lecture 1

1.1.1 Addition/Multiplication

First, let us consider the problem of adding two n-bit numbers, a and b. If both are a different amount of bits from each other, we can pad 0's to the left of the smaller one until it reaches the length of the larger one (note that padding 0's doesn't change the sum). The grade-school algorithm (add column-wise and do carries) has to compute at most $n + 1$ additions, which we can assume are all constant-time. Thus, we say addition has complexity linear in n, or $\Theta(n)$ (we drop the constant because 1 can pale in significance to how great n can grow).

But can we do better for addition? The answer is no; since it takes $n + 1$ bits to write our answer, any algorithm returning the sum must require $n + 1$ operations. Thus, we say as a lower bound, addition is $\Omega(n)$ (the full definitions of these terms will come shortly).

Now we turn to the problem of multiplication. How fast is the grade school algorithm? First, we multiply digit-wise and then do a bunch of additions. In binary, multiplying by a 0 or 1 and then right-padding with 0s (bitshifting) corresponds to a constant time operation for each bit of b. Then, we have to add together n potentially $2n$ bit numbers. This adds n^2 time. Thus, the runtime is quadratic in n or $\Theta(n^2)$.

Now, can we do better for multiplication? It turns out we can. We do this by le

2 Divide and Conquer Algorithms

2.1 Lecture 2

TODO: Fill in this section

2.2 Lecture 3

2.2.1 Recurrences and Master Theorem

The idea of divide-and-conquer algorithms are to divide the input inot smaller parts, recurse on parts, and combine the parts to build an answer.

To analyze the runtime of divide-and-conquer algorithms, it is useful to derive the following result.

Theorem 2.1 (Master Theorem)
Suppose we have a recurrence

$$T(n) = aT(\frac{n}{b}) + cn^d$$

then, we have

$$T(n) = \begin{cases} \Theta(n^d) & a < b^d \\ \Theta(n^d \log n) & a = b^d \\ \Theta(n^{\log_b a}) & a > b^d \end{cases}$$

Master's theorem can be shown by drawing a recursion tree, and then summing up the work done in each level (proof omitted for brevity). We can also think of the cases as symbolizing the following:

Case	Interpretation
$a < b^d$	The root does most of the work
$a = b^d$	The root and the leaves do an equal amount of work
$a > b^d$	The leaves do most of the work

2.2.2 Matrix Multiplication

Our first example of a divide and conquer algorithm is matrix multiplication.

Consider multiplying two n-by-n matrices A and B.

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{bmatrix}$$

Then the resultant C has entries given by:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

The natural implementation is then to to loop this summation over i and j. This means this will be three nested loop (a loop is needed for the summation). In flops, this runs in $\Theta(n^3)$ operations.

We can try to break our input instead into $n/2$-by-$n/2$ blocks, as shown:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

To find the runtime of this algorithm, let us realize there are 8 multiplications (recursively) and then finally a Θn^2 addition at the end. This means our recurrence is:

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

Note that for our Master theorem setup, $8 > 2^2$, so we have

$$T(n) = n^{\log_2 8} = n^3$$

so this is no better than our naive approach.

Using a similar realization to Karatsuba, Strassen in 1969 found the following:

Algorithm 2.1 (Strassen's Algorithm)
Consider two matrices X and Y which are both n-by-n. Break them up into block matrix form of $n/2$-by-$n/2$ matrices as follows:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Define the following:

$$\begin{aligned} P_1 &= A(F - H) \\ P_2 &= (A + B)H \\ &\vdots \\ P_7 &= (A - C)(E + F) \end{aligned}$$

Then,

$$Z = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

Analyzing the runtime, we see that there are 7 $n/2$-by-$n/2$ multiplications, and some n^2 additions, meaning the total runtime is

$$T(n) = 7T(n/2) + \mathcal{O}\left(n^2\right)$$

which Master theorem brings to

$$T(n) = \mathcal{O}\left(n^{\log_2 7}\right) \approx \mathcal{O}\left(n^{2.81}\right)$$

However, Strassen has such a big constant factor that the normal n^3 algorithm is still the most widely used.

2.2.3 Sorting

Consider the problem of sorting a length n array A.

Algorithm 2.2 (MergeSort)
First, we define a procedure MERGE that takes two sorted lists and merges them in linear time. To do this, we first keep a pointer on both lists that starts at the beginning of each list. We then compare the pointed-to elements of each list. The lesser element is then added to the output, and the pointer of the list with that element is incremented by one place. This keeps going until all the elements are used. We then use merge to divide-and-conquer the list as follows:

function MERGE-SORT($A[1 \dots n]$)
$B \leftarrow$ MERGE-SORT($A[1 \dots \frac{n}{2}]$)

$C \leftarrow$ MERGE-SORT$(A[\frac{n}{2}+1 \ldots n])$
return MERGE(B, C)

We get the following recurrence for MERGE-SORT:

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

, which through master theorem gives us a running time of

$$T(n) = \Theta(n \log n)$$

Another way of implementing this is a bottom up approach:

Algorithm 2.3 (Iterative Merge Sort)
function MERGE-SORT-ITER$(A[1 \ldots n])$
$Q \leftarrow$ Divide A into n lists of size one
while Q.size() > 1 **do**
$X, Y \leftarrow Q$.pop(), Q.pop()
Q.push(MERGE(X, Y))
return Q.pop()

Now we can think about the runtime of this algorithm. Think of the algorithm as running in phases. Phase 0 is when lists popped have size 1. Phase 1 is when lists popped have size 2. Phase i is when lists popped have size 2^i. Note that in each phase, each element is looked at exactly once. Thus, the total runtime must be proportional to $n \cdot$ number of phases. How many phases are there? There are $\log n$ phases, giving us the same $\Theta(n \log n)$ runtime!

The best way to see this is with an example:

Example 2.1
Suppose our initial list is:

$$A = \begin{bmatrix} 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{bmatrix}$$

We then split this into sublists of size one in our Q and start iterating:

$$\begin{aligned} Q &= [[8], [7], [6], [5], [4], [3], [2], [1]] \\ Q &= [[6], [5], [4], [3], [2], [1], [7, 8]] \\ &\vdots \\ Q &= [[7, 8], [5, 6], [3, 4], [1, 2]] \\ Q &= [[3, 4], [1, 2], [5, 6, 7, 8]] \\ Q &= [[1, 2, 3, 4], [5, 6, 7, 8]] \\ Q &= [[1, 2, 3, 4, 5, 6, 7, 8]] \end{aligned}$$

Our list has been sorted!

Can we sort faster than $n \log n$? It turns out we cannot do any better with an algorithm that uses comparisons to sort (the problem is $\Omega(n \log n)$).

Theorem 2.2 (Fastest Shorting in Comparison Model, Lower Bound)
We will show that $\Omega(n \log n)$ comparisons are needed even if promised A is some permutation of $\{1, \ldots, n\}$ (all distinct as well).

Proof
The first comparison such an algorithm might make might be: is $A_i < A_j$? Then we branch off into two cases for each, where we require another comparison. We can construct a binary tree that models the situation.
First, notice that each leaf is when the algorithm terminates. Note that since every run of the algorithm with a different input produces a different output permutation of the input, there must be at least $n!$ leaves. Consider the maximum depth of this tree T. There are at most 2^T leaves, meaning that $2^T \geq n!$. This means

$$T \geq \log(n!)$$
$$T = \Omega(n \log n)$$

The last claim can be shown by realizing

$$n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$
$$\log(n!) \geq \frac{n}{2} \log\left(\frac{n}{2}\right)$$
$$\log(n!) \geq \frac{n}{2} \log n - \frac{n}{2}$$
$$\log(n!) = \Omega(n \log n)$$

However, there are way more operations than comparisons that can be used for sorting. An example of this is counting sort: if you have n integers and all the integers have values between 1 and b, then you can sort in $\Theta(n + b)$, by just keeping a hashmap of all the elements that fit in each value "bucket" between 1 and b.

The Word RAM model: suppose your machine can store words of size w and you can do any common C operations. The fastest known algorithm following this model (not just comparisons) is: $\mathcal{O}\left(n\sqrt{\log \log n}\right)$. This is also a randomized algorithm.

There are two types of randomized algorithm (which will be revisited). A Monte Carlo randomized algorithm is one whose output may be incorrect with small probability. A Las Vegas algorithm is one whose runtime is fast in expectation, but may be slow with small probability.

2.2.4 Selection/Medians

We now consider the problem of selection. Suppose we want to select the k smallest integer in a list A. Without loss of generality, assume all elements of A are distinct (since we could replace $A[i]$ with the tuple $(A[i], i)$). For selection, there is Quick Select, which we will explore later. Notably, quick select requires knowing the median in linear time. We will instead focus on that problem, (the same as the selection problem for $k = n/2$).

Algorithm 2.4 (Median of Medians)
Take an array A. Then break up the array into subarrays of size 5. Next, we will recursively compute the median of each subarray. Note that this takes constant time to complete since 5 is a constant. Now we have a $N/5$ size array. We then find the median recursively of this smaller problem. Call this median m_1.
Now, we change the array such that all the elements bigger than m_1 end up on the right of m_1, elements smaller than m_1 end up on the left, and m_1 is in the middle of these two parts (this only requires a linear scan). If m_1 is in position less than $n/2$, then the true median sits on its right, so we can recurse on the right half. If m_1 is in position greater than $n/2$, then the true median sits on its left, so we can recurse on the left half. Finally, if m_1 is at exactly position, $n/2$, then we have found the median.
The claim is that at least 30% of the elements are filtered out by comparisons to m_1. To show this, consider m_1 compared to the other medians. Note that it is bigger than 3 of the elements in every median it is bigger than, so, it is bigger than $\frac{3}{5} \cdot \frac{N}{10} = \frac{3}{10}$ of the elements. Thus, if m_1 is in the first half of the array, then it will filter out at least $\frac{3}{10}$ of the numbers. Similarly, you can make a symmetric argument that if m_1 is in the second half, then

it must be less than $\frac{3}{10}$ of the numbers and thus will filter out 30% of them. Either way, we can then produce the following recurrrence:

$$T(n) \leq T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + \Theta(n)$$

This recurrence gives $T(n) = \mathcal{O}(n)$. We can give an inductive argument:

Proof
We will show that $T(n) \leq Bn$ for sufficiently large B, which will imply our big-$\mathcal{O}$ runtime.
Base Case: if n is 1, then we just return the input, so if B is greater than the time needed to return then the base case holds.
Inductive Hypothesis: Suppose that the claim holds for $k < n$.
Inductive Step: By the recurrence and the inductive hypothesis, we have that

$$T(n) \leq B\frac{7n}{10} + B\frac{n}{5} + Cn$$

where C is some other constant. Now, we have

$$\begin{aligned} T(n) &\leq \left(\left(\frac{7}{10} + \frac{1}{5}\right)B + C\right)n \\ &\leq \left(\frac{9}{10}B + C\right)n \\ &\leq Bn \end{aligned}$$

as long as $C \leq \frac{B}{10}$. Since C is fixed, we can set $B = 10C$ to make this true. ∎

2.3 Lecture 4

2.3.1 Polynomial Multiplication

Suppose we have two polynomials as inputs:

$$A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{d-1}x^{d-1}$$

$$B(x) = b_0 + b_1x + b_2x^2 + \cdots + b_{d-1}x^{d-1}$$

Then we want the output polynomial C in the following form:

$$C(x) = c_0 + c_1x + \cdots + c_{2d-2}x^{2d-2}$$

Define $N = 2d - 1$ for simplicity, and notice that these can all be considered $N - 1$ degree polynomials if we pad A and B with 0 coefficients on higher order coefficients.

There is a relationship between polynomial and integer multiplication. Given integers α, β, if we want $\gamma = \alpha \times \beta$, we first write them digit-wise as

$$\alpha = \alpha_{N-1}\alpha_{N-2}\ldots\alpha_0$$

$$\beta = \beta_{N-1}\beta_{N-2}\ldots\beta_0$$

$$A(x) = \alpha_0 + \alpha_1x + \cdots + \alpha_{N-1}x^{N-1}$$

$$B(x) = \beta_0 + \beta_1x + \cdots + \beta_{N-1}x^{N-1}$$

Note that $\alpha = A(10)$ and $\beta = B(10)$, and $\gamma = (A \cdot B)(10)$ plugging in integers for the polynomials is fairly fast (just some additions and multiplication). This shows that integer multiplication and polynomial multiplication are fairly connected.

Algorithm 2.5 ("Straightforward" Algorithm for Polynomial Multiplication)

$$C(x) = c_0 + c_1x + \cdots + c_{2d-2}x^{2d-2}$$

What are these coefficients in terms of a_i and b_i?

$$\begin{aligned} c_0 &= a_0b_0 \\ c_1 &= a_0b_1 + a_1b_0 \\ &\vdots \\ c_k &= \sum_{j=0}^{k} a_jb_{k-j} \end{aligned}$$

Then the algorithm looks something like this:

- Loop over $k = 0$ to $N - 1$
  - Compute c_k with a loop from $j = 0$ to k

Note that this algorithm runs $\Theta(N^2)$.

However, we can do better, since integer multiplication is close to polynomial multiplication.

Algorithm 2.6 (Karatsuba for Polynomials)
Call:

$$\begin{aligned} A_l(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{N/2-1} x^{N/2-1} \\ A_h(x) &= a_{N/2} x^{N/2} + \cdots + a_{N-1} x^{N-1} \\ B_l(x) &= b_0 + b_1 x + b_2 x^2 + \cdots + b_{N/2-1} x^{N/2-1} \\ A_h(x) &= b_{N/2} x^{N/2} + \cdots + b_{N-1} x^{N-1} \end{aligned}$$

Note that $A(x) = A_l(x) + x^{N/2} A_h(x)$ and $B(x) = B_l(x) + x^{N/2} B_h(x)$. Using the Karatsuba trick, you see that you need 3 multiplications, giving the recurrence:

$$T(N) \leq 3T\left(\frac{3}{2}\right) + \Theta(N)$$

which solves to: $T(N) = \Theta(N^{\log_2 3})$

Here is a fact from elementary algebra:

Note 2.1 (Polynomial Interpolation)
A degree $< N$ polynomial is fully determined by its evaluation on N distinct points.

Proof
Here is an argument for why interpolation works. Represent C as the vector

$$\begin{bmatrix} c = c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix}$$

Then consider the following matrix-vector multiplication:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{N-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N-1} & x_{N-1}^2 & \dots & x_{N-1}^{N-1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix}$$

I.e. we can call this $Vc = y$, solving this equation is only possible if V is full rank.
Fact for "Vandermonde" matrix V:

$$\det\{V\} = \prod_{i<j} (x_i - x_j) \neq 0$$

so we can solve $c = V^{-1} y$, so there is a unique polynomial that interpolates.

This gives rise to the following idea: rather than multiplying directly, we instead evaluate $C(x_0), C(x_1), \dots, C(x_{N-1})$ for distinct x_i.

To do this evaluation, just evaluate $A(x_i)$ and $B(x_i)$, then finally combine to get $C(x_i)$. Finally, interpolate to set back coefficients from C in terms of these points.

However, interpolation is way too slow. You have to use inversion which takes $\mathcal{O}(n^3)$ flops. Instead, what if we choose V carefully such that it's faster to invert?

Let us establish some types:

1. The Discrete Fourier Transform (DFT) is a **matrix**.
2. The Fast Fourier Transform (FFT) is a **algorithm**.

Definition 2.1 (Discrete Fourier Transform (DFT))
Define $\omega = e^{2\pi\sqrt{-1}/N}$ (primitive root of unity). Now define the DFT matrix F such that $F_{ij} = (\omega^i)^j = \omega^{ij}$.
Imagine evaluating a polynomial at points $1, \omega, \omega^2, \ldots, \omega^{N-1}$
This gives the Vandermonde matrix:

$$V = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \ldots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \vdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

i.e. the DFT matrix.

Here is the most important property of the DFT matrix:

Note 2.2 (Inverse of the DFT matrix)

$$F^{-1} = \frac{1}{N}\overline{F}$$

Note that we can view the following algorithm in two lenses: either a fast way to multiply by F^{-1}, or a fast way to calculate $P(\omega), P(\omega^2), \ldots$. We take the latter interpretation.

Algorithm 2.7 (Fast Fourier Transform (FFT))
The goal of this algorithm is to multiply by the DFT quickly. We will take a polynomial interpretation. Consider our polynomial, assuming N is a power of 2:

$$\begin{aligned} P(z) &= p_0 + p_1 z + p_2 z^2 + \cdots + p_{N-1} z^{N-1} \\ &= \left(p_0 + p_2 z^2 + p_4 \left(z^2\right)^2 + \cdots + p_{N-2}\left(z^2\right)^{N/2-1}\right) + z\left(p_1 + p_3 z^2 + p_5\left(z^2\right)^2 + \ldots\right) \\ &= P_{\text{even}}(z^2) + z P_{\text{odd}}(z^2) \end{aligned}$$

For all N roots of unity, squaring any of them will just give you another Nth root of unity. In fact, it'll give you a $N/2$ root of unity. This means that we only have to evaluate P_{odd} and P_{even} on $N/2$ roots of unity, which means our recurrence becomes:

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

which solves to:

$$T(N) = N \log N$$

Finally, we give an algorithm for our initial problem of multiplying polynomials.

Algorithm 2.8 (Polynomial Multiplication Via FFT)
First, we use the Fast Fourier Transpose to compute $\hat{a} = Fa$ and $\hat{b} = Fb$.
Then, for $i = 0$ to $N - 1$, we compute, $\hat{c_i} = \hat{a_i} \times \hat{b_i}$.
Finally, we have to bring c back into the original basis, i.e. compute $c = F^{-1}\hat{c}$. To do this, notice

$$\begin{aligned} c &= \frac{1}{N}\overline{F}\hat{c} \\ &= \frac{1}{N}\overline{F\overline{\hat{c}}} \end{aligned}$$

which requires just one more use of FFT to multiply $F\overline{\hat{c}}$.
The runtime is thus dominated by the 3 FFTs, giving us: $\mathcal{O}(N \log N)$ runtime, assuming we can multiply/add/-conjugate complex numbers in constant time.

It may seem paradoxical that since it takes N^2 entries to write down F, how come we are faster? The heart of this is that we **never write down** F. Instead, we just use FFT to multiply F by a vector (quickly).

2.3.2 Cross Correlation

Suppose you have two vectors: x and y:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-1} \end{bmatrix}$$

$$y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

where $n \geq m$. Suppose we want all shifted dot products of x with y, i.e.

$$\begin{aligned} &x_0y_0 + x_1y_1 + \cdots + x_{m-1}y_{m-1} \\ &x_0y_1 + x_2y_2 + \cdots + x_{m-1}y_m \\ &\vdots \end{aligned}$$

However, we can reduce cross correlation to a problem we have already solved, polynomial multiplication!

Algorithm 2.9 (Cross Correlation Via Polynomial Multiplication)
Define the following:

$$\begin{aligned} X(z) &= x_{m-1} + x_{m-2}z + \cdots + x_0z^{m-1} \\ Y(z) &= y_0 + y_1z + y_2z^2 + \cdots + y_{n-1}z^{n-1} \\ Q(z) &= (X \cdot Y)(z) = q_0 + q_1z + \ldots \end{aligned}$$

Then let us investigate the coefficients of Q:

$$\begin{aligned}
q_0 &= x_{m-1}y_0 \\
q_1 &= x_{m-1}y_1 + x_{m-2}y_0 \\
&\vdots \\
q_{m-1} &= x_{m-1}y_{m-1} + \cdots + x_1y_1 + x_0y_0 \\
q_{(m-1)+1} &= x_{m-1}y_m + \cdots + x_1y_2 + x_0y_1 \\
&\vdots
\end{aligned}$$

Thus, we can multiply the polynomials X and Y, then take the coefficients $m-1$ and bigger to get all of our cross correlation terms.

3 Graph Algorithms

3.1 Lecture 5

3.1.1 Graph Representation

Definition 3.1 (Graph)
A graph is a pair $G = (V, E)$ where V is the set of vertices and E is the set of edges. We generally call $n = |V|$ and $m = |E|$.
If G is a directed graph, then $E \subseteq V \times V$. G is simple if (a, a) (a self loop) is not allowed.
If G is an undirected graph, then E is a set of unordered pairs from V. If there are no self-loops, G is simple.

Example 3.1 (Examples of graphs)
Graphs are often used as convenient ways to represent data.

1. Road network where vertices are intersections, and edges are road segments connecting intersections. (Would be directed, also maybe "weighted" according to distance)
2. Social networks where vertices are people, and edges are friendships. (Facebook would undirected, Twitter/IG would be directed)

How do we represent graphs on a computer? We will assume: $V = \{1, \ldots, n\}$. Then to store edges, we will either:

a Adjacency Matrix: A is an n-by-n matrix where

$$A_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & (i,j) \notin E \end{cases}$$

for a weighted graph this becomes:

$$A_{ij} = \begin{cases} w_{ij} & (i,j) \in E \\ \infty & (i,j) \notin E \end{cases}$$

b Adjaceny List: represent E as an array B of linked lists. Then, $B[i]$ is a linked list containing all j such that $(i, j) \in E$.

We can compare the cost of these representations as follows:

	Adjancency Matrix	Adjancency List
Space	n^2 bits	$\Theta(m+n)$ words
$(u, v) \in E$?	$\mathcal{O}(1)$	$\Theta(1 + d_u)$
Print all the neighbors of u	$\Theta(n)$	$\Theta(d_u)$

where d_u is the degree of u, i.e. $|\{w \mid (u, w) \in E\}|$.

Note that we could also choose alternative representations. Remember that graphs are abstractions used to store data, so there is no one-size-fits-all solution.

3.1.2 Depth First Search (DFS)

We discuss graph exploration, i.e. visiting all vertices in a graph.

Algorithm 3.1 (Depth First Search)

```
function DFS(V, E)
    global clock = 1
    global visited = boolean[n]
    global preorder, postorder = int[n], int[n]
    for v ∈ V do
        if visited[v] is false then explore(v)
function EXPLORE(v)
    visited[v] = true
    preorder[v] = clock
    clock = clock + 1
    for (v, w) ∈ E do
        if visited w is false then explore(w)
    postorder[v] = clock
    clock = clock + 1
```

The time window between postorder[v] and preorder[v] is exactly the amount of time we spend in recursive calls from v.

We have a claim about the subroutine EXPLORE. Namely, EXPLORE(u) explores exactly: $\{v \mid \exists$ a path from u to $v\}$, i.e. the connected component of u.

An argument for this claim is as follows:

Proof

We need to show two directions.

First, if we explored v, there must be a path from u to v, since every call in the recursion is from one neighbor to another. We can construct the path by just following the path in the recursion tree.

In the other direction, for the sake of contradiction, suppose there exists a reachable v that doesn't get explored. Let the path from u to v be:

$$(u \to x_1 \to x_2 \to \cdots \to x_r = v)$$

Let x_j be the first vertex on the path which isn't explored. Then, look at x_{j-1}, which was explored, which means we looped over all of x_{j-1}'s neighbors. This means we had the opportunity to visit x_j, but we didn't. This means we must've visited x_j and failed the if check. This is a contradiction, we visited x_j and didn't. Thus, we must have that all reachable v's are explored.

Now we look at the runtime of the routine. First, note that the outer for loop runs n times. Then, we have to enumerate the neighbors of u for all vertices u in the graph. We only have to do this once, since we visit each vertex only once.

This means that the total running time is:

$$T(n) = \Theta(n) + \sum_{u \in V} \text{time to enumerate neighbors}$$

which depends on our graph representation. With our adjacency matrix, the second term is quadratic. In the adjacency list, it is just the sum of (degrees + 1) which will be exactly $2m + n$. (Since by adding degrees we double count the number of edges).

	Adjancency Matrix	Adjancency List
DFS time	$\Theta(n^2)$	$\Theta(m+n)$

TODO: Add a pictoral example of DFS from lecture.

There are many applications of the DFS:

- Reachability - Identifying the separate connected components
- Articulation points - The set of vertices whose removal disconnects the graph

- Finding biconnected/triconnected components - if there are two/three disjoint paths between any two vertices
- Strongly connected components - In a directed graph, two vertices are strongly connected if there is a path from one vertex to another and from that other vertex back to the original
- Planarity testing - Testing if a graph is planar, i.e. you can draw it without crossings
- Isomorphism of planar graphs - Telling whether two planar graphs are isomorphic, i.e. we can turn one into the other with a bijection

The first set of problems we have already solved. Reachability is everything we visit in one iteration of the outer loop. The amount of connected components is the amount of explore calls in the outer loop.

Definition 3.2 (Preorder, Postorder)
The notation $\text{pre}(u)$ and $\text{post}(u)$ denote the preorder and postorder number of a given vertex u in a run of the DFS algorithm on some graph containing u.

Now let us explore another claim about the DFS.

Note 3.1
For some vertex u, the set of intervals $[\text{pre}(u), \text{post}(u)]$ are either pairwise nested or disjoint. The argument is that pairwise, either two are in the same connected component or different components. In the same component, one must have started before the other, then the recursion finished the inner one, then it must've ended. In different components, the intervals are disjoint.
Let us denote previsiting a vertex u as $[_u$ and postivisiting that vertex as $]_u$.

It helps us to classify the graph edges during a DFS.

Definition 3.3 (Types of Graph Edges)
These are the types of edges in a DFS traversal of G.

1. Tree edge: Traversed edge during DFS (in the DFS tree).
2. Forward edge: Goes from an ancestor to a descendant in the DFS tree, and is not a tree edge.
3. Back edge: Goes from a descendant to an ancestor in the DFS tree, and is not a tree edge.
4. Cross edge: An edge that is not any of the above.

The useful facts about these edges are as follows:

Theorem 3.1
Suppose $(u, v) \in E$. $\text{post}(u) < \text{post}(v)$ if and only if (u, v) is a back edge.

Proof
The backwards part of the claim is simpler. Basically u is a decendant of v, so the order is something like $[_v \ldots [_u$, so $]_u \ldots]_v$ must happen, i.e. the postorder numbers have the given order.

Theorem 3.2
G has a cycle if and only if it has a back edge.
We again take the backwards case first. By the definition of a back edge, there must be a series of tree edges from some vertex u to v and then a back edge from v to u. This implies there is a cycle from u to u.

Proof
Now the forward part of the claim, consider the cycle:

$$u_0 \to u_1 to \cdots \to u_k \to u_0$$

Take u_i with the lowest postorder number. Then the edge:

$$(u_i, u_{(i+1) \mod k})$$

goes from a higher postorder number to a lower postorder number, which means that by the previous result, this edge is a back edge, showing our claim.

3.2 Lecture 6

3.2.1 Topological Sort

We consider the problem of topological sort. We take as input a directed acyclic graph (DAG). We seek to find an ordering of V such that $(u, v) \in E \implies u$ must come before v in the ordering. Note that such an ordering of the graph is not necessarily unique. Intuitively, it represents the order that dependencies must be filled in order to solve a problem.

Algorithm 3.2 ("Brute-force" Search)
Try all permutations of the vertices. Return the first one that is a topological sort.
The runtime is $\mathcal{O}n!mn$, because for each of $n!$ permutations, we have to check m edges that the edge is respected, and finding each endpoint of the edge needs to be found in the sorted order.

To find a faster algorithm, let us define some terms.

Definition 3.4 (Source, Sink)
In a directed acyclic graph, a source is a vertex with no incoming edges. A sink is a vertex with no outgoing edges. Note that every DAG has a source and a sink.

Algorithm 3.3 (Source Peel-Off)
We can iteratively "peel-off" source vertices one at a time. It is possible to do this algorithm in linear time: $\mathcal{O}(m + n)$.
To do so, have n linked list nodes, one for each vertex. Then we keep an array B where the ith entry of the array stores a pointer to node i. Then, we also keep an array B of linked lists where the ith entry of that array is a doubly-linked list that has i incoming edges. To peel off a source, you take anything in the 0th entry of the array and take it out, call it u. Then, for each outgoing edge (u, v), move v to the lower bucket in the array (since it now has one less incoming edge). Rinse and repeat.
This is a bit too complicated. There is an easier linear-time algorithm.

Consider the following pre and postorder traversals (subscripts dropped for clarity):

$$[[][]][][[]]$$

then, it's easier to see that the vertex associated with last closing bracket is a source; it has nowhere else that lead to it. This leads to the following result:

Theorem 3.3 (Finding a Source with DFS)
The vertex v with the largest postorder number must be a source.

Proof
Suppose for the sake of contradiction that v is not a source. That means there must exist a $u \in V$ such that $(u, v) \in E$.
We know that the only possibilities are that $[_u]_u[_v]_v$ or $[_v[_u]_u]_v$. The latter means that (u, v) is a back edge, which would mean there is a cycle. However there are no cycles since this is a DAG, so this cannot be the case. The former means that u and v are disconnected, but there is clearly an edge between them or that v was already visited, which is also not the case.
By exhaustion of cases, we have a contradiction, so v must be a source. ∎

Algorithm 3.4 (Topological Sort)
We begin by doing a DFS traversal on G. Then, we sort by postorder number. This exactly a topological sort

of the graph.
This works because peeling off the source causes the DFS to be the exact same, so the order of postordering numbers does not change.
Note that postorder number is bounded between 2 and $2n$, so it is easy to sort in linear time. Thus the runtime is dominated by DFS, $\mathcal{O}(n+m)$.

3.2.2 Strongly Connected Components

Next we try to find strongly connected components in a directed graph.

Definition 3.5 (Strongly Connected)
For some directed graph $G = (V, E)$, $u, v \in V$ are strongly connected if u has a parth to v and v has a path to u.

Definition 3.6 (Strongly Connected Component (SCC))
A strongly connected component in a graph G is a maximal subset of strongly connected vertices.

An easy algorithm to find SCCs would be to do n DFS's from each vertex and check strong connectivity. However, this is fairly slow, about $\mathcal{O}(mn + n^2)$.

Theorem 3.4 (SCC Graph is a DAG)
The graph made from treating the SCCs as nodes is acyclic (a DAG).

Proof
Suppose there was a cycle. Then, all the vertices in those components forming a cycle are strongly connected, which would contradict the fact that the SCCs were maximal. Thus, there cannot be a cycle.

We want to find a sink of this reduced DAG. In particular, running a DFS from the sink SCC, we can peel it off and recurse. This means we want to find the SCCs in reverse topological sorted order. Imagine we had a subroutine that could find a vertex in a sink in constant time. Then, the runtime would be: $\mathcal{O}(n+m)$ because we have to DFS over every single strongly-connected component, visiting every vertex and edge once.

To do this, first let us reverse the graph (make all the edges go the other direction). This will turn any source into a sink and vice versa. Now we have to find a source SCC in the reversed graph G_{rev}. The claim is that the highest postorder number still works.

Theorem 3.5
If u has the highest postorder number, then u is in the source SCC.

Proof
Suppose that u is not in the source SCC. That means there is another SCC that point into that SCC of u. This means that $(v, w) \in E$, where w has a path to u and u has a path to w. Since v thus has a path to u but u has the highest postorder number, the intervals cannot be disjoint. Furthermore, the intervals cannot be inside each other, as this would imply u would have a path v; but that would mean they're in the same SCC, which they're not. Thus, the edge (v, w) cannot exist, meaning that u is in the source SCC.

Finally our algorithm to find the SCCs is as follows:

Algorithm 3.5
1. Reverse the graph and run topological sort on it. This will give you the ordering S of sinks in the original

graph.

2. Run DFS on each sink s in order. The vertices reached are in SCC of s, so we can remove them from the graph (mark them) and keep going.

3. The SCCs removed are all the SCCs.

This algorithm requires a topological sort and then small DFS's which amount to a single big DFS. This is $\mathcal{O}(m+n)$.

3.3 Lecture 7

3.3.1 Single Source Shortest Paths

We now consider shortest paths. Suppose we are given a directed graph G and a start vertex s. We wish to find the shortest path from s to all other vertices. More precisely, we want two arrays:

- prev$[1, \ldots, n]$, where prev$[v]$ is the previous vertex to v on the shortest path from s to v.
- dist$[1, \ldots, n]$, where dist$[v]$ is the length of the shortest path from s to v.

Note that the prev array has enough information to give us the actual shortest path from s to v.

There are many algorithms for single-source shortest paths. Here are a few:

- Breadth-First Search - Assumes edges all have weight 1 ("unweighted")
- Djikstra's Algorithm - Assumes edge weights are all non-negative.
- Bellman-Ford Algorithm - Arbitrary edge weights.
- Dynamic Programming on DAGs - Arbitrary edge weights, but G must be a DAG.

Note that depth-first search does not work because going deep first may yield a longer path than some other traversal. Consider the graph:

$$V = \{S, A, B\}, E = \{(S, A), (A, B), (S, B)\}$$

Then suppose the DFS traversal was S, A, B. Then the path to B would seem like it's distance 2, when in reality it's 1, since the (S,B) wouldn't be traversed by DFS.

Algorithm 3.6 (Breadth-First Search)
Here is the main algorithm:

```
function BFS(G, s)
    dist[1...n] ← ∞
    prev[1...n] ← null
    vis[1...n] ← False
    Q ← queue(s)
    dist[s] ← 0
    vis[s] ← True
    Q.push(s)
    while Q.size> 0 do
        u ← Q.pop()
        for (u, v) ∈ E do
            if !vis[v] then
                vis[v] ← True
                dist[v] ← dist[u] + 1
                prev[v] ← u
                Q.push(v)
    return dist, prev
```

Runtime Analysis

Proof
The first 3 operations are linear in n, and the last 4 are constant time. Then note that every vertex is added to

the Q once, and all of its edges are looped over. So the total runtime is:

$$T(n) \leq \Theta(n) + \sum_{v \in V} C \cdot (1 + \text{outdegree}(v))$$

Note that the sum of the outdegrees counts every edge once. Thus, $T(n) = \mathcal{O}(m + n)$.

The interesting thing about BFS is that it is implemented the exact same as iterative DFS, but the stack is replaced with a queue.

To show correctness, we require a few intermediate results:

Theorem 3.6 (BFS Lemma 1)
$\forall v \in V$, $\text{dist}[v] \leq \delta(s, v)$ (the true shortest distance).

Proof
We proceed by induction on k, the number of push operations to Q so far. $k = 1$ is trivial, since the dist of s is 0 and everyone else's distance is infinity, satisfying the inequality. Consider the $k + 1$st push (with the induction holding for $\leq k$), during the edge (u, v). We push v when we visit u. By the inductive hypothesis, $\text{dist}[u] \geq \delta(s, u)$, so

$$\text{dist}[v] = 1 + \text{dist}(u) \leq \delta(s, u) + 1 = \delta(s, u) + \delta(u, v) \geq \delta(s, v)$$

completing the induction.

Theorem 3.7 (BFS Lemma 2)
Look at any point in time i. Say $Q = [v_1, \ldots, v_r]$. Then

1. $\forall i, \text{dist}[v_i] \leq \text{dist}[v_{i+1}]$
2. $\text{dist}[v_r] \leq \text{dist}[v_1] + 1$

Proof
The proof is similar to the theorem above. Induct on the number of queue operations.

Now we can show the correctness of BFS.

Theorem 3.8 (The Correctness of Breadth-First Search)
We will show that BFS finds shortest paths from s.

Proof
For the sake of contradiction, suppose the dist arry is incorrect. Then, there is some $v \in V$ such that $\text{dist}[v] \neq \delta(s, v)$. By BFS Lemma 1, we must have $\text{dist}[v] > \delta(s, v)$. This may be the case for many such v. Let us pick the v such that $\delta(s, v)$ is minimum (note that $v \neq s$). Then, let us take the shortest path from s to v.

$$s \to v_1 \ldots v_{r-1} \ldots v$$

This means for all intermediate vertices until v_{r-1}, the dist array was set correctly. Look at the point in time when v_{r-1} was popped off Q. But this means that v was not put into the Q (since the distance was resolved incorrectly) at this point. This means that v was visited already by some other vertex u and by BFS Lemma 2, this means that $\text{dist}[u] \leq \text{dist}[v_{r-1}]$. But this would mean that v got set to some value:

$$\text{dist}[v] = \text{dist}[u] + 1 \leq \text{dist}[v_{r-1}] + 1 = \delta(s, v)$$

However, we initially claimed $\text{dist}[v] > \delta(s,v)$, so this is a contradiction! So we cannot have any place where the dist array is incorrect.

3.3.2 Weighted Graphs

Note that if all weights $w(e) \in \mathbb{N}$, then we can reduce finding the SSSP to the unweighted case by just subdividing edges into $w(e)$ fake vertices in the middle. If $w(e) \leq L$, then the BFS runtime is $\mathcal{O}(n + mL)$. We can do better (and also use $w(e) \in \mathbb{R}$).

To do this, we use heaps (or priority queues). A min-heap is a data structure that maintains a set of (key, value) pairs S subject to the following three operations:

1. delMin(), returns (k, v) from S with the smallest k and removes it from S.
2. decKey(O, k'), where O is a pointer to the v object, replaces the old key with a smaller key k'
3. insert(k, v), inserts (k, v) into S

We can now use these heaps to solve SSSPs for arbitrary non-negative weighted graphs.

Algorithm 3.7 (Djikstra's Algorithm)

```
function DJIKSTRA(G, s)
    dist[1...n] ← ∞
    prev[1...n] ← null
    H ← heap()
    for v ∈ V do
        H.insert(∞, v)
    dist[s] ← 0
    H.decKey(s, 0)
    while H.size> 0 do
        u ← H.delMin()
        for (u, v) ∈ E do
            if dist[u] + w((u, v)) < dist[v] then
                H.decKay(v, dist[u] + w(u, v))
                dist[v] ← dist[u] + w((u, v))
                prev[v] ← u
    return dist, prev
```

Runtime Analysis We do n insertions and thus must do n deleteMins, and we are doing potentially $\deg(v)$ insertions for each, so overall, considering t_I as the runtime of insert, t_{dK} for decKey and t_{dM} for delMin, we have the runtime is something close to: $T(n) = \mathcal{O}(n + m + nt_I + nt_{dM} + mt_{dK})$ which is:

$$T(n) = \mathcal{O}((m + n) \log n)$$

for a binary heap implementation.