ECE358: Foundations of Computing

Aman Bhargava

September-December 2020

Contents

0.1 Introduction and Course Information

This document offers an overview of the ECE358 course. They comprise my condensed course notes for the course. No promises are made relating to the correctness or completeness of the course notes. These notes are meant to highlight difficult concepts and explain them simply, not to comprehensively review the entire course.

Course Information

- Professors: Prof. Andreas Veneris and Prof. Zissis Poulos
- Course: Engineering Science, Machine Intelligence Option
- Term: 2020 Fall

Chapter 1

Math Review

1.1 Logarithms

Logarithm Rules:

- **Definition:** $a = b^c$ iff $log_b a = c$.
- $a = b^{\log_b a}$.
- $\log_c(ab) = \log_c a + log_c b$.
- $\log_c(a^n) = n \log_c(a)$.
- $\log_b(a) = \log_a(b)$.
- $\log(1/a) = -\log a$.
- $\log(a/c) = \log a - \log c$.
- $a^{\log n} = n^{\log a}$.

Variations on Logarithm

$$\log^{(i)} n = \begin{cases} n & \text{iff } i = 0 \\ \log(\log^{(i)} n) & \text{otherwise} \end{cases} \tag{1.1}$$

$$\log^* n = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + log^*(\log n) & \text{otherwise} \end{cases} \tag{1.2}$$

1.2 Sequences and Series

Theorem 1 *Fibonnaci Definition:*

$$F_i = F_{i-1} + F_{i-2} \tag{1.3}$$

Where $F_0 = 0,\ F_1 = 1.$

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}} \tag{1.4}$$

Where $\phi = \frac{1+\sqrt{5}}{2},\ \hat{\phi} = \frac{1-\sqrt{5}}{2}$

Theorem 2 *Arithmetic Series:*

$$\sum_{i=1}^{n} = \frac{n(n+1)}{2} = \Theta(n^2) \tag{1.5}$$

Theorem 3 *Geometric Series:*

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1} \tag{1.6}$$

Theorem 4 *Infinite Series:*

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \ \textit{iff}\ |x| < 1 \tag{1.7}$$

Theorem 5 *Telescoping Series:*

$$\sum_{i=1}^{n} a_i - a_{i-1} = a_n - a_0 \sum_{i=1}^{n} a_i - a_{i+1} = a_0 - a_n \tag{1.8}$$

1.3 Combinatorics

Theorem 6 *Binomial Coefficient:*

$$(x+y)^r = \sum_{i=0}^{r} \binom{r}{i} x^i y^{r-i} \tag{1.9}$$

Where $\binom{r}{i}$ *is the binomial coefficient which equals*

$$\frac{r!}{i!(r-i)!}$$

Chapter 2

Asymptotics

What are Asymptotics? Analysis method for performance and complexity of an algorithm (space/memory, time/clock cycle complexity).

- Tight Bound: $\Theta()$.
- Worst Case: $O()$.
- Best Case: $\Omega()$
- Average & Expected Case: *Randomized + probabilistic analysis. Not a focus for the course.*
- Amortized Case: Discused later on. "Average worst case".

Useful Facts

- Θ bound is not always possible to find.
- **Transitivity:** if $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.
- **Symmetry:** $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
- **Transpose:** $f(n) = O(g(n))$ if and only if $g(n)\Omega(f(n))$.
- $n^a \in O(n^b)$ iff $a \leq b$.
- $\log_a(n) \in O(\log_b(n))\, \forall a, b$.
- $c^n \in O(d^n)$ iff $c \leq d$.

- If $f(n) \in O(f'(n))$ and $g(n) \in O(g'(n))$ then

$$f(n) \cdot g(n) \in O(f'(n) \cdot g'(n))$$

$$f(n) + g(n) \in O(\max(f'(n), g'(n))$$

Theorem 7 *Big O Definition: $f(n) = O(g(n))$ if and only if: There exists $c > 0$ and $n_0 > 0$ such that*

$$0 \leq f(n) \leq cg(n) \, \forall n \geq n_0 \tag{2.1}$$

Theorem 8 *Big Ω Definition:*

"Lower Bound" Big Ω Definition: *$f(n) = \Omega(h(n))$ if and only if: There exists $c_1, n_1 > 0$ such that*

$$0 \leq c_1 h(n) \leq f(n) \, \forall n \geq n_1 \tag{2.2}$$

Theorem 9 *Big Θ Definition:*

"Tight Bound" Big Θ Definition: *$f(n) = \Theta(g(n))$ if and only if:*
There exists $c_1, c_2, n_0 > 0$ such that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \, \forall n > n_0 \tag{2.3}$$

Chapter 3

Graphs

3.1 Definitions

Theorem 10 *Graph Definition: A* ***graph*** *is a collection of* ***vertices*** *and* ***edges*** *represented by*

$$G = (V, E) \tag{3.1}$$

Graphs can be:

- *Directed or Undirected.*
- *Weighted or Unweighted.*

Path: A sequence of **edges** between adjacent vertices.

- **Simple Path:** No vertex is repeated in the path.
- **Cycle:** A path that starts and ends with the same vertex.
- A **connected** graph has a path between any two vertices (else it is **disconnected**)

Bipartite Graphs: A graph is **bipartite** if and only if vertices V can be divided into V_1, V_2 such that:

1. $V_1 \cap V_2 = \emptyset$.
2. $V_1 \cup V_2 = V$.
3. Adjacencies exist only between elements and V_1 and V_2 (i.e., vertices within each set are disconnected from eachother, but are connected to elements from the other set).

Vertex Degree is the number of **edges** adjacent to a vertex (includes incoming and outgoing edges).

- **In-degree** is the number of incoming edges.
- **Out-degree** is the number of outgoing edges.

Clique: For all $v_1, v_2 \in V$, there exists an edge connecting them (i.e. a fully-connected set of vertices).

Representations for Graphs: We either use an **adjacency matrix** or an **adjacency matrix**.

- **Adjacency List:** Each element in the list represents a vertex and "has" a collection of **pointers** connecting them to other elements of the list.
  - **Time Complexity:** $O(n)$ – Determining if E_{v_1,v_2} exists would, at worst, require you to check n pointers arising from vertex v_1 and/or v_2.
  - **Space Complexity:** $O(|E|)$ – with worst case that $|E| = n^2$ for clique.
- **Adjacency Matrix:** Matrix contains weights connecting v_i to v_j at index $M_{i,j}$.
  - **Time Complexity:** $O(1)$ – Just check address $M_{i,j}$ and $M_{j,i}$.
  - **Space Complexity:** $O(n^2$ every time.

3.2 Graph Traversals

How do we choose an order to traverse all nodes in a graph?

3.2.1 Breadth First Search

Key Ideas:

- From a given point, we traverse all immediate neighbours.
- Only then do we go to second-order neighbours of the original point.
- Implemented with a **Queue** (first-in-first-out).

Breadth First Search Algorithm: Given $G = (V, E, s)$ where s is the starting vertex,

1. Initialize $d[u] = \infty \, \forall u \in V$. This is the **distance** for each vertex.
2. Let Q be an empty queue, and **enqueue** vertex s.
3. While Q is not empty:
   (a) $u = \text{dequeue}(Q)$.
   (b) For $v \in adj(u)$:
      i. if $d[v] = \infty$, let $d[v] = d[u] + 1$ and **enqueue** $v \to Q$.

Performance: Time complexity is $O(|V| + |E|)$.

3.2.2 Depth First Search

Key Ideas:

- **Input:** $G = (V, E)$ (no source vertex this time).
- **Output:**
  1. $d[v]$: Time taken to **discover** vertex v (in number of algorithm "steps").
  2. $f[v]$: Time taken to **finish** with vertex v (meaning to exhaust all 'down stream' vertices).
  3. $\Pi[v]$: Predecessor of v (i.e. the vertex immediately before it in the search).

Color Coding:

- **White:** Node is as of yet undiscovered.
- **Gray:** Node has been discovered, but the descendants have not all been discovered (node is not "exhausted").
- **Black:** The algorithm has traversed the node AND all its descendants.

Algorithm 1: Depth First Search Algorithm.

```
Input   : $G = (V, E)$
Output: List of discovery times, finishing times, and predecessors
        for each $v \in V$.

color[$u$] = White $\forall u \in V$;
time = 0;
for each $u \in V$ do
    if color[$u$] = White then
        DFS-Visit($u$);
    end
end
```

Algorithm 2: DFS-Visit Algorithm (subroutine for DFS Search)

```
Input   : Vertex $u$
Output: Recursively traverses nodes depth-first.

color[$u$] $\leftarrow$ Gray;
time+ = 1;
$d[u]$ = time;
for each $v \in adj(u)$ do
    if $v$ = White then
        DFS-Visit($v$);
    end
end
color[$u$] $\leftarrow$ Black;
time+ = 1;
$f[u] \leftarrow$ time;
```

Edge Classification: **DFS** Produces one or more **trees**, leading to the following classification.

1. **Tree Edges:** Edges found during DFS exploration.
2. **Back Edge:** (u, v) when u is a descendant of v.
3. **Forward Edge:** (u, v) when v is a descendant of u **but** (u, v) is not a tree edge.
4. **Cross Edge:** Any other type of edge.

Theorem 11 *White Path Theorem: v is a descendant of u if, and only if: At time $d[u]$, there exists a **white-only** path from $v \rightarrow u$.*

Theorem 12 *Parenthesis Theorem: For all $u, v \in V$: Exactly* ***one*** *of the following holds:*

1. $d[u] < f[u] < d[v] < f[v]$ *or* $d[v] < f[v] < d[u] < f[u]$. *That is,* ***neither*** u, v ***are descended from eachother***.
2. $d[u] < d[v] < f[v] < f[u]$. *That is, v is a descendant of u.*
3. $d[v] < d[u] < f[u] < f[v]$. *That is, u is a descendant of v.*

Theorem 13 *Depth First Search Edge Types: The depth-first search of* ***undirected graph*** *G yields only tree edges and back edges .*

3.2.3 Topological Sort

Key Idea: All DAG (directed, acyclic graphs) have *some* amount of **implied order**. Think of it like a dependency graph. We use topological sort to turn **partial order** $\rightarrow$ **total order** (that is, an absolute list).

Algorithm 3: Topological Sorting Algorithm.

```
Input  : DAG $G = (V, E)$
Output: Total ordering of elements of graph.

Call Depth First Traversal($V, E$);
Return the vertices in decreasing order of finish time;
```

3.3 Minimum Spanning Trees

Key Idea: Given a weighted graph $G = (V, E)$, we want a tree $G' = (E', V)$ where $E' \subseteq E$ such that there is a path between all vertices **and** $\sum_{e \in E'} e$ is minimized.

- $|E'| = |V| - 1$.
- There are **no cycles** (hence "tree").
- It may or may not be a **unique** solution.

3.3.1 Generic Minimum Spanning Tree Approach

High-Level Description:

- Set A is instantiated as empty.

- Edges from the graph are added to A, ensuring **loop invariant** $A \subset$ Some Minimum Spanning Tree.
- Edge (u, v) is **safe** iff $A \cup \{(u, v)\}$ is also a subset of some minimum spanning tree.

Generic MST Algorithm: Given graph G with weights w,

1. Let A be an empty set.
2. While A is not a spanning tree:
   (a) Find a **safe** edge (u, v).
   (b) $A \leftarrow A \cup (u, v)$.

Terminology: Relating to how to pick a **safe** edge.

- Let S represent the set of vertices currently "covered" by edges in A, our current working minimum spanning tree.
- We can partition V into set S and $V - S$.
- Edge (u, v) **crosses** the **cut** $(S, V - S)$ if one of u, v is in S and the other is in $V - S$.
- A cut **respects** A iff no edge in A **crosses** the cut.
- The **light edge** with respect to cut $(S, V - S)$ is the lowest weight edge that **crosses** the cut.
- **The light edge is the safe edge**.

3.3.2 Prims Algorithm

Algorithm 4: Prim's Algorithm

```
Input  : Graph $G = (V, E)$, weights $w$, root node $r$.
Output: Minimum spanning tree of $G$.

$Q \leftarrow \{\emptyset\}$;
for each $u \in V$ do
    key[$u$] $\leftarrow \infty$;
    $\Pi[u] \leftarrow$ NIL;
    Insert$u \rightarrow Q$;
end
Set key of root node $r$ to 0 in $Q$.
while $Q \neq \{\emptyset\}$ do
    $u \leftarrow$ Pop-Minimum($Q$);
    for each $v \in Adj(u)$ do
        if $v \in Q$ and $w(u, v) \leq key(v)$ then
            $\Pi[v] \leftarrow u$;
            Set the key of $v$ in $Q$ to $w(u, v)$;
        end
    end
end
```

Complexity: Time complexity is $O(|E| + |V| \log |V|)$ when using Fibonnaci heaps.

3.4 Shortest Paths

Goal: Find path of minimum total weight between two vertices in a graph. Shortest path from $u \rightarrow v$ is represented by $\delta(u, v)$. It's the set of vertices from one to the other.

Variants of Shortest Paths Problem:

1. Single Source Shortest Paths (SSSP).
2. Single Destination Shortest Paths.
3. Single Pair – *most efficient method is to just run SSSP.*
4. All Pairs Shortest Paths.

Key Algorithms:

1. Dijkstra's – for no negative weights.
2. Bellman-Ford – when there are negative weights.
3. Difference Constraints – a surprising application of shortest paths.

Properties and Assumptions of Shortest Paths:

1. No negative weight cycles allowed accessible from source node.
2. **Optimal Substructure:** If p is in the optimal path from $u \to v$ $\delta(u, v)$, then sub-path $u \to p$ is the optimal path $\delta(u, p)$.
3. A shortest path will **never contain a cycle**.
    - Negative weight cycles aren't allowed (per above).
    - Positive weight cycles would be a pointless cost expenditure.
    - 0-value cycles are redundant (by convention, we just skip them).

3.4.1 Djikstra's Algorithm

Key Points:

- No negative weight edges allowed.
- We let S be the set of vertices with known shortest path weight.
- Priority queue Q is the remaining set $V - S$ of vertices.
- Priority is dictated by $d[v]$.
- **Time Complexity:** If we use a binary heap, we get

$$O(|E| \log |V|)$$

  though Fibbonacci heap is faster.

Algorithm 5: Dijkstra's Algorithm

```
Input  : Graph $G = (V, E)$, weights $w$ and starting node $s$.
Output: Shortest paths for each $v \in V$ starting at $s$.

Init-Single-Source($V, s$);
$S \leftarrow \{\emptyset\}$;
$Q \leftarrow V$;
while $Q \neq \{\emptyset\}$ do
    $u \leftarrow$ Extract-Min($Q$);
    $S \leftarrow S \cup \{u\}$;
    for each $v \in adj(u)$ do
        Relax($u, v, w$);
    end
end
```

Algorithm 6: Initialize Single Source

```
Input  : Graph $G = (V, E)$ and source node node $s$
Output: Initialized Graph for starting Single Source Shortest Paths.

for each vertex $v \in V$ do
    $v.d \leftarrow \infty$;
    $v.\pi \leftarrow$ NIL;
end
$s.d \leftarrow 0$;
```

Algorithm 7: Relax Algorithm

```
Input  : Edges $u, v$ and weight function $w : V \times V \to \mathbb{R}$
Output: Corrects the path to $v$ from the source, using neighbour $w$
        if necessary.

if $v.d > u.d + w(u, v)$ then
    $v.d \leftarrow u.d + w(u, v)$;
    $v.\pi \leftarrow u$;
end
```

3.4.2 Bellman-Ford Algorithm

Key Points:

- Solves single-source shortest paths (allowing for **negative edges**, unlike Dijkstra's).

- Returns **False** if there is a negative weight edge.
- **Time Complexity:** $O(|V| \cdot |E|)$.

Algorithm 8: Bellman-Ford Algorithm.

```
Input  : Graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$, source
         node $s$
Output: Boolean indicating if the question is valid, single-source
         shortest path result if possible.

Initialize-Single-Source($G, s$);
for $i = 1$ to $|V| - 1$ do
    for each $(u, v) \in E$ do
        Relax($u, v, w$);
    end
end
for each $(u, v) \in E$ do
    if $v.d > u.d + w(u, v)$ then
        return FALSE;
    end
end
return TRUE;
```

3.4.3 Application: Difference Constraints

Key Points:

- Under specific conditions, we can convert Linear Programs into graph problems.
- We can then use **Bellman-Ford** to solve for a feasible solution to the linear program.

Requirements for Conversion to Graph Problem: Within $Ax \leq b$,

- Each row of a must have all zeros except for one -1 and one $+1$ entry.
- The graph approach only gives **a single feasible point**, if any.
- No additional cost optimization is done in this method.

Conversion to Graph Problem:

1. Vertices $V = \{v_0, v_1, ..., v_n\}$. Each v_i, $i \in \{1, ..., n\}$ represents an index of vector x, our feasible point. v_0 is a special additional point.
2. $E = \{(v_i, v_j) = x_i - x_j \leq b_k\} \cup \{(v_0, v_i) : 0 \forall i \in [n]\}$. In other words,
   - Each connection from v_i to v_j is directed with weight b_k, the difference constraint b_k that $x_i - x_j \leq b_k$.
   - We also have 0-weight edges connecting $v_0 \to v_i$ for all $i \in [n]$.

Solution:

$$x^* = \begin{bmatrix} \delta(v_0, v_1) \\ \delta(v_0, v_2) \\ \vdots \\ \delta(v_0, v_n) \end{bmatrix} \tag{3.2}$$

As determined by **Bellman-Ford** with v_0 as the source node.

3.5 Maximum Flow

Key Idea: Directed graphs can represent **flow** of resources. If we add properties of **conservation** and a few others, we can come up with the hard problem of how to allocate flow through the network to get the maximum total transmission from one side to the other, but not exceed the limit of each 'pipe' or overflow at a juncture.

3.5.1 Formal Definition

Theorem 14 *Flow Network Definition: Flow network $G = (V, E)$ is a **directed** graph where:*

- *Edges represent **capacity** c, and $c(u, v) \geq 0$.*
- *If E has edge (u, v), then (v, u) **cannot exist**.*
- *$c(u, v) = 0$ when $(u, v) \notin E$.*
- ***Two important vertices:** Source s and sink t.*
- *It is assumed that each vertex is on a path between s and t.*
- *Therefore, $|E| \geq |V| - 1$.*

Flow: ***flow*** *in G is a function mapping two vertices to a real number ($f : V \times V \to \mathbb{R}$) under the constraints:*

1. ***Capacity:*** $u \leq f(u,v) \leq c(u,v)$ *for all* $u, v \in V$.
2. ***Conservation:*** $\sum_{v\in V} f(v,u) = \sum_{v\in V} f(u,v)$ *for all* $u, v \in V$ *except for the source s and sink t.*

The ***value*** *of a flow is the total flow through the system. It is given by*

$$|f| = \sum_{v\in V} f(s,v) - \sum_{v\in V} f(v,s) \tag{3.3}$$

In other words, it's the total flow out of the source minus any flow coming into the source.

Maximum Flow Problem: Given graph G with defined s, t: What is the maximum possible value for $|f|$?

Dealing with "antiparallel edges": Originally we said we couldn't have edges from $v \to u$ if we have one from $u \to v$. We get around this by **transforming** an antiparallel graph to one with no problematic edges:

- For each antiparallel edge (v_1, v_2):
- Split into two edges (v_1, v') and (v', v_2).
- Set capacity of both to be the same as the original.
- End behavior is **exactly identical**.

Multiple Sources and Sinks:

- For m sources and n sinks:
- Create a super source s with edges leading to each s_i (i.e. $(s, s_i) \forall i \in [m]$).
- Make the capacity $c(s, s_i) = \infty$.
- Do the same, with a super sink t leading from each of the n sinks. Should also have infinite capacity.
- End behavior is the same.

3.5.2 Ford-Fulkerson Method

Residual Capacity of Edge: Amount of additional flow allowable through an edge (u, v) given an existing flow $f(u, v)$.

$$c_f(u,v) = c(u,v) - f(u,v) \tag{3.4}$$

Residual Network: Residual network is represented by $G_f = (V, E_f)$. E_f is given by

$$E_f = \{(u,v) \in V^2 : c_f(u,v) > 0\} \tag{3.5}$$

Note that G_f depends on the specific flow f at a given time.

- **Key Point:** Every edge ALSO has a residual edge doubling back on it with $c_f(v, u) = f(u, v)$.

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (u,v) \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{3.6}$$

This is a very clever definition.

Using Residual Network: If we can find a path from source $\rightarrow$ sink in G_f, then we can increase flow along that path in real graph G by the **minimum value in the path in the residual graph**.

Augmenting Path: Is a simple path from source to sink $s \rightarrow t$ in G_f.

Residual Capacity of Augmenting Path p**:** $c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$.

Algorithm 9: Ford Fulkerson Method

```
Input  : Directed flow graph $G$, source $s$, sink $t$
Output: Optimal flow $f$

Initialize flow $f(u,v) \leftarrow 0$ $\forall (u,v) \in E$;
while $\exists$ augmenting path $p$ in residual graph $G_f$ do
    augment flow $f$ along $p$ (see above);
end
```

Importance of 0 **weight** $\rightarrow$ **Non-Existence:** Once a residual is reduced to zero, the edge no longer exists. This is what enables the algorithm to terminate.

Cut of Flow Network: **cut** (S, T) of $G = (V, E)$ partitions $V \to S, T = V - S$. Net flow from $S \to T$ is given by

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \tag{3.7}$$

Capacity of Cut:

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v) \tag{3.8}$$

Minimum Cut: Cut with minimum capacity relative to all cuts of the network.

Why does Ford-Fulkerson Terminate?

Theorem 15 *Maximum Flow Minimum Cut Theorem: For any flow f on graph G, all of these statements are equivalent:*

- *f is a maximum flow of G.*
- *G_f has no augmenting path.*
- *$|f| = c(S,T)$ for some cut (the **minimum cut**).*

Runtime of Ford-Fulkerson: $O(|f_{\max}| \cdot E)$.

- Using Breadth-First Search to search for Augmenting Paths ($O(E)$).
- We have to repeat this, at most, $f_{\max}$ times.

3.5.3 Edmond-Karp's Algorithm

More efficient version of Ford-Fulkerson.

- **Always** choose the augmenting path with **minimum number of edges**.
- Find this by running shortest paths on the graph but using edges of uniform weight 1.

Runtime of Edmond-Karp: $O(VE^2)$ (vast improvement from $O(|f_{\max}|E)$).

3.5.4 Maximum Bipartite Matching

Bipartite Graph Review: You can split vertices into two disjoint sets. All edges must have one end in set 1 and the other in set 2.

Bipartite Matching: Find $M \subseteq E$ for bipartite graph G such that:

$$\forall v \in V \exists \text{ at most one edge from } M \text{incident on } V \tag{3.9}$$

Maximal Matching: You cannot increase the size of M without violating the constraint.

Maximum Matching: Greatest possible $|M|$. This is what we are interested in solving for.

Reduction to Maximum Flow:

1. Introduce **super source** s that connects to each vertex in set 1 with **infinite connectivity** from s outward.
2. Introduce **super sink** t that connects to each vertex in set 2 with **infinite connectivity** in direction of t.
3. Let each connection between two sets have weight 1.
4. Run **Edmond-Karp** Algorithm.

After this process, $|f| = |M_{maximum}|$.

Chapter 4

Trees

4.1 Definitions

Theorem 16 *Tree Definition: A **tree** is a **connected, acyclic, undirected** graph.*

- ***Root node*** *is usually defined.*
- ***Parent node*** *is the 'next node up' going toward the root.*
- ***Child node*** *is one of the 'next node(s) down' going away from the root.*
- ***Binary tree:*** ≤ 2 *children per node.*
- ***Depth of node:*** *Length of path from* **root** $\rightarrow$ **node**.
- ***Height of node:*** *Number of edges on the* **longest path** *from the node* $\rightarrow$ *leaf.*
- ***Complete** k**-ary tree:*** *Every internal node has* k *children and all leaves are at the same depth.*

Theorem 17 *One-for-all Tree Theorem:* ***If one of these is true, they all are:***

- G *is a tree.*
- *Every pair of vertices* $v_1, v_2 \in G$ *is connected by a* ***unique, simple path***.
- G *is disconnected, but becomes disconnected when one edge is removed.*

- *G is connected with $|E| = |V| - 1$.*
- *G is acyclic.*
- *If an edge is added G **becomes cyclic**.*

4.2 Binary Search Trees

Theorem 18 *Binary Search Tree Definition: For any node x of a binary search tree:*

- *If $y \in$ left sub-tree: $y.key \leq x.key$.*
- *If $y \in$ right sub-tree: $y.key \geq x.key$.*

Traversals:

1. **In-Order:** LNR – Actually prints them from least to greatest.
2. **Pre-Order:** NLR
3. **Post-Order:** LRN

4.2.1 Querying Binary Search Trees

We can perform minimum, maximum, successor, predecessor in $O(h)$ time where h is the height of the tree.

Tree Search: Given a node x and value we are looking for k:

1. If x.key is k or x is NIL: return x. *We either found it or it wasn't there.*
2. If x.key $< k$: Run **Tree Search** on the left sub-tree.
3. Otherwise, run **Tree Search** on the right sub-tree.

Iterative Tree Search: Same input/output as above: **Repeat while x.key is not k and NIL**

1. If x.key $< k$: $x = x$.left.
2. Else, $x = x$.right.
3. After break condition above, return x.

Minimum Tree Search: To find the minimum, keep setting $x = x$.left while x.left isn't NIL.

Tree Successor: To find the successor of node x:

1. If x has a right child: Call **Tree Minimum** on x.right.
2. Else:
   (a) $y = x$.parent
   (b) While $y \neq$ NIL and $x == y$.right:
      i. $x = y$
      ii. $y = y.parent$
   (c) return y

Tree Insert: To insert z into BST x, repeatedly compare z to nodes of x, moving right/left according to result of comparison. Eventually, you will add z as a leaf node.

Tree Deletion: To delete node from z from BST:

1. z has **no children** $\rightarrow$ modify z.parent.child = NIL.
2. z has **one child** $\rightarrow$ modify z.parent.child $\leftarrow z$.child.
3. z has **two children**:
   (a) Let y = Tree Successor(z) [must be from right sub-tree of z]
   (b) Convenient case: $y = z$.right. In this case, just replace z with y.
   (c) Replace $z \leftarrow y$, ensuring y keeps its right sub-tree.
   (d) z.right $\rightarrow y$.right.
   (e) z.left $\rightarrow y$.left.

4.3 Red-Black Trees

Theorem 19 *Red Black Tree Definition: A red-black tree is a* ***binary search tree*** *where each node has a "color" of either* ***red or black****.Each BST operation has additional rules that relate to color, with the end results being:*

- *No path from the* ***root*** *to a* ***leaf*** *is more than* ***two times the shortest path****.*
- *The tree is approximately balanced and therefore is efficient.*

Properties of Red-Black Tree

1. *All nodes are either **red or black**.*
2. *The root node is black.*
3. *All leaves are black.*
4. *Red nodes have black children.*
5. *All simple paths from a node to descendant leaves have **the same numberof black nodes**.*

Resultant Properties

1. *$bh(x)$ is the **"black height"** of node x (numberof black nodes from (but not including) x to a leaf.*
2. *Red-black tree with n internal nodes has height $\leq 2\log(n+1)$.*
3. *All leaves are set to $T.nil$ values instead of regular NIL.*

Easy Red-black Tree Operations: These are exactly the same as BST operations. They all run in $O(\log n)$ time, too.

1. Search.
2. Minimum.
3. Maximum.
4. Successor.
5. Predecessor.

Rotations: Rotations swap two nodes while maintaining the BST properties. You can visualize or "feel" it as taking the edge connecting two nodes and physically **twisting** it either right or left for their respective rotation.

Red-Black Insert Fixup: This algorithm reinstates the red-black properties of the tree around a given recently-inserted node z.

Red-Black Insertion: To insert node z...

1. Perform conventional BST insertion of z, setting the new node's children to T.nil instead of NIL.
2. Set z's color to RED.
3. Call **Red-Black Insert Fixup**(T, z).

Red-Black Transplant: Same function as before, just replace 'NIL' with T.nil.

Red-Black Deletion: Deletion is substantially more involved and uses a particular **Red-Black Delete Fixup** function.

Chapter 5

Proof Methods

5.1 Induction

Basic Idea: To prove by induction, we show that the statement holds for some base case $n = 1$, then show that the statement holding for aritrary n implies that the statement holds for $n + 1$. That concludes the proof.

Basis: We prove that the statement holds for some set values of n (usually $n = 1$ or $n = 0, 1, ..., 4$).

Hypothesis: We hypothesize that the thing we are trying to prove is true.

Inductive Step: We "plug in" the $n+1$ case to the hypothesis and show that it boils down to the n case and some algebraic equivalence. Then you can make the claim that it holds for all n.

5.2 Contradiction

Basic Idea: Given a true or false proposition P, we assume that $\neg P$ ("not P") holds. After some clever algebra and symbol-shunting, we get a contradiction. This implies that P must hold instead.

Illustrative Example: P: If $x^2 - 5x + 4 < 0$, then $x > 0$.

- To prove P, we **assume towards a contradiction** (ATaC) that $\neg P$ holds. That is, we assume that if $x^2 - 5x + 4 < 0$ then $x \leq 0$.

- But then:

$$\begin{aligned} x^2 &< 5x - 4 \\ x^2 &< 0 \end{aligned} \tag{5.1}$$

Results in a contradiction! Therefore P must hold.

5.3 Master Theorem

This provides a "hammer" to prove the time complexity of recurrent functions.

Theorem 20 *The Master Theorem: Let $a \geq 1$ and $b \geq 1$ and $f(n)$ be some function.*

$$T(n) = aT(\frac{n}{b}) + f(n) \tag{5.2}$$

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ *for some* $\epsilon > 0$*. Then*

$$T(n) = \Theta(n^{\log_b a})$$

Case 2: $f(n) = \Theta(n^{\log_b a})$*. Then*

$$T(n) = \Theta(n^{\log_b a} \log n)$$

Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ *for some* $\epsilon > 0$ ***AND*** $af(n/b) \leq cf(n)$ *for* $0 < c < 1$*. Then*

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

.

5.4 Substitution

Substitution is a method for determining the **closed from** runtime of an algorithm via induction.

Example – Mergesort: The runtime for mergesort is recursively defined as $T(n) = 2T(\lceil n/2 \rceil) + n$.

1. We start by **guessing** $T(n) = O(n \log n)$.

2. We **hypothesize** that $T(n) = O(n \log n)$ for all cases $\leq n$, meaning that
$$T(n/2) \leq c\lfloor n/2 \rfloor \log\lfloor n/2 \rfloor$$

3. **Inductive step:** We prove that
$$T(n) \leq cn \log n$$
Which is pretty obvious from there.

Chapter 6

Sorting Algorithms

6.1 Heapsort

6.1.1 Heaps

Theorem 21 *Heap Definition: A heap is an* ***array*** *representing a* ***binary tree*** *with the following properties:*

- *For* ***max heap****:* $A[parent(i)] \geq A[i]$.
- *For* ***min heap****:* $A[parent(i)] \leq A[i]$.
- ***Height*** *of tree is* $\Theta(\log n)$ *(i.e. number of edges to get from root to leaf node).*

Parent-child relationships are as follows:

- $Parent(i) = \lfloor i/2 \rfloor$ *(Parent node of i).*
- $Left(i) = 2i$ *(Left child).*
- $Right(i) = 2i + 1$ *(Right child).*

6.1.2 Heap Algorithms

Max-Heapify is an algorithm that sorts array A into a heap at index i, assuming that right and left sub-trees satisfy the heap property at i.

Algorithm 10: Max Heapify

```
Input  : Array $A$, index $i$. Left and right sub-trees of $i$ should
         already satisfy heap property.
Output: $A$ is sorted in place to satisfy heap property.

$l$ = left($i$);
$r$ = right($i$);
Let largest = max($l, r, i$).
if $i \neq$ largest then
    Exchange $i$, largest;
    MaxHeapify($A$, largest);
else

end
```

Algorithm Complexity: $O(1)$ Space complexity (sorts in place), $O(\log n)$ time complexity.

Build Heap is an algorithm that sorts a completely unsorted array A into a heap.

Algorithm 11: Build Heap

```
Input  : Unsorted list $A$
Output: Heap $A$

for $i = \lfloor |A|/2 \rfloor : 1$ do
    MaxHeapify($A, i$);
end
```

Algorithm Complexity: $O(1)$ Space Complexity, $O(n)$ time complexity.

Heap Sort is a sorting algorithm that runs in $O(n \log n)$ time.

Algorithm 12: Heap Sort Algorithm

```
Input  : Unsorted array $A$
Output: Sorted array $A$

for $i = |A| : 2$ do
    exchange $A[i]$, $A[1]$;
    $A$.heapsize -= 1;
    MaxHeapify($A, 1$);
end
```

Algorithm Complexity: $O(1)$ Space Complexity, $O(n)$ time complexity.

6.2 Quicksort

Key Facts: Quick sort sorts in $\Theta(n^2)$ in the worst case and $\Theta(n \lg n)$ in the average case. The factors on the average case are small, so it ends up being very practical.

Algorithm Description: Given algorithm A, we:

1. Divide A into two **partitions**. We usually select the last element as the partition.
2. We call the partition function to put all values less than or equal to the pivot before the pivot and all values greater than the pivot after the pivot in $\Theta(n)$ time.
3. Then we call the partition function on the two remaining partitions recursively.
4. We end with a sorted list.

Algorithm 13: Partition Subroutine for Quicksort ($\Theta(n)$)

```
Input  : Array $A$, partition start index $r$, partition end index $r$.
Output: $A$ partitioned about $A[r]$, index of partition returned.

$x = A[r]$ $i = p - 1$ for $j = p : r - 1$ do
    if $A[j] \leq x$ then
        i = i+1 exchange $A[i], A[j]$
    end
end
exchange $A[i + 1], A[r]$ return i+1
```

Algorithm 14: Quicksort Algorithm.

```
Input  : Unsorted $A$, partition start index $p$, end index $r$
Output: Sorted $A$

if $p < r$ then
    q = Partition$(A, p, r)$;
    Quicksort$(A, p, q - 1)$;
    Quicksort$(A, q + 1, r)$;
end
```

Algorithm Complexity: $\Theta(n^2)$ worst case, $O(n \log n)$ average case with small coefficients.

6.3 Linear Time Sort

6.3.1 Counting Sort

Key Idea: If we know that the input are integers and can be used as the address in an array, we can go through one-by-one and count the number of each possible element. This requires a lot of space, though, if the range of integers is high.

```
Algorithm 15: Counting Sort Algorithm.
  Input  : Input array $A$, output array $B$, upper bound on integers $k$
  Output:

  $C$ = array of zeros of size $k+1$;
  for $j = 1 : |A|$ do
      $C[A[j]]$++;
  end
  for $i = 1 : k$ do
      $C[i]$ += $C[i] + C[i-1]$;
  end
  for $j = |A| : 1$ do
      $B[C[A_j]] = A[j]$;
      $C[A_j]$–;
  end
```

Time Complexity: $\Theta(k+n)$.

Stability: The order of the entries with the same values is maintained from input to output. Relevant for **radix sort**, and for when “satellite” data is carried with “keys” used in sort.

6.3.2 Radix Sort

Algorithm Idea: Start by sorting based on least significant digit then the most. This, again, only works for integer sorting or similar. We make use of a **stable sort** (usually **counting sort**).

```
Algorithm 16: Radix Sorting Algorithm.
  Input  : Integer array $A$, number of digits in each number $d$
  Output: Sorted array $A$

  for $i = 1 : d$ do
      StableSort $A$ based on digit $i$;
  end
```

Algorithm Complexity: $O(d(n+k))$ time complexity, assuming that StableSort is $O(n+k)$.

For b-bit numbers and any $r \in [0, b]$:

$$\text{Radix Sort} = \Theta((\frac{b}{r}(n + 2^r)))$$

6.4 Order Statistics

Theorem 22 *Order Statistic Definition: The* ***Order Statistic*** *of set A is the ith smallest element of the set. So,*

- $\max A$ *is the nth order statistic of A.*
- $\min A$ *is the 1st order statistic of A.*
- *Median A is the $\lfloor \frac{n+1}{2} \rfloor$ order statistic.*

6.4.1 Minimum and Maximum Algorithms

Minimum and maximum algorithms just iterate through to find min or max.

Finding minimum and maximum efficiently: Iterate through, but compare **next two elements** at a time. Compare the larger element to the current maximum and the smaller of the two to the current minimum.

- **Even length array:** $3n/2 - 2$ comparisons.
- **Odd length array:** $3\lfloor n/2 \rfloor$ comparisons.

6.4.2 Selection in Average Linear Time

Outcome: It is possible to find the ith smallest element in average case $O(n)$ time (worst case $\Theta(n^2)$).

Algorithm 17: Randomized Select Algorithm

```
Input  : Array $A$, partition bounds $p, r$, desired order statistic $i$
Output: The $i$th order statistic of $A$

if $p = r$ then
    return $A[p]$;
end
$q$ = RandomizedPartition($A, p, r$);
$k = q - p + 1$;
if $i = k$ then
    return $A[q]$;
else
    $i < k$
end
return RandomizedSelect($A, p, q - 1, i$);
return RandomizedSelect($A, q + 1, r, i - k$);
```

6.4.3 Selection in Worst Case Linear Time

Outcome: This algorithm can select the ith order statistic in $O(n)$ time.

Algorithm:

1. Partition A into $\lfloor n/5 \rfloor$ groups of 5 and one "extras" group.
2. Find median of each group of find using **insertion sort**.
3. Let $x =$ **Select**(medians, $\lfloor numberofgroups/2 \rfloor$).
4. Let $k =$ **Partition**$(A, 0, |A|, \text{pivot} = x)$. k is now the number of elements larger than x.
5. If $i = k$ then return x. Else, if $i < k$ then return **Select**$(A[0 : k], i)$. Otherwise, return **Select**(A[k:],i-k).

By substitution, we can show that the algorithm runs in $O(n)$.

Chapter 7

Hash Tables

Perform well for tasks like **insert, search, and delete**. $\Theta(n)$ worst case for search, $O(1)$ is the practical average.

7.1 Direct Addressing

- Each element we want to store has a **key** k in **universe** U.
- $|U| = m$, $U = \{0, 1, ..., m-1\}$.
- $T[0, ..., m-1]$ is the **direct address table**.

Direct Address Search (T, k): return $T[k]$.

Direct Address Insert (T, x): $T[x.\text{key}] = x$.

Direct Address Delete (T, x): $T[x.\text{key}] = \text{NIL}$.
The problem arises when the universe U gets large. This happens when you have a ton of unique elements.

7.2 Hash tables

Theorem 23 *Hash Function Definition: A hash function h maps the latent universe of keys U to a universe of some specific size m. That is,*

$$h : U \to \{0, ..., m-1\} \tag{7.1}$$

Collision Resoution with Chaining: Just add to a linked list.

1. **Insertion:** x is added to the **head** of the linked list.
2. **Search:** Search for element in the linked list that it hashes to.
3. **Delete:** Delete from linked list as usual.

Performance: If the number of elements $n = O(m)$, you'll get, on average, $O(1)$ performance.

Simple Uniform Hashing: Each element has same probability of hashing to given value.

7.3 Hash Functions

Here are a few types of commonly-used hash functions. Note that most datatypes (strings, characters, etc.) can be converted to natural numbers by place value. They just might be very large.

7.3.1 Divisision Method

To map key k to one of m slots, we can let

$$h(k) = k\%m \tag{7.2}$$

Considerations:

- m should not be 2^n because then it'd just extract bottom n bits.
- Usually good to pick prime number m.

7.3.2 Multiplication Method

We multiply k by some $A \in [0, 1]$ then use division method.

$$h(k) = \lfloor m(kA\%1) \rfloor \tag{7.3}$$

Considerations:

- Value of m does not matter any more (2^n makes for easy hardware implementation.
- $A \approx \frac{\sqrt{5}-1}{2}$ tends to work well.

7.3.3 Open Addressing

The key idea is to let $h()$ take in a "probe number" i. We increment this number to make sure hash table gets filled without doubling up on a single slot.

- **Probe sequence** of k is $\langle h(k,0), ..., h(k,m-1)\rangle$.
- The probe sequence for each k is a **permutation of** $\langle 0, ..., m-1\rangle$.
- Because of this property, we can always fill up the table perfectly.

Insertion: Probe table, incrementing i until you reach an empty slot. Add the element there.

Search: Keep probing until you either find the element **or you find NIL**.

Deletion: When you find value, set it to DELETED – NIL would stop searches that would have to traverse.

Performance of Open Addressing:

- If $n/m \leq 1$: The expected number of probes for an **unsuccessful** search is
$$\gamma \leq \frac{1}{1-\frac{n}{m}}$$
- Inserting to open address table also takes γ probes on average.
- Number of probes for a successful search is
$$\leq \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$
Where $\alpha = n/m \leq 1$.

Linear Probing: If $h'(k)$ is single-variable hash function, we can make a probing hash function via:

$$h_{lin}(k,i) = (h'(k)+i)\%m \tag{7.4}$$

Drawback: Long strings of filled slots build up one after another.

Quadratic Probing: $h'(k)$ is a single-variable hash function:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2)\%m \tag{7.5}$$

Where $c_1, c_2 \geq 0$, $i \in [m-1]$.

- Better performance than linear.
- We still struggle with **secondary clustering** – if $h'(k_1) = h'(k_2)$, we still have the same hash sequences.

Double Hashing

$$h(k,i) = (h_1(k) + ih_2(k))\%m) \tag{7.6}$$

Now probe sequences are highly unlikely to overlap for different keys. **CONDITIONS:**

- $h_2(k)$ should be "relatively prime" to m. The entire should be searched by multiples of each h_2.
- **Method 1:** $m = 2^x$, make sure $h_2(k)$ maps to an odd number.
- **Method 2:** m is prime, make sure $h_2(k)$ maps to positive inteer less than m.

Chapter 8

Dynamic Programming

Motivation: We often solve the same sub-problem many times, particularly during recursive optimization algorithms.

Solution: Memorize the intermediate optimization answers.

8.1 Elements of Dynamic Programming

1. Optimal Substructure.
2. Overlapping sub-problems.

Optimal sub-structure: Optimal solution is $f(\text{input}, \text{optimal solutions to sub-problem})$. A common pattern is:

1. Make some choice (hopefully optimal choice).
2. Realize you need to solve basically the same problem (but smaller).
3. Observer recurrence.
4. Prove **by contradiction** that optimal sub-solution leads to the optimal solution.

Tips and Tricks

- Keep sub-problem space as simple as possible.
- Think through sequential alternatives you might try to make things countable.

- You can try to create a sub-problem graph, too.
- **PRACTICE**.

8.2 Examples

High-level descriptions of dynamic programming algorithms. It seems that most problems are variations on a smaller set of dynamic programming problems.

8.2.1 Rod Cutting

Given: A rod of length n and a vector $p = [15, 20, 31, ..., 100]$ representing selling price for different lengths of rod.

Wanted: Maximum price to sell the rod for given optimal cutting.

Dynamic Programming Solution:

- The first step is to determine where to place your first cut.
- To decide between a 1-unit long cut and a $p[10]$ unit long cut, you would want to know the reward from each possible cut AND the maximum possible reward for what remains.
- Therefore,

$$\text{optim}(n, p) = \max\{p_n, p_{n-1} + \text{optim}(1, p), ..., p_1 + \text{optim}(n-1, p)\} \quad (8.1)$$

- With a general recursive strategy, we would end up calculating the same optim(n, p) many time.
- Therefore, as soon as we calculate some optim(n, p), we store it in a hash table (called **memoization**.

Chapter 9

Greedy Algorithms

9.1 Introduction

The essence of a greedy algorithm is to select the **locally optimal** decision at every opportunity. It is generally efficient, and when it is optimal, is a very convenient choice. Greedy algorithms approximate NP-complete problems well.

Chapter 10

Amortized Analysis

What is amortized analysis?

- We are often interested in the **aggregate** cost over a sequence of operations.
- There are often connections between the number of one operation and another.
  - For instance, you can only pop as many items as you have pushed to a stack.
- Amortized analysis guarantees the **average performance for the worst case** via *deterministic analysis*.

10.1 Aggregate Method

1. Define an $O(1)$ operation for the problem.
2. Observe the worst-case number of operations for n executions of a process.
3. Divide the cost for n executions by n to get amortized cost.

Example: Stack

- Push and pop are $O(1)$.
- Multipop is $O(n)$ – this is when you pop every element.
- Therefore, for n operations, we have at most $n \cdot O(n)$, or n multipops.

- **Aggregate:**
  - We cannot pop more items than have been pushed.
  - The maximum number of items pushed is n.
  - Therefore the average worst case cost for n operations is $O(n)$, meaning that we have average $O(1)$ for each operation.

10.2 Accounting Method

Central Ideas:

- We charge some number of **dollars** for each operation. This is the **amortized cost**.
- The datastructure gets those **dollars** to complete the operation.
- We define some "real cost" of each operation. The exact dollar value doesn't matter, the important thing is **whether it is a function of n or just constant**.
- If our imposed **amortized cost** is more than the "real cost", the data structure gets **credit**.
  - Else, the data structure **uses credit** it has stored to complete the operation.
  - **NEGATIVE BALANCE MEANS A WRONG CREDIT ALLOCATION**.

10.3 Splay Trees

Motivation: We need efficient insertion, deletion, search, and sorting.

- If key access frequencies are known, we can use **dynamic programming**.
- if the key frequencies are unknown, we use **splay trees**.

Splay trees are theoretically optimal in the amortized case.

Key Idea: We simply use a conventional BST, but introduce a **splay** operation.

- An unbalanced tree is **credit rich**.
- A balanced tree is not credit rich.
- Credit is spent to make the tree more balanced, generally.

Algorithm 18: Splay Algorithm

```
Input  : Node $x$ of splay tree.
Output: Tree is splayed (re-balanced) in place.

while $x \neq root$ do
    if $P(x) = root$ then
        rotate $P(x)$;
    end
    if $P(x)$ and $x$ are both (left or right) children then
        rotate $P(P(x))$;
        rotate $P(x)$;
    else
        rotate $P(x)$;
        rotate new $P(x)$;
    end
end
```

Cost of Splay:

- $wt(x)$ is defined as the **number of nodes** in the sub-tree rooted at x (descendants of x including x).
- $\text{rank}(x) = \lceil \log wt(x) \rceil$ is the **height** of sub-tree rooted at x.
- **Credit Invariant:** Node x has \$ rank(x) in credit.
- Therefore, each {zig, zig-zig, zig-zag} costs

$$3[\text{new rank}(x) - \text{old rank}(x)]$$

- The amortized cost of **splay** is therefore

$$O(\log n)$$

Splay Tree Operations:

1. **Search:**
   - (a) BST Search for x.
   - (b) **Splay**(x).
   - (c) *Cost:* $O(\log n)$
2. **Insert:**
   - (a) BST Insert x (x is now a leaf).
   - (b) **Splay**(x).
   - (c) *Cost:* $O(\log n)$
3. **Delete**
   - (a) BST Delete x.
   - (b) **Splay**$(P(x))$.
   - (c) *Cost:* $O(\log n)$
4. **Split:** If you want to create two new sub-BST's, one with all values less than x and one with values greater than x,
   - (a) Perform **Splay Search**(x) – x is now the root.
   - (b) Split off the right and left trees.
   - (c) *Cost:* $O(\log n)$
5. **Join:** Basically the reverse operation of **split**. Given two splay trees, one with values $\leq x$ and one with values $> x$,
   - (a) Make the $\leq x$ tree the left child of x, the $> x$ the right child of x.
   - (b) **Credit the tree** $O(\log n)$ dollars so that x satisfies the credit invariant.
   - (c) *Cost:* $O(\log n)$

Chapter 11

Theory of Computation and Automata

11.1 Definitions and Preliminaries

11.1.1 Framing

Course Focus: Decision problems (i.e. problems with "yes" and "no") answers. E.g. 'does this graph have a clique of size 5 or more?"

Measure of Difficulty: A problem is more difficult if it takes **more computational steps** for a machine to solve. The key factor is whether it takes **polynomial time** or **non-polynomial time** (e.g., exponential time).

Motivation: We utilize an extra layer of abstraction to answer questions about **classes of problems** (e.g. shortest paths, finding cliques, etc.). It is infeasible to go through the (literally infinite) potential **instances** of a problem class.

11.1.2 Definitions

Alphabet: Finite, ordered set of symbols used to convey meaning. All alphabets are **reducible** to the binary alphabet, so we stick to that for the most part.

String: A *string over alphabet A* is a sequence of symbols from a given alphabet A.

- Strings with no characters are represented by ε.

- Union between alphabets is possible, but you need to define some order.

Concatenation: We represent concatenation of string α and β as $\alpha\beta$.

- α^i means concatenating i copies of string α.
- $\alpha^0 = \varepsilon$.

Set of all Strings: Σ^* is the set of all possible strings over alphabet Σ.

Reverse: α^R is the reverse string of α.

Theorem 24 *Definition of a Language: A language is a* ***set of strings*** *over an alphabet. L is a* ***language*** *over alphabet Σ if*

$$L \subseteq \Sigma^* \tag{11.1}$$

Each element of L is a ***string over the language****.*

Operations on Languages:

1. *$L_1 \cup L_2$: Union of two languages*
2. *$L_1 \cap L_2$: Intersection of two languages*
3. *$\bar{L}$: Compliment of language L.*
4. *$L_1 - L_2$: Strings in L_1 not contained in L_2.*
5. *L^i: Concatenating i copies of* ***each string*** *in language L.*
6. *$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$:* ***Kleene Closure*** *of L – taking all possible concatenations of strings in the language.*

We think of languages as **problems**. The decision problem for language L is: "Does string α belong in L or not?".

11.2 Regular Languages

11.3 Complexity Classes P, NP

Theorem 25 *Complexity Class P Definition: All languages for which there exists an algorithm A that* ***decides*** *L in* ***polynomial time*** *belong to class P.*

Theorem 26 *Complexity Class NP Definition: NP is the set of languages L where:*

- *There exists a* ***polynomial time*** *algorithm A and constant c*
- *such that, given a "certificate" (****candidate solution****) y of size* $|y| = O(|x|^c)$
- $A(x, y) = 1$.
- *That is, A can* ***verify*** *a candidate solution for an instance of the problem in polynomial time with respect to the size of the initial problem.*

Theorem 27 *Relationship between class P and NP*

$$P \subseteq NP \tag{11.2}$$

Therefore, if we have a polynomial-time algorithm A that can decide L, then L is definitely a member of both P and NP.

Theorem 28 *P is Closed under Compliment If* $L \in P$*, then* $\bar{L} \in P$*. Simply flip the output of the algorithm that can decide L in polynomial time to decide* $\bar{L}$ *in polynomial time.*

Open Questions in the Field:

- $NP = co - NP$?
- $P = NP \cap co - NP$?
- $NP = P$?

Theorem 29 *Reducibility: Language* L_1 *is* ***polynomially reducible*** *to* L_2 *(i.e.* $L_1 \leq_P L_2$*) iff* $\exists$ *polynomial-time algorithm* $f()$ *such that*

$$x \in L_1 \textit{ iff } f(x) \in L_2 \tag{11.3}$$

Theorem 30 *Reducibility implication: If* $L_1 \leq_P L_2$ *and* $L_2 \in P$*, then* $L_1 \in P$*.*

11.4 NP-Completeness

Theorem 31 *NP-Completeness Definition: $L \in NP - Complete$ iff:*

- *$L \in NP$ (L can be decided in polynomial time).*
- *For every $L' \in NP$, $L' \leq_P L$. (Any problem in NP can be transformed into L in polynomial time).*

11.4.1 Methodology to Prove NP-Completeness

Given language L, we show it is NP-Complete by:

1. Prove that the language can be verified in polynomial time ($L \in NP$) *(5/15 marks, easy).*
2. Select known $L' \in NPC$.
   (a) Describe algorithm $f()$ that converts instances of L' into instances of L such that
   $$[x \in L'] \leq_P [f(x) \in L] \tag{11.4}$$
   (b) Prove that $x \in L'$ iff $f(x) \in L$ (be sure to prove this both ways).
   (c) Show that $f()$ takes polynomial time to solve. Usually, this means saying "we introduce polynomial objects for L for each object in L'.".