PART

I

ECE358: Foundations of Computing

Taught by Prof. Shur

Section 1

Admin and Preliminary

Subsection 1.1

Lecture 1

Topics covered will include:

- Graphs, trees
- Bunch of sorts
- Fancy search trees; red-black, splay, etc
- DP, Greedy
- Min span tree, single source shortest paths
- Maximum flow
- NP Completeness, theory of computation
- Blockchains??
- Θ

Solutions will be posted on the window of SF2001. Walk there and take a picture.

1.1.1 Mark Breakdown

Table 1. Mark Breakdown

Homework x 5	25
Midterm (Open book)	35
Final (Open book)	40

Section 2

Complexities

Subsection 2.1

Lecture 2

This lecture we talked about big O notation. For notes on this refer to my tutorial notes for ESC180, ESC190: https://github.com/ihasdapie/teaching/

Definition 1 Big O notation (upper bound)
$g(n)$ is an asymptotic upper bound for $f(n)$ if:

$$O(g(n)) = \{f(n) : \exists \quad c, n_0 \quad s.t. \quad 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_o\} \tag{2.1}$$

Proof **What is the big-O of $n!$?**

$$n! \leq n \cdot n \cdot n \cdot n \ldots n = n^n \Rightarrow n! \in O(n^n) \tag{2.2}$$

□

Definition 2 Big Ω notation (lower bound)
$h(n)$ is an asymptotic lower bound for $f(n)$ if:

$$\Omega(h(n)) = \{f(n) : \exists \quad c, n_0 > 0 \quad s.t. \quad 0 \leq c \cdot h(n) \leq f(n), \forall n \geq n_o\} \tag{2.3}$$

Proof **Find Θ for** $f(n) \sum_i^n i$.
For this we will employ a technique for the proof where we take the right half of the function, i.e. from $\frac{n}{2} \ldots n$ and then find the bound

$$\begin{aligned} f(n) &= 1 + 2 + 3 \ldots + n \\ &\geq \lceil \frac{n}{2} \rceil + (\left\lceil \frac{n}{2} \right\rceil + 1) + (\left\lceil \frac{n}{2} \right\rceil + 2) + \ldots n \quad n/2 \text{ times} \\ &\geq \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{2} \right\rceil + \ldots \left\lceil \frac{n}{2} \right\rceil \\ &\geq \frac{n}{2} \cdot \frac{n}{2} \\ &= \frac{n^2}{4} \end{aligned} \tag{2.4}$$

And therefore for $c = \frac{1}{4}$ and $n = 1$, $f(n) \in \Theta(n^2)$

□

Definition 3 Big Θ notation (asymptotically tight bound)

$$\Theta(g(n)) = \{f(n) : \exists \quad c_1 c_2, n_0 \quad s.t. \quad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_o\} \tag{2.5}$$

Proof Prove that

$$\sum_{j=1}^{n} i^k = \Theta(n^{k+1}) \tag{2.6}$$

First, prove $O(f(n)) = O(n^{k+1})$

$$\begin{aligned} f(n) = \sum_{j=1}^{n} i^k &= 1^k + 2^k + \ldots n^k \\ &\leq n^k + n^k + \ldots n^k \\ &= n^{k+1} \end{aligned} \tag{2.7}$$

Next, prove $\Omega(f(n)) = \Omega(n^{k+1})$

$$
\begin{aligned}
f(n) = \sum_{j=1}^{n} i^k &= 1^k + 2^k + \dots n^k \\
&= n^k + (n_1)^k + \dots 2^k + 1^k = \sum_{i=1}^{n} (n-i+1)^k \qquad (2.8) \\
&\geq \frac{n}{2}^k * n \geq \frac{n^{k+1}}{2^k} = \Omega(n^{k+1})
\end{aligned}
$$

Therefore $f(n) = \Theta(n^{k+1})$ □

Note that we may not always find both a tight upper and lower bound so not all functions have a tight asymptotic bound.

Theorem 1 **Properties of asymptotes:**
Note: $\wedge$ means AND
Transitivity [1]

$$(f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n))) \Rightarrow f(n) = \Theta(h(n)) \qquad (2.9)$$

Reflexivity[2]

$$f(n) = \Theta(f(n)) \qquad (2.10)$$

Symmetry

$$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n)) \qquad (2.11)$$

Transpose Symmetry

$$
\begin{aligned}
f(n) = O(g(n)) &\iff g(n) = \Omega(f(n)) \\
f(n) = o(g(n)) &\iff g(n) = \omega(f(n))
\end{aligned} \qquad (2.12)
$$

[1] *The f*

[2] *The foll*

Runtime complexity bounds can sometimes be used to compare functions. For example, $f(n) = O(g(n))$ is like $a \leq b$

- $O \approx \leq$
- $\Omega \approx \geq$
- $\Theta \approx \approx$
- $o \approx <$; an upper bound that is **not** asymptotically tight
- $\omega >$ a lower bound that is **not** asymptotically tight

Note that there is no trichotomy; unlike real numbers where we can just do $a < b$, etc, we may not always be able to compare functions.

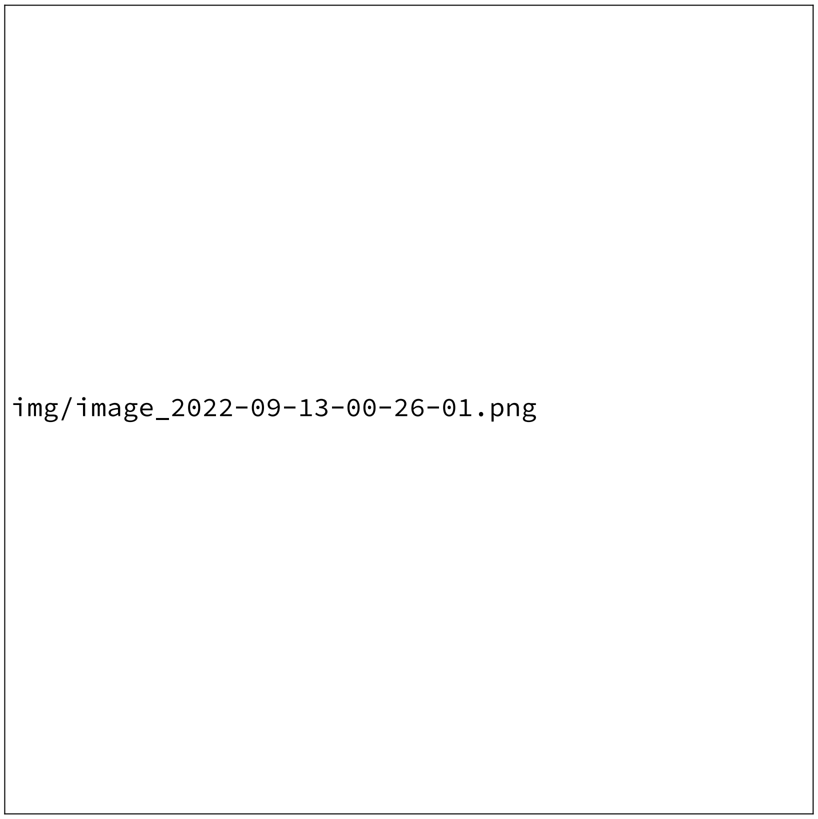

Figure 1. Complexity Cookbook

Subsection 2.2

Lecture 3: Logs & Sums

2.2.1 Functional Iteration

Recall:

$$a = b^c \Leftrightarrow log_b a =$$

$f^{(i)}(n)$ denotes a function iteratively applied i times to value n.

For example, a function may be defined as:

$$f^{(i)}(n) = \begin{cases} f(n) & \text{if } i = 0 \\ f(f^{(i-1)}(n)) & \text{if } i > 0 \end{cases} \tag{2.14}$$

Given (**??**) we see that

1. $f(n) = 2n$
2. $f^{(2)}(n) = f(2n) = 2^2 n$
3. $f^{(3}(n) = f(f^{(2)}(n)) = 2^3 n \vdots$
4. $f^{(i)}(n) = 2^i n$

As an exercise we may look at an iterated logarithm function, 'log star'

$$lg^*(n) = \min\{i \geq 0 : lg^{(i)} n \leq 1\} \tag{2.15}$$

This describes the number of times we can iterate $log(n)$ until it gets to 1 or smaller.

- $log^*2 = 1$
- $log^*4 = 2 = log^*2^2 = 1 + log^*2 = 2 \vdots$
- for practical reasons log^* doesn't really get bigger than 5. This is one of the slowest growing functions around.

Summations & Series

PROOF Proof for a finite geometric sum:

$$\begin{aligned}\sum_{k=0}^{n} x^k &= S \\ S &= 1 + x + x^2 \dots x^n \\ xS &= x + x^2 + x^3 \dots x^{n+1} \\ S &= \frac{1 - x^{n+1}}{1 - x}\end{aligned} \tag{2.16}$$

□

$$\sum_{i=1}^{\infty} x^i = \frac{1}{1-x} \quad \text{if } |x| < 1 \tag{2.17}$$

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \quad \text{if } |x| < 1 \tag{2.18}$$

PROOF Begin by differentiating both sides over x

$$\sum_{k=0}^{\infty} x^k = \frac{1}{(1-x)} \quad \text{if } |x| < 1 \tag{2.19}$$

$$\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2} \tag{2.20}$$

And then multiply both sides by x, therefore (**??**) follows. □

Telescoping Series

$$\sum_{k=1}^{n} a_k - a_{k-1} = a_n - a_0 \tag{2.21}$$

PROOF Write it out and cancel out terms

$$(a_1 - a_0) + (a_2 - a_1) \dots (a_n - a_{n-1}) = a_n - a_0 \tag{2.22}$$

Therefore the sum telescopes □

Another telescoping series may be proved similarly:

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} \xrightarrow{math} \sum_{k=1}^{n-1} (\frac{1}{k} - \frac{1}{k+1}) = (1 - \frac{1}{2}) + \dots (\frac{1}{n-1} - \frac{1}{n}) = a_o - a_n \tag{2.23}$$

SUBSECTION 2.3

Lecture 4: Induction & Contradiction

2.3.1 Induction

The general steps for proving a statement by induction are:

1. Basis
2. Hypothesis
3. Inductive step

I.e. if the basis holds for some i, i.e. $0, 1, 2, 3, 12, \ldots$ AND if we assume that the hypothesis holds for an arbitrary number k, then we just need to prove that the inductive step follows, or that $P(n+1)$ holds.

Example Prove that $P(n) = 1 + 2 + 3 + \ldots n = \frac{n(n+1)}{2}$

Proof

1. Basis: $P(0) = 0 = \frac{0(0+1)}{2}$
2. Hypothesis (assume that it is true): $P(k) = \frac{k(k+1)}{2}$
3. Inductive step (need to prove $P(k+1) = \frac{(n+1)(n+2)}{2}$): $P(k+1) = \underbrace{1 + 2 + \ldots + n}_{\frac{n(n+1)}{2}} + (n+1) = \ldots = \frac{n^2+3n+2}{2} = \frac{(n+1)(n+2)}{2}$

□

Example Show that for any finite set S, the power set 2^S has $2^{|S|}$ elements (that is, there are $2^{|S|}$ distinct subsets of S)

Proof

1. Basis:
$$n = 0, |S| = 0, |2^S| = 1 = 2^0 \tag{2.24}$$
$$n = 1, |S| = 1, |2^S| = 2 = 2^1 \tag{2.25}$$
2. Hypothesis: Assume that 2^S has 2^n elements when $|S| = n$
3. Inductive step: need to prove that when $|S| = n + 1, |2^S| = 2^{n+1}$

   Let $B = S \not\{a\}$ for some $a \in S$. Now there are two types of subsets of S; those that include a and those who do not include a

   For subsets that do *not* include a , $|2^B| = 2^{|B|} = 2^n$, by the hypothesis.

   For subsets that do include a, these sets are of size $2^B \cup \{a\}, which is 2^n$.

   Therefore the total number of subsets of S is $2^n + 2^n = 2^{n+1}$, as desired.

□

The same kind of argument can be applied to problems such as the Towers of Hanoi and the tiling problem.

2.3.2 Contradiction

1. Assume the theorem is false
2. Show that the assumption is false (leads to a contradiction)
   - Therefore the theorem is true

Example Prove that $\sqrt{2}$ is irrational

PROOF Assume that $\sqrt{2}$ is rational.
Therefore we can write $\sqrt{2}$ as

$$\sqrt{2} = \frac{a}{b} \tag{2.26}$$

Where a, b **have no common factors**.
We can square both sides

$$2 = \frac{a^2}{b^2} \rightarrow a^2 = 2b^2 \tag{2.27}$$

Therefore a^2 is even.
Let $a = 2c$

$$2^2c^2 = 2b^2 \rightarrow b^2 = 2c^2 \tag{2.28}$$

Therefore b is even as well.
This results in a contradiction since we assumed that a, b have no common factors, but our analysis shows that both would have to be even (and share a common factor of 2). □

SUBSECTION 2.4

Lecture 5: recurrences

Many recursive algorithms can be thought of as a divide-and-conquer approach where we break the problem into subproblems that are similar to the origianl but smaller in size, solve them resurisively, then combine them to create a solution to the original problem.

Definition 4 A recurrence is a function defined in terms of:

- 1+ base cases
- Itself, with smaller arguments

For example, finding a Fibonacci number is a recurrence;
$T(n) = T(n-1) + T(n-2)$ with some base cases.

Example **Mergesort**
Sorting $[3, 1, 7, 5]$

1. Divide: break into partitions: $[[3, 1], [7, 5]]$
2. Sort partitions: $[[1, 3], [5, 7]]$
3. Create result array
4. Compare: have two pointers to front of array
   - compare 1, 5. 1 is smaller; $result = [] \leftarrow 1$
   - Move ptr to left array (1, 3) ahead one. Compare 3, 5. 3 is smaller, so $result = [3] \leftarrow 3$
   - One of the arrays is now empty so we can just append the rest
5. $result = [1, 3, 5, 7]$

Definition 5 Pseudocode for mergesort is given by:

```
        mergesort(A, p, r)
                if p < r
                        q = [(p+r)/2]
                        mergesort(A, p, q)  // N/2
                        mergesort(A, q+1, r)  // N/2
                        merge(A, p, q, r) // merge the sorted
                         ↪  subarrays
```

This mergesort partitions in half each time[3].
In the worst case we will compare $N-1$ times, so $O(N)$ worst case.
Proving that merge sort is $\Omega(N)$

How much time does MergeSort take?
The time of mergesort is defined recursively as:

$$T(N) = \begin{cases} O(1) & n = 1 \\ T(N) = 2T(\frac{N}{2} + \Theta(N) & \end{cases} \tag{2.29}$$

More generally we can find the runtime of a recurrence algorithm via

Note that $\Theta(N)$ is merge operation

- Recurrence trees
- Substitution
- Master theorem

2.4.1 Recurrence Trees

Example Recurrence trees can be used to find the time complexity of mergesort.

img/image_2022-09-19-12-26-07.png

The height of the tree is $\log N$
The total cost is the total cost per level times the number of levels, which is

$$N \cdot logN \tag{2.30}$$

So the complexity is $O(N \log N)$

2.4.2 Substitution

1. Guess the answer
2. Apply induction

Example Determine an asymptotic upper bound on $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + \Theta(n)$

Proof This expression can be simplified since we don't care too much about floors or ceilings for asymptotic behaviour.

$$T(n) = 2T(\frac{n}{2}) + N \tag{2.31}$$

Let's guess that the upper bound is $O(n \log n)$
Then, we need to prove that $T(n) < C \cdot n \log n$ for some $C > 0$. Let's apply induction.

1. Basis: this is tricky since if $n = 1$ we end up with $T(1) \leq C \cdot 1 \cdot \log 1 = 0$ which cannot hold since that would just not make sense. Instead, observe that $T(1) = 1$, $T(2) = 2T(1) + 2 = 4, T(3) = 2T(1) + 3 = 5, T(4) = 2T(2) + 4 \ldots$.

   So $T(n)$ is therefore independent of $T(1)$, so we can use two bases, $T(2), T(3)$. Since $T(2) \leq C * 2 \log 2 = 2C, T(3) \leq C \cdot \log$
2. Hypothesis: Assume that the upper bound holds for all possible $m < n$, let $m = \left\lfloor \frac{n}{2} \right\rfloor$. This yields $T(\frac{n}{2}) \leq C \cdot \left\lfloor \frac{n}{2} \right\rfloor \cdot \log \left\lfloor \frac{n}{2} \right\rfloor$
3. Inductive step: substitute hypothesis into recurrence yields

$$T(N) \leq C \cdot (C \cdot \left\lfloor \frac{N}{2} \right\rfloor \cdot \log \left\lfloor \frac{N}{2} \right\rfloor + N = cN \log N - (1-c)N \leq Cn \log n \tag{2.32}$$

□

A few pitfalls to avoid is guessing $T(n) = O(n) = c \cdot n$ and so forth we would get

$$T(N) \leq 2C \cdot \left\lfloor \frac{n}{2} \right\rfloor + n = cn + n = (c+1)n \tag{2.33}$$

This would be wrong since we cannot change the constant to $c+1$; we have to prove it with exactly the hypothesis given.

2.4.3 Master Theorem

Definition 6 The master method applies to recurrences of the form

Proof is out of scop course

$$T(n) = aT(\frac{n}{b}) + f(n) \qquad \text{where } a \geq 1, b > 1, f \text{asymptotically positive} \tag{2.34}$$

It distinguishes 3 common cases b comparing $f(n)$ with $n^{\log_b a}$

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$ and $af(\frac{n}{b}) \leq cf(n)$ for some $c < 1$, then then $T(n) = \Theta(f(n))$

There are a few technicalities to be a ware of. In each example we compare $f(n)$ with $F = n^{\log_b a}$ and take the larger of each as the solution the recurrence. For the first case we note that $f(n)$ must be *polynomially* smaller than F; i.e. it must be asymptotically smaller than F by some factor of n^{ε}. In the third case $f(n)$ must be greater than F as well as being polynomially larger and satisfy the regularity condition $af(\frac{n}{b}) \leq cf(n)$. There are areas where the master theorem does not cover, for example a gap between cases 1, 2 where $f(n) > F$ but is not polynomially larger. If $f(n)$ falls in one of these gaps or the regularity condition does not hold, the master method cannot be used to solve the recurrence.

Example What is the closed form of $T(n) = T(\frac{2n}{3}) + 1$?

PROOF a = 1, b = 2/3, f(n) = 1.

$$\log_b a = \log_{\frac{2}{3}} 1 = 0 \tag{2.35}$$

$$f(n) = \Theta(n^0) \tag{2.36}$$

So

$$T(n) = O\log(n) \tag{2.37}$$

□

SUBSECTION 2.5

Lecture 6

2.5.1 Graphs

Definition 7 A **graph** is a data structure comprised from set of vertices V and a set of edges E, where each edge connects a pair of vertices. A **directed graph (digraph)** is a graph where edges E have a *direction*, i.e. an edge (u, v) is different from (v, u). Conversely, an **undirected graph** is a graph where edges E do not have a direction.

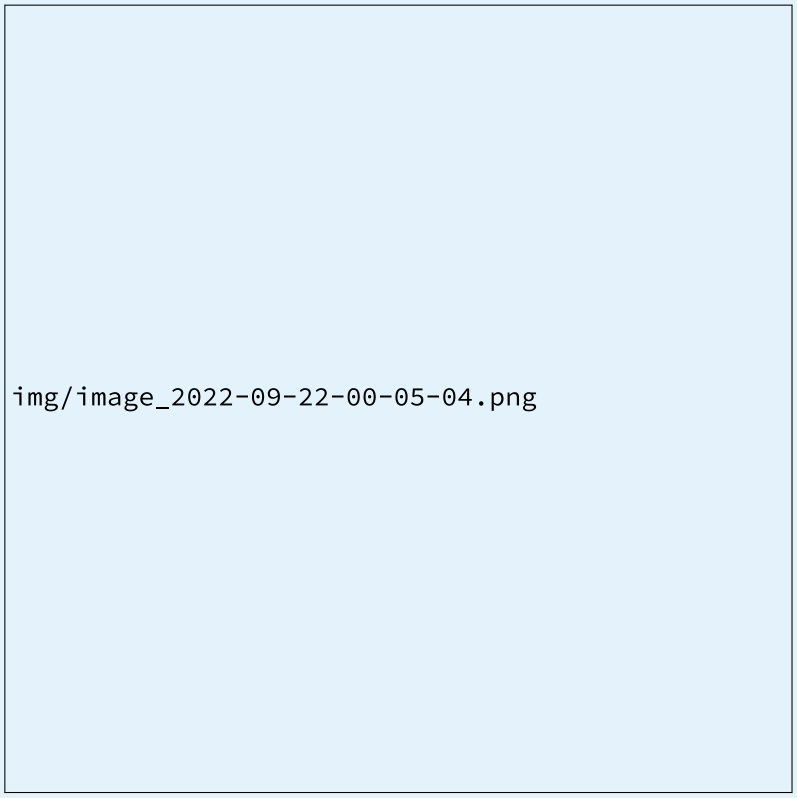

Figure 2. (a) directed graph, (b) undirected graph, (c) a subgraph of (a)

Some conventions:

- Edges are denoted by (u, v)[4] where $u, v \in V$

- If (u, v) is a edge in a directed graph, then (u, v) is incident from or leaves u, and is incident to or enters v
- If (u, v) is a edge a graph, then u , v are adjacent
- The **degree** of a vertex is the number of edges incident to it
- A **path** is a sequence of vertices $(v_0, v_1, \ldots v_k)$ from from vertex to another such that each vertex is incident[5] to the ones prior and after.
  - If there exists a path from a to b then b is **reachable** from a and a
  - A path is **simple** if no vertex is repeated
  - A path forms a cycle if the first and last vertices are the same
- A directed graph with no self-loops is **simple**
- A graph with no simple cycles is acyclic
- An undirected graph is **connected** if there exists a path between any two vertices
- A directed graph is **strongly connected** if every vertex is reachable from every other vertex
- Two graphs V, V' are **isomorphic** if there exists a bijection[6] between the vertices of the two graphs such that the edges are preserved
- Given graph $G, G' = (V', E')$ is a **subgraph** of G if $V' \subseteq V$ and $E' \subseteq E$
- Given a set $V' \subseteq V$, the subgraph of G **induced** by V' is the graph $G' = (V', E')$ where $E' = \{(u, v) \in E | u, v \in V'\}$
- Given an undirected graph, the **directed version** of G is $G = (V, E')$ where $(u, v) \in E'$ if and only if $(u, v) \in E$. In other words we replace all undirected edges and replace them with their directed counterpart.
- The corollary can be applied to a directed graph to get the **undirected version** of G.
- A **neighbor** of u in a directed graph is any vertex v such that $(u, v) \in E$ where E is the set of edges for the undirected counterpart of the graph
- A **complete** graph is a graph where every pair of vertices are connected by an edge
- A **bipartite graph** is an undirected graph G in which it's V can be partitioned into two disjoint sets V_1, V_2 such that every edge $(u, v) \in E$ connects a vertex in V_1 to a vertex in V_2 or vice-versa
- An acyclic undirected graph is a **forest**
- A connected acyclic undirected graph is a **free tree**.
  - A directed acyclic graph is often termed a DAG
- A multi-graph is a graph where edges can be repeated and self-loops are allowed
- A hyper-graph is a graph where edges can connect more than two vertices
- The **contraction** of an undirected graph G by an edge $e = (u, v)$ is a graph G' where $V' = V - \{u, v\} \cup \{x\}$, where x is a new vertex. Then, for each edge connected to u, v the edges are deleted and then reconstructed with x, effectively 'contracting' u, v into a single vertex

In code graphs are commonly represented as adjacency lists or adjacency matrices. This was covered in ESC190, but for reference:

`img/image_2022-09-22-00-29-19.png`

Table 2. Time complexities of graph representations

	adjacency list	matrix
Time	$O(n)$	$O(1)$
Memory	$O(E)$	$O(n^2)$

2.5.2 Trees

A **tree** is a common and useful subset of graphs img/image_2022-09-22-00-39-29.png

Definition 8 A tree is a common subset of graphs, i.e. ones that are **connected, acyclic, and undirected**. This gives a few useful properties, i.e. the existence of a *root* node, the parent-child relationship, and the existence of a unique path between any two nodes.

Some conventions for trees

- Depth of node: length from root to node
- Height length of longest path from node to leaf
- Degree of node: number of children. Binary trees have degree 2, n-ary tree has degree n

Theorem 2 All of the following statements are equivalent for a tree $T = (V, E)$:

1. $\forall v \in V, v$ is a tree; all nodes in a tree are trees unto themselves.
2. Every two nodes are connected by a unique path
3. T is connected by if any edge is removed the resulting graph is disconnected T is connected and $|E| = |V| - 1$ T is acyclic and $|E| = |V| - 1$ T is connected but if a edge is added the resulting graph has a cycle

Subsection 2.6

Lecture 7, 8: Probability and Counting

Most of the probability stuff is review from ECE286 so I'll be omitting most notes.

Definition 9 Probability distribution $Pr\{\}$: mapping from events of S to real numbers where

1. $Pr\{\emptyset\} = 0$
2. $Pr\{A\} \geq 0 \forall A \in S$
3. $Pr\{S\} = 1$
4. $Pr\{A \cup B\} = Pr\{A\} + Pr\{B\} \forall A, B \in S$

For any two events A, B we can define the triangle inequality

The complement of a
is $\overline{A} = S - A$, and th
bility is $Pr\{\overline{A}\} = 1$

Definition 10 $Pr\{A \cup B\} \leq Pr\{A\} + Pr\{B\}$

Definition 11 Baye's thereom

$$Pr\{A \mid B\} = \frac{Pr\{B \mid A\} Pr\{A\}}{Pr\{B\}} = \frac{Pr\{A\} Pr\{B|A\}}{Pr\{A\} Pr\{B|A\} + Pr\{\overline{A}\} Pr\{B|\overline{A}\}} \tag{2.38}$$

The expected value of a random variable is

$$E[X] = \int_{-\infty}^{\infty} x Pr\{X = x\} dx \tag{2.39}$$

And in the discrete case,

$$E[X] = \sum_{x \in \mathbb{Z}} x Pr\{X = x\} \tag{2.40}$$

The variance is

$$Var[X] = E\left\{(X - E[X])^2\right\} = E[X^2] - E^2[X] \tag{2.41}$$

The **standard devia**
is the positive squar
the variance.

Subsection 2.7
Lecture 9: Heapsort

Heapsort is a sorting algorithm that runs in $O(nlog(n))$ and sorts its elements in place by leveraging a *heap*.

Th
ES

Definition 12 A heap is a data structure that satisfies the *heap property*. For a max heap, the value of a mode is at most the value of its parent and vice-versa for a min-heap. Heaps are commonly used to implement a priority queue because it offers a `pop{min, max}` operation in $O(1)$ time and insertion in $O(log(n))$ time. Another reason for its use is it's ability to be represented as an array with `Parent(i) = floor(i/2)`, `Left(i) = 2iRight(i) = 2i+1`

The `MAXHEAPIFY` method is used to coerce heaps back into fulfilling the heap property by 'bubbling down' nodes until the heap property is satisfied. The intuition for this is that as long as the heap property is not satisfied[7], node i is exchanged for the largest of its' children. Since the heap property is now satisfied for i but not necessarily its' children, this heapify operation is then recursively called on the larger of i's children.

[7] *which we can tell by looking at the node's children*

```
MAXHEAPIFY(A, i)
        l = LEFT(i)
        r = RIGHT(i)
        if l <=A.size and A[l] > A[i]
                largest = l
        else largest = i

```

```
HEAPSORT(A)
for i = A.length -> 2,
        swap A[1], A[i]
        A.heapsize = A.heapsize - 1
        HEAPIFY(A, 1)
```

Subsection 2.8

Lecture 10, 11: Quicksort

Subsection 2.9

Lecture 12: lower bound, counting and radix sort

Subsection 2.10

Lecture 13: Select & BST