Discrete Math and Algorithms: ICME Refresher Course

2021

These notes have been adapted from Julia Olivieri's 2018 Refresher course notes and Aaron Sidford's CME 305 course notes. Thank you to both of them!

Hello!

Hi! I'm glad you're all here and am excited for you to start your classes! As things get tough this year (I hope they don't but they probably will) please remember that we've all been though the same course sequence and have had similar struggles– please, please let any one of us know how we can help you, even if it's just to lend a listening ear!

Overview

In this refresher course we'll go through some things that will hopefully help you through all of your core classes, but especially in CME 305 in the Winter. We'll have three sessions together and will go over **Asymptotics and Big-O Notation**, **Recurrence Relations**, **Graph Theory and Algorithms**, **Combinatorics**, and **Complexity Theory** all in varying levels of depth and detail.

Asymptotics and Big-O Notation

Big-O notation lets us talk about the asymptotic runtime of algorithms as a function of their input without worrying about specific details of implementation. Big-O notation expresses an asymptotic upper bound of a function so we can talk about and compare different algorithms at a higher level.

Big-O notation expresses an asymptotic upper bound on a function. We say that a function f is $\mathcal{O}(g)$ if the rate of growth of f is less than or equal to the rate of growth of g, up to a constant.

Definition 1. ***Big-O*** *Let $f, g : \mathbb{R} \to \mathbb{R}$. We say that f is $\mathcal{O}(g)$ if there are positive constants c and n_0 such that for $n > n_0$, $|f(n)| \leq c|g(n)|$. Sometimes this is written as $f(x) = \mathcal{O}(g)$, and other times it's written as $f \sim \mathcal{O}(g)$.*

If it's helpful to think of it in words, f is "big-O-of" g if after some value $x = N$, $f(x)$ is always less than or equal to some constant times $g(x)$.

Big-O notation is commonly used to understand the running time of an algorithm based on the size of its input.

Example 1. *The number of multiplications and additions needed to multiply two $n \times n$ matrices together is $n^2(2n-1)$ which is $\mathcal{O}(n^3)$.*

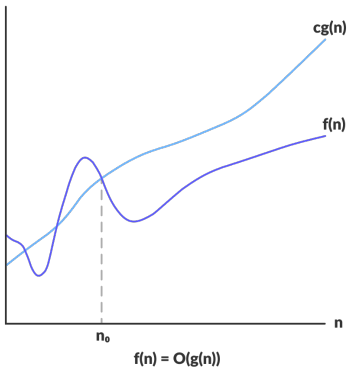


Figure 1: In this figure we see that after

To see this, let $f(n) = n^2(2n-1)$ and $g(n) = n^3$. For $n_0 = 1$ and $c = 2$, then for $n > n_0$, $f(n) = 2n^3 - n^2 < 2n^3 = cg(n)$.

Example 2. *Let $a_0, a_1, \ldots, a_k$ be real numbers. Then $f(x) = a_0 + a_1 x + \cdots + a_k x^k$ is $\mathcal{O}(x^k)$.*

To see this, choose $n_0 = 1$ and let $c = |a_0| + |a_1| + \cdots + |a_k|$. Let $g(x) = x^k$. Then for $x > n_0$, we have

$$\begin{aligned} |f(x)| &= |a_0 + a_1 x + \cdots + a_k x^k| \\ &\leq |a_0| + |a_1| x + \cdots + |a_k| x^k \\ &\leq |a_0| x^k + |a_1| x^k + \cdots + |a_k| x^k \\ &= (|a_0| + |a_1| + \cdots + |a_k|) x^k = c|g(x)|. \end{aligned}$$

Exercise 1. *Show the following:*

- *If f_1 and f_2 are both $\mathcal{O}(g)$, then so are $\max(f_1, f_2)$ and $f_1 + f_2$.*
- *If f_1 is $\mathcal{O}(g_1)$ and f_2 is $\mathcal{O}(g_2)$, then $f_1 f_2$ is $\mathcal{O}(g_1 g_2)$.*
- *$\log(n!)$ is $\mathcal{O}(n \log n)$.*

Theorem 1. *For any k, $f(x) = x^k$ is $\mathcal{O}(2^x)$.*

Proof. The Taylor Series of 2^x centered at $x = 0$ is,

$$2^x = 1 + (\ln 2)x + \frac{\ln^2 2}{2}x^2 + \cdots + \frac{\ln^k 2}{k!}x^k + \frac{\ln^{(k+1)} 2}{(k+1)!}x^{(k+1)} + \cdots$$

Choose $c = +\frac{k!}{\ln^k 2}$ and $n_0 = 1$. Then for $x > n_0$, we know that

$$\begin{aligned} |f(x)| = x^k &< \frac{k!}{\ln^k 2} + \frac{k!}{\ln^k 2}(\ln 2)x + \frac{k!}{\ln^k 2}\frac{\ln^2 2}{2}x^2 + \cdots + x^k + \cdots \\ &= \frac{k!}{\ln^k 2}(2^x) = c|g(x)|. \end{aligned}$$

□

Big-O notation lets us compare functions in an asymptotic sense by describing *upper bounds*. However, we may want to discuss functions through lower bounds, equivalency, or tight bounds. We have this notation!

Definition 2. *Let $f, g : \mathbb{R} \to \mathbb{R}$.*

- **Ω *(big-Omega):*** *We say that $f(x)$ is $\Omega(g(x))$ if there exist positive constants c and n_0 such that for $x > n_0$, $|f(x)| \geq c|g(x)|$. This indicates that f is asymptotically lower bounded by g. In other words, f grows at least as fast as g, up to a constant.*

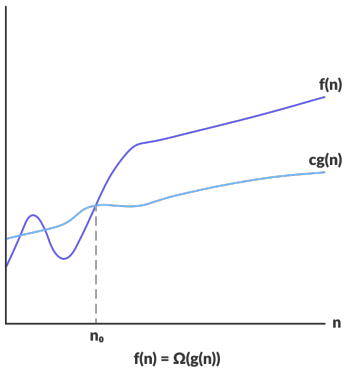


Figure 2: Big-Omega notation gives us a *lower bound* for a function f.

Example 3. $\cos(x) + 2$ *is* $\Omega(1)$.

- Θ ***(big-Theta):*** *We say that* $f(x)$ *is* $\Theta(g(x))$ *if there exists positive constants* c_1, c_2, *and* n_0 *such that for* $x > n_0$, $c_1 g(x) \leq f(x) \leq c_2 g(x)$. *Equivalently,* $f(x)$ *is both* $\mathcal{O}(g(x))$ *and* $\Omega(g(x))$. *This indicates that* f *and* g *have the same rate of growth (up to a constant).*

  Example 4. $n^2(2n-1)$ *is* $\Theta(n^3)$.

- o ***(little-o):*** *We say that* $f(x)$ *is* $o(g(x))$ *if for* ***every*** *positive constant* c, *there is a positive constant* n_0 *such that for* $x > n_0$, $|f(x)| \leq c|g(x)|$. *This more or less indicates that* g *is a strict upper bound for* f.

  Example 5. $n \log n$ *is* $o(n^2)$ *and for any* $k > 0$, $\log n$ *is* $o(n^k)$.

- ω ***(little-omega):*** *We say that* $f(x)$ *is* $\omega(g(x))$ *if for* ***every*** *positive constant* c *there is a positive constant* n_0 *such that for* $x > n_0$, $|f(x)| \geq c|g(x)|$. *This more or less indicates that* g *is a strict lower bound for* f.

  Example 6. n^2 *is* $\omega(n \log n)$.

Theorem 2. $\log(n!)$ *is* $\Theta(n \log n)$.

Proof. Above, we showed that $\log(n!)$ is $\mathcal{O}(n \log n)$ so it is sufficient to show that $\log(n!)$ is $\Omega(n \log n)$. Note that the first $n/2$ terms in $n! = n \cdot (n-1) \cdot \cdots \cdot 1$ are all greater than $n/2$. Thus, $n! \geq (n/2)^{(n/2)}$. Therefore,

$$\begin{aligned} \log(n!) &\geq \log((n/2)^{(n/2)}) = (n/2)\log(n/2) \\ &= (n/2)(\log n - \log 2) = \Omega(n \log n). \end{aligned}$$

□

Theorem 3. *Let* $a_0, a_1, \ldots, a_k$ *be real numbers with* $a_k > 0$. *Then* $f(x) = a_0 + a_1 x + \cdots + a_k x^k$ *is* $\Theta(x^k)$.

Proof. From the example above, we know that $f(x)$ is $\mathcal{O}(x^k)$, so it is sufficient to show that $f(x)$ is $\Omega(x^k)$. Let $d = \max_{1 \leq i \leq (k-1)} |a_i|$. Then,

$$\begin{aligned} |f(x)| &= |a_0 + a_1 x + \cdots + a_k x^k| \\ &\geq a_k x^k - |a_{k-1}|x^{k-1} - |a_{k-2}|x^{k-2} - \cdots - |a_0| \\ &\geq a_k x^k - d x^{k-1} \text{ for } x > 1. \end{aligned}$$

Let $c = a_k/2$. Then if we want $|f(x)| \geq a_k x^k - d x^{k-1} \geq c|x^k| = (a_k/2)x^k$ for $x > 1$, we must have that $(a/2)x^k - d x^{k-1} \geq 0$. This holds when $x > 2d/a_k$, so $|f(x)| \geq c|x^k|$ for all $x > \max(1, 2d/a_k)$. Therefore we have both that $f(x)$ is both $\mathcal{O}(x^k)$ and $\Omega(x^k)$, so $f(x)$ is $\Theta(x^k)$. □

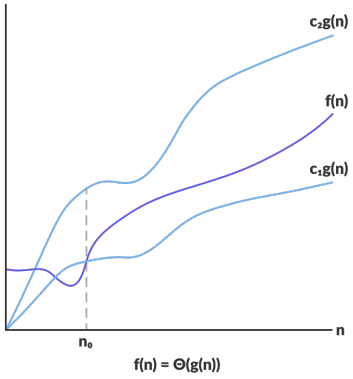


Figure 3: Big-Theta notation gives a tight bound for a function f. Here you can see how after some constant n_0, the function $f(n)$ is always sandwiched between the functions $c_1 g(n)$ and $c_2 g(n)$.
Little-o notation can be a little confusing to understand. Another way to think of it is as an "asymptotically loose upper bound." In addition to the mathematical definition given here, if $f(x) = o(g(x))$, this means that $\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0$. A helpful example: $n^2 = o(n^3)$.

Theorem 4. $\binom{n}{k}$ *is* $\Theta(n^k)$ *for fixed constant* $k \leq n/2$.

Proof. Note that

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k!},$$

which is a polynomial with leading term n^k. Therefore, $\binom{n}{k}$ is $\Theta(n^k)$ by the above Theorem. Note here that if $k > n/2$, then the terms in the denominator begin to cancel out with the terms in the numerator. For *any* constant k, then $\binom{n}{k}$ is $\Theta(n^\ell)$ for $\ell = \min(k, n-k)$. □

Recurrence Relations

In this section we will learn about recurrence relations and how they can help us understand algorithmic complexity. You'll use this in a lot of your classes.

Definitions and examples

Definition 3. *A recurrence relation for a function $T(n)$ is an equation for $T(n)$ in terms of $T(0), T(1), \ldots, T(n-1)$. That is, a recurrence relation for a function is a recursive definition based on previous values that requires konwledge of some baseline function values to compute.*

Example 7. *The Fibonacci sequence:*

- $f_0 = 0, f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}, n \geq 2.$

Recurrence relations can be the easiest way to describe some functions, and it can be useful to know a function's recurrence relation! We can solve recurrence relations to get explicit formulae for functions and we can use recurrence relations to analyze the complexity of algorithms.

Theorem 5. *The explicit form of the recurrence relation $T(n) = aT(n-1) + bT(n-2)$ is*

$$T(n) = cr^n + ds^n,$$

*where c and d are constants and r and s are the **distinct roots** of the equation $x^2 - ax - b = 0$. Constants c and d can be found from the base case of the recurrence relation.*

Exercise 2. *Find the explicit form of the Fibonacci sequence.*

Relating the Fibonacci sequence to the above theorem, we know that $a = b = 1$. To find r and s we consider the roots of the equation

$$x^2 - x - 1 = 0$$

which gives $r = \frac{1+\sqrt{5}}{2}$ and $s = \frac{1-\sqrt{5}}{2}$. Plugging this into the result from the above theorem gives,

$$f_n = c\left[\left(1+\sqrt{5}\right)/2\right]^n + d\left[\left(1-\sqrt{5}\right)/2\right]^n.$$

Plugging in $f_0 = 0$ and $f_1 = 1$ gives $c = 1/\sqrt{5}$ and $d = -1/\sqrt{5}$. Therefore the explicit form of the Fibonacci sequence is,

$$f_n = \frac{1}{\sqrt{5}}\left[\left(1+\sqrt{5}\right)/2\right]^n - \frac{1}{\sqrt{5}}\left[\left(1-\sqrt{5}\right)/2\right]^n.$$

In the above theorem the equation $x^2 - ax - b = 0$ had two distinct roots. What if there is only one, repeated root?

Theorem 6. *The explicit form of the recurrence relation* $T(n) = aT(n-1) + bT(n-2)$ *is*

$$T(n) = cr^n + dnr^n,$$

where c and d are constants and r is the ***repeated root*** *of the equation* $x^2 - ax - b = 0$.

Example 8. *Consider the following game. In the first two steps of the game you are given numbers* z_0 *and* z_1*. At each subsequent step of the game you flip a fair coin. If the coin is heads your new score is four times your score from the previous step. If the coin is tails, your new score is the negation of two times your score from two steps ago. What is your expected score at the* n*th step of the game?*

Let X_n be the score at the nth step of the game. Then the recurrence relation is,

$$X_n = 4X_{n-1}\mathbb{I}(n\text{th coin toss is heads}) - 2X_{n-2}\mathbb{I}(n\text{th coin toss is tails}).$$

Then finding the expected score at the nth step of the game is,

$$\mathbb{E}[X_n] = 4\mathbb{E}[X_{n-1}]\mathbb{P}[\text{heads}] - 2\mathbb{E}[X_{n-2}]\mathbb{P}[\text{tails}] = 2\mathbb{E}[X_{n-1}] - \mathbb{E}[X_{n-2}].$$

This is a recurrence relation! To see this, define $T(n) = \mathbb{E}[X_n]$. By recognizing this, we can use one of the two theorems above by finding the roots of the equation $x^2 - 2x + 1 = 0$. This equation has only one repeated root at $r = 1$, so the explicit form of the recurrence relation is

$$T(n) = c1^n + dn1^n = c + dn.$$

Plugging in $T(0) = z_0$ and $T(1) = z_1$, we find $c = z_0$ and $d = z_1 - z_0$. Therefore, the expected score at the nth step is

$$\mathbb{E}[X_n] = T(n) = z_0 + (z_1 - z_0)n.$$

The master theorem and examples

Above, we saw how we could use two theorems to solve recurrence relations for their explicit form. In this section we show how recurrence relations can help us understand the asymptotic complexity of algorithms. The main theory below is frequently used to understand algorithmic complexity.

Theorem 7. ***The Master Theorem.*** *Suppose that* $T(n) = aT(\lceil n/b \rceil) +$

$\mathcal{O}(n^d)$ *for constants* $a > 0, b > 1, d \geq 0$. *Then,*

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

The master theorem is applicable to a wide range of *divide and conquer* algorithms. A *divide and conquer* algorithm is an algorithm that divides a problem into a number of subproblems recursively where each subproblem is of size n/b.

Example 9. ***Fast Matrix Multiplication.*** *Consider the product* $\boldsymbol{C}$ *of two* $n \times n$ *matrices,* $\boldsymbol{A}$ *and* $\boldsymbol{B}$ *(for* $n = 2^k$*).*

If we partition the matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ into four equally sized blocks, we see that,

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_2 & M_3 + M_4 \\ M_5 + M_6 & M_7 + M_8 \end{bmatrix}.$$

Where

$$\begin{aligned} M_1 &= A_{11}B_{11} \quad M_2 = A_{12}B_{21} \quad M_3 = A_{11}B_{12} \quad M_4 = A_{12}B_{22} \\ M_5 &= A_{21}B_{11} \quad M_6 = A_{22}B_{21} \quad M_7 = A_{21}B_{12} \quad M_8 = A_{22}B_{22}. \end{aligned}$$

We see that the multiplication of two $n \times n$ matrices has been subdivided into the multplication of eight $(n/2) \times (n/2)$ matrices and the addition of four $(n/2) \times (n/2)$ matrices. This is the exact scenario described above wherein we can use the Master theorem to find the algorithmic complexity!

Let $T(n)$ be the number of floating point operations used to multiply two $n \times n$ matrices with this algorithm. There are two costs: eight multiplications and four additions. Since adding two $n \times n$ matrices uses $\mathcal{O}(n^2)$ operations, the cost of the algorithm is

$$T(n) = 8T(n/2) + \mathcal{O}(n^2).$$

Applying the Master theorem with $a = 8$, $b = 2$, and $d = 2$, we get that

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

Example 10. ***k-select.*** *Consider the k-select algorithm which finds the kth smallest number in a list of numbers L. What is the asymptotic running time of this algorithm?*

Algorithm: k**-select**, `select(L,k)`
Input: List L of numbers and integer k
Output: the kth smallest number in L, denoted x_*.
`pivot` $\leftarrow$ uniform random element of L
$L_< \leftarrow \{x \in L \mid x <$ `pivot`$\}$
$L_> \leftarrow \{x \in L \mid x >$ `pivot`$\}$
$L_= \leftarrow \{x \in L \mid x =$ `pivot`$\}$
if $k \leq |L_<|$ **then**
    $x_* \leftarrow$ `select(` $L_<$`,k)`
else if $k > |L_<| + |L_=|$ **then**
    $x_* \leftarrow$`select(`$L_>$`,`$k - |L_<| - |L_=|$`)`
else
    $x_* \leftarrow$`pivot`
    return x_*

The lists $L_<, L_>$, and $L_=$ can be formed in $\Theta(n)$ time, where n is the number of elements in the list L. To consider the worst case running time, if the pivot element is *always* chosen to be the largest element in the array and if $k = 1$, then the algorithm would take $n + (n-1) + \cdots + 1 = \Theta(n^2)$ time.

To consider the *average-case* running time first consider how long it takes on average for the current list to have $3n/4$ or fewer elements. This happens if the pivot we choose has at least $1/4$ elements of L smaller and at least $1/4$ of the elements larger. That is, the pivot has to be in the middle half of the elements of L. Since we choose the pivot uniformly at random, this condition holds with probability $1/2$ in the *first* call to the k-select algorithm. The expected number of steps until choosing a pivot in the middle half of the data is two. Then the expected running time of the algorithm, $T(n)$ satisfies

$$T(n) \leq T(3n/4) + \mathcal{O}(n)$$

where here the $\mathcal{O}(n)$ term contains the expected number of steps (in this case, two) to find a pivot that partitions the data so that the algorithm operates on at most $3/4$ the size of the original list. Using the master theorem, the expected running time is $\mathcal{O}(n)$. Since we have to read all n elements of L, the expected running time is $\Theta(n)$.

Graph Theory and Algorithms

In this section we go over basic graph definitions and some algorithms on graphs. Graphs will be a main topic in 305, so it's great to get a head start on some definitions and topics now!

Graph definitions

Graphs are a common abstraction to represent relational data, and examples of graphs are all around us: road networks, protein interaction networks, friendship networks, food chains, etc. Let's begin with a definition of a graph.

Definition 4. *A graph $G = (V, E)$ is defined by a set of vertices, V and a set of edges, E. Each edge $e \in E$ is associated with two vertices $u, v \in V$ and we write $e = (u, v)$. We say that u is **adjacent to** v, u is **incident to** v, and u is a **neighbor of** v.*

With this definition, we can relate the vocabulary of graphs to common graphs we interact with. In the road network, vertices are cities which are connected with an edge if there is a shared highway between them. In the protein interaction network, vertices are proteins and the edges represent interactions between them. In the friendship network, vertices are people and an edge connects two people if they are friends. In the food chain, vertices are animals and an edge represents that one of the animals eats the other. This example highlights the need for *directed graphs*.

Definition 5. *A directed graph $G = (V, E)$ is defined by a set of vertices, V and a set of directed edges, E. Each edge $e \in E$ is an **ordered** pair of vertices in V and if there is an edge from u to v, we write $e = (u, v)$. We say that u **points to** v.*

In an undirected graph, an edge (u, v) implies an edge (v, u) as well. In a directed graph, however, an edge between vertices u and v does *not* mean that there is an edge between vertices v and u. There are a *lot* of instances wherein this distinction is important.

Consider a social network wherein two people (vertices) are connected by an edge if they've called one another in the past week. If we want to include more information in the graph by denoting each edge with the *number* of phonecalls between two people in the past week, we would want to use a *weighted graph*.

Definition 6. *A weighted graph $G = (V, E, w)$ is a graph (directed or undirected) (V, E) with an associated weight function $w : E \to \mathbb{R}$. In other words, each edge e has an associated weight $w(e)$.*

There are lots of attributes of a graph that we might want to know about. The following definitions are examples of such attributes.

Definition 7. *In an undirected graph $G = (V, E)$, the **degree** of a node*

$u \in V$ is the number of edges $e = (u, v) \in E$. We write d_u to denote the degree of node u.

Lemma 1. ***The handshaking lemma.*** *In an undirected graph $G = (V, E)$, there are an even number of nodes with odd degree.*

Proof. Consider graph $G = (V, E)$ and let d_v be the degree of node v. Then

$$2|E| = \sum_{v \in V} d_v = \sum_{\{v \in V | d_v \text{ is even}\}} d_v + \sum_{\{v \in V | d_v \text{ is odd}\}} d_v.$$

The terms $2|E|$ and $\sum_{\{v \in V | d_v \text{ is even}\}} d_v$ are both even, so that means that $\sum_{\{v \in V | d_v \text{ is odd}\}} d_v$ must also be even. □

Definition 8. *In a directed graph $G = (V, E)$, the **in-degree** of a node $u \in V$ is the number of edges that point **to** u, i.e., the number of edges $(v, u) \in E$. The **out-degree** of u is the number of edges that point **away** from u, i.e., the number of edges $(u, v) \in E$.*

Definition 9. *A **path** on a graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that the edge $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, k$.*

Definition 10. *A **cycle** on graph $G = (V, E)$ is a path $v_1, v_2, \ldots, v_k$ with $k \geq 3$ such that $v_1 = v_k$.*

The social network where the vertices are people in this refresher course and an edge means two people are in the same room is an example of a complete graph!

Definition 11. *A **complete graph** $G = (V, E)$ is a graph where for every $u, v \in V, u \neq v$, the edge $(u, v) \in E$.*

Definition 12. *The **complement** of a graph $G = (V, E)$ is the graph $G' = (V, E')$ where $E' = \{(u, v) \mid (u, v) \notin E\}$.*

Storing a Graph

There are two common ways to store or represent a graph mathematically or on a computer, which we will discuss in this section.

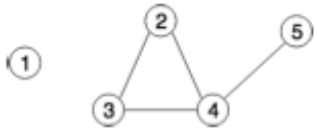
Definition 13. *The **adjacency matrix**, denoted $\boldsymbol{A}$, of a graph $G = (V, E)$ is a matrix of size $|V| \times |V|$ where*

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

The main advantage of an adjacency matrix is that we can query the existence of an edge in $\Theta(1)$ time: we just have to look at the corresponding entry in the matrix. The main disadvantage is that the storage is $\Theta(|V|^2)$. This much storage doesn't make much sense for most graphs– in many cases, there will be *much less* than $\mathcal{O}(|V|^2)$ edges. If storage is more important than access in a given context, we would rather use an adjacency list.

Definition 14. *The **adjacency list** of a graph $G = (V, E)$ is a length-$|V|$ array of linked lists. The ith linked list contains all nodes adjacent to node i.*

In an adjacency list, we can access the neighbour list of i in $\Theta(1)$ time, but querying for an edge takes $\mathcal{O}(d_{\max}$ time, where $d_{\max} = \max_j\{d_j\}$.



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

1	
2	3,4
3	2,4
4	2,3,4
5	4

Figure 4: An example graph with its corresponding adjacency matrix and adjacency list.

Graph Exploration

In this section we describe two different approaches to exploring a graph: depth-first search (DFS) and breadth-first search (BFS).

Definition 15. *We say that a node v is **reachable** from a node u if there is a path from u to v.*

The goal of both BFS and DFS is to find all nodes reachable from a given node, or simply to explore all nodes in a graph. In the below sections we describe the algorithms for undirected graphs, although they generalize to directed graphs.

Depth-first search In depth-first search we explore a graph by starting at a vertex and following a path of unexplored vertices away from this vertex as far as possible.

An example of DFS is exploring your family tree only one lineage at a time. Following the path from your maternal-grandmother directly to you (grandmother, mom, you) before considering her path to your brother (grandmother, mom, brother), or to your cousin (grandmother, uncle, cousin), uses DFS to explore a graph.

```
DFS(G, s)
Input: Graph G = (V, E), starting node s
Output: Boolean vector C of covered nodes where |C| = |V| and
  C[u] =True iff u is reachable from s.
C ← list of length |V| initialized to False
C[s] =True
for (s, u) ∈ E
    if C[u] =False
        DFS(G, u)
```

Breadth-first search In breadth-first search we explore a graph by looking at neighbours, then neighbours of neighbours, and so on. In the algorithm below, we use BFS to return the minimum-length path between each node in the graph.

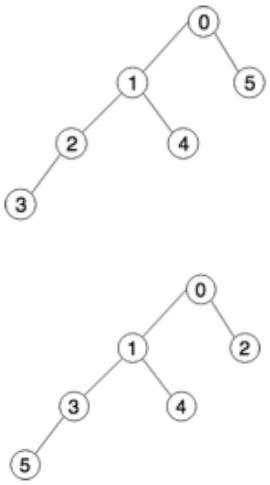


Figure 5: The order of node-exploration using DFS (above) vs. BFS (below).

An example of BFS is exploring your family tree one generation at a time. Who were *all* of your great-grandfather's children and their spouses? Who were all of their children and those spouses? Who are all your siblings and cousins?

More graph attributes

Sorry for the dump of definitions! The following define some common vocabulary in the world-of-graphs:

Definition 16. *An **induced subgraph** of $G = (V, E)$ specified by $V' \subset V$, is the graph $G' = (V', E')$, where $E' = \{(u, v) \in E \mid u, v \in V'\}$.*

Input: Graph $G = (V, E)$, source node s
Output: List `dist` where $|\texttt{dist}| = |V|$ and `dist[t]` $= \ell_{s \to t}$ if t is reachable from s and `dist[t]` $= \infty$ otherwise. Here, $\ell_{s \to t}$ is the length of the shortest path from s to t.

```
for v ∈ V do
    dist[v] ← ∞
dist[s] ← 0
Queue ← empty queue
Queue.enqueue(s)
while Queue is not empty do
    u ← Queue.dequeue()
    for (u, v) ∈ E do
      if dist[v]=∞ then
          Queue.enqueue(v)
          dist[v]←dist[u]+1
```

Definition 17. *In an undirected graph G, a **connected component** is a maximal induced subgraph $G' = (V', E')$ such that for all nodes $u, v \in V'$, there is a path from u to v in G'.*

Here, *maximal* means that there is no V'' such that $V' \subset V''$ and the induced subgraph specified by V'' is also a connected component.

Definition 18. *In a directed graph $G = (V, E)$, a **strongly connected component** is a maximal induced subgraph $G' = (V', E')$ such that for all nodes $u, v \in V'$, there is both a path from u to v and from v to u in G'.*

Definition 19. *An undirected graph is **connected** if G is a connected component. A directed graph is **strongly connected** if G is a strongly connected component.*

In a connected graph, *every* vertex v is reachable from *every* vertex u.

Trees

Trees are a special kind of graph which we are all familiar with!

Definition 20. *A **tree** is a connected, undirected graph with no cycles.*

Definition 21. *A **rooted tree** is a tree in which one node is called the root and edges are directed away from the root.*

An undirected tree can be transformed into a rooted tree by specifying any vertex in the tree as the root. This is true because there are no cycles! Imagine drawing your family tree rooted at your maternal uncle instead of rooted at your great-grandfather.

Definition 22. *In a rooted tree, for each directed edge (u, v), the vertex u is the **parent** of v and the vertex v is the **child** of u.*

Definition 23. *In a rooted tree, a **leaf** is a node with no children.*

Theorem 8. *A tree with n vertices has $n - 1$ edges.*

Proof. We prove by induction. When there are $n = 1$ vertices, there are no edges ($n - 1 = 0$). Assume that for a tree with $k - 1$ vertices, there are $k - 2$ edges. Now consider a graph with k vertices. Let v be a leaf node and consider removing v and (u, v) from the tree where u is the parent of v. The remaining graph is a tree with $k - 1$ vertices and one fewer edge. By the inductive step, it has $k - 2$ edges, so the original tree had $k - 1$ edges. □

Definition 24. *A **spanning tree** of an undirected graph $G = (V, E)$ is a tree $T = (V, E')$ where $E' \subset E$.*

Definition 25. *The **minimum spanning tree** $T = (V, E', w)$ of a weighted, connected, undirected graph $G = (V, E, w)$ is a spanning tree where the sum of the edge weights of E' is minimal.*

Two algorithms for finding minimum spanning trees are Prim's algorithm and Kruskal's algorithm. We will go over Kruskal's algorithm and its proof of correctness. You will do *lots* of "proofs of correctness" in 305, so it's nice to get a head start now!

`Kruskal(G)`
Input: Connected, weighted, undirecte graph $G = (V, E, w)$
Output: Minimum spanning tree $T = (V, E')$
$E' \leftarrow \emptyset$
`edges` $\leftarrow \{(u, v) \in E \mid w($`edges[i]`$) \leq w($`edges[j]`$)$ for $i < j\}$
for each e in `edges` **do**
  if adding e to E' does not create a cycle **then**
    $E' \leftarrow E' \cup \{e\}$
$T \leftarrow (V, E')$

Here `edges` is a list of all the edges in the graph, ordered by increasing $w(e)$. So the first edge in the list is the edge with minimum weight and the last edge in the list is the edge with maximal weight.

The running time of Kruskal's algorithm is dominated by the step where we sort the edges by increasing weight. Therefore the running time of Kruskal's algorithm is $\mathcal{O}(|E| \log(|E|))$.

Theorem 9. *When all the weights of G are distinct, the output T from Kruskal's algorithm is a minimum weight spanning tree for the connected graph G.*

Proof. **Is T a spanning tree of G?** We note that by construction T has no cycles and is thus a tree. T is spanning because each edge in the graph G is considered and added if adding it does not create a cycle. Because G is connected, this means the algorithm must have considered and added one edge connecting each otherwise disconnected component in G.

Is T a *minimum-weight* spanning tree of G? Assume that $T = (V, E')$ **is not** a minimum-weight spanning tree of G. Let $T' = (V, E'')$ be a minimum-weight spanning tree of G.

So $\sum_{e \in E''} w(e) < \sum_{e \in E'} w(e)$

Let e be the first edge added to T during Kruskal's algorithm that is **not** in T'. Then we know that $T' \cup \{e\}$ contains a cycle. Therefore, one of the edges in this cycle (we'll call it edge f) must not be in T (because T is a tree so can't have any cycles). Then $T'' = (T' \cup \{e\}) \backslash f$ is also a spanning tree.

Because T' is a minimum spanning tree, and because all the weights of G are distinct, it must be that $w(e) > w(f)$. However, it also must be that $w(e) < w(f)$ because otherwise edge f would have been added before e in Kruskal's algorithm. Thus we have a contradiction and conclude that T is a minimum spanning tree. □

Cycles

Definition 26. *A **Hamiltonian cycle** of a graph G is a cycle that visits each node of G exactly once.*

Definition 27. *A **Hamiltonian path** of a graph G is a path that visits each node of G exactly once.*

Definition 28. *An **Eulerian cycle** of a graph G is a cycle that visits each edge of G exactly once.*

Definition 29. *An **Eulerian path** of a graph G is a path that visits each edge of G exactly once.*

Cuts

Definition 30. *A **cut** $C = (A, B)$ of a graph $G = (V, E)$ is a partition of V into two subsets A and B. The **cut-set** of C is the set of edges that cross the cut, $\{(u, v) \in C \mid u \in A, v \in B\}$.*

For super cool reasons you'll see in 305, we usually care most about the edges that are being cut. In a weighted graphs, we care about the sum of the weights of the edges being cut, and in unweighted graphs we care about the number of edges being cut.

Definition 31. *An $s - t$ **minimum cut** is the cut $C^* = (S, T)$ with $s \in S$ and $t \in T$such that the cut-set is of minimum weight.*

In this definition we assume that in an unweighted graph each edge $e = (u, v) \in E$ has $w(e) = 1$.

Definition 32. *A **global minimum cut** is a cut $C^* = (A, B)$ in the graph G such that both A and B are non-empty and the cut-set has minimum weight.*

Definition 33. *A **global maximum cut** is a cut $C^* = (A, B)$ in the graph G such that both A and B are non-empty and the cut-set has maximum weight.*

Miscellaneous definitions and examples

Definition 34. *A **bipartite graph** is a graph whose vertices can be divided into two disjoint sets A and B such that every edge connects a vertex in A to one in B.*

In general, global maximum cut of graphs is very hard to find. However, for a bipartite graph with nonnegative weights the global maximum cut is $C(A, B)$.

Exercise 3. *Show that every tree is bipartite.*

We won't prove it here, but we can show that a graph is bipartite iff it contains no cycles of odd length.

Definition 35. *An **independent set** is a set of vertices $I \subseteq V$ in a graph such that no two in the set are adjacent.*

Definition 36. *A **vertex cover** of a graph is a set of vertices $C \subseteq V$ such that each edge of the graph is incident to at least one vertex in C.*

Definition 37. *An **edge cover** of a graph is a set of edges $C \subseteq E$ such that every vertex of the graph is incident to at least one edge in C.*

The definitions above are related to the definition for a bipartite graph! In fact, each side of a bipartite graph is necessarily both an independent set *and* a vertex cover.

Definition 38. *A **matching** M in a graph G is a set of edges such that no two edges share a common vertex.*

Definition 39. *A **maximal matching** is a matching M of a graph G with the property that if any edge of G not already in M is added to M, the set is no longer a valid matching.*

Definition 40. *A **maximum matching** of graph G is the matching that contains the greatest number of edges.*

All maximum matchings must be maximal, but the opposite is not true.

Definition 41. *A **perfect matching** of a graph G is a matching such that for any vertex v in G, there exists an edge in the matching that is adjacent to v.*

Combinatorics

This section includes some handy theorems and examples that will help you out a lot in 305 and in your other core courses! Some of these won't ever go away, and it's helpful to see them over and over (and over) again so that you can become familiar with them.

Definition 42. *Let k, n be integers such that $k \leq n$. An **ordered arrangement** of k elements from a set of n distinct elements is called a **k-permutation**. An **unordered collection** of k elements from a set of n distinct elements is called a **k-combination**.*

Theorem 10. *The number of k-permutations from a set of n elements is $\frac{n!}{(n-k)!}$, and the number of k-combinations is $\frac{n!}{k!(n-k)!}$.*

We write $\frac{n!}{k!(n-k)!}$ as $\binom{n}{k}$ and say it as "n choose k."

Proof. (*Informal.*) There are n ways to pick the first element, $n-1$ ways to pick the second element, and so on. Therefore there are $n \cdot (n-1) \cdot \cdots \cdot (n-k+1) = \frac{n!}{(n-k)!}$ k-permutations. There are $k!$ ways to order k elements, so there are $\frac{n!}{k!(n-k)!}$ k-combinations. □

Theorem 11. ***The binomial theorem.***

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k.$$

Proof. All terms in the left-hand side product are of the form $x^{n-k}y^k$ for some k. For a fixed k, we count how many ways we get a term $x^{n-k}y^k$. We have a collection of n instances of the term $(x+y)$ and choosing exactly k *y-terms* gives us y^k. The number of ways to choose exactly k *y terms* is a k-combination from a set of n elements. □

The $\binom{n}{k}$ term is called the *binomial coefficient*. The binomial theorem is a clever way of proving interesting polynomial and combinatorial qualities.

Example 11.

$$\sum_{k=0}^{n} \binom{n}{k} (-1)^k = 0.$$

To see this, apply the binomial theorem with $x = 1$ and $y = -1$.

Theorem 12. ***The pigeonhole principle.*** *If $n \geq k+1$ objects are put into k containers, then there is a container with at least two objects.*

If you'd like to, you can prove this by contradiction!

Theorem 13. ***The generalized pigeonhole principle.*** *If n objects are put into k containers, then there is a container with at least $\lceil n/k \rceil$ objects.*

Example 12. *Show that every single simple graph on n vertices with $n \geq 2$ has two vertices of the same degree.*

Proof. If G has no zero-degree nodes, then we have n nodes all with degree between 1 and $n-1$, so at least two of these nodes must have the same degree. If G has a single zero-degree node, then the other $n-1$ nodes must have degree between 1 and $n-2$, so again at least two must have the same degree. If G has more than one zero-degree node, then there are already at least two nodes of the same degree (zero!). □

Complexity Theory

The field of complexity theory deals with classifying computational problems according to their inherent difficulty and understanding how these classes of problems relate with each other. Knowing the given complexity of a problem (or being able to show it) can help us understand how to solve a problem. If we are having trouble developing an efficient algorithm for a particular problem, we may find it useful to prove the complexity of the problem: if it is a very difficult problem, we know we cannot hope to solve it correctly in an efficient manner so we should look for other options.

Decision problems are problems that are posed as *yes-no* questions of the input values. We represent a particular instance of a decision problem as an input string s, which encodes all the information about the instance of the problem we are interested in. Let X be the set of all problem instances represented by strings s that are solvable. A decision problem asks if $s \in X$.

Definition 43. *We say a problem X is **polynomial time reducible** to problem Y, denoted $X \leq_p Y$ if and only if there exists an algorithm for solving X in polynomial time plus the time needed to solve Y a polynomial number of times on polynomially sized input. We define $X =_p Y$ to denote the condition that $X \leq_p Y$ and $Y \leq_p X$.*

Intuitively $X \leq_p Y$ should be interpreted as roughly saying that Y is at least as hard X up to polynomial time.

Lemma 2. *If $X \leq_p Y$ and Y can be solved in polynomial time then so can X.*

Lemma 3. *If $X \leq_p Y$ and X cannot be solved in polynomial time, then neither can Y.*

Definition 44. *We define P as the class of decision problems that can be correctly decided in polynomial time, i.e., there exists some deterministic verifier $A(s)$ that will return True iff $s \in X$ in polynomial time.*

Definition 45. *We define NP as the class of decision problems for which a yes certificate can be verified in polynomial time. That is, if there exists a polynomial-time deterministic verifier $B(s,t)$ such that if $s \in X$, there is some $t : |t| \leq |s|$ (called a certificate) such that $B(s,t)$ returns True, otherwise (if $s \notin X$) then $B(s,t)$ returns False for all t.*

Definition 46. *Any problem X **NP-Hard** if for all problems $Y \in NP, Y \leq_p X$. That is, Y is polynomial-time reducible to X, so if we have a black-box algorithm to solve problem X, we may solve every instance of Y with a polynomial amount of work and polynomial number of calls to the black box.*

Definition 47. *A problem is NP-complete if it is in NP and is NP-hard.*

We now have a great list of problems that are NP-hard, which include SAT, 3-SAT, the vertex cover decision problem, the independent set decision prob- lem, integer programming and many more. Do not worry about the details of these problems right now, you will see them in much greater detail in the future.

The vertex cover decision problem asks whether a graph has a vertex cover of size at most k where G and k are the input to the problem. The independent set decision problem asks whether a graph has an independent set of size at least k where again G and k are the input to the problem.