CS170 Complete Study Guide

Kanyes Thaker

October 2018

This document is a summary of **Algorithms** by Dasgupta, Papadimitriou, and Vazirani, and as such covers the material taught in UC Berkeley's COMPSCI 170 course. It is by no means a replacement for watching lectures, attending discussion, or preparing for midterms. It should, however, be a good enough summary to grasp the key concepts for the course and be a good way to catch up after falling behind. I'm sure there are several errors in this document, so feel free to contact me if you find anything wrong!

0.1 Big-O Notation

Let $f(n)$ and $g(n)$ be functions mapping the (positive) integers to the (positive) reals. Then:

1. $f = \mathrm{O}(g)$ if there exists a $c \in \mathbb{R}_+$ such that $f(n) \leq cg(n)$. We say f grows **at most** as fast as g, or that f is **upper bounded** by g.
2. $f = \Omega(g)$ if there exists a $c \in \mathbb{R}_+$ such that $f(n) \geq cg(n)$. We say that f grows **at least** as fast as g, or that f is **lower bounded** by g.
3. $f = \Theta(g)$ if $f = \mathrm{O}(g)$ **and** $f = \Omega(g)$. f is **tightly bounded** by g.

For asymptotic analysis, multiplicative constants can be omitted, i.e. $14n^2$ becomes n^2. Polynomial-time runtimes are determined by the highest degree of the polynomial, i.e. $\mathrm{O}(n^a + n^b) = \mathrm{O}(n^a)$ if $a > b$. Every exponential function dominates every polynomial function, and any polynomial dominates every logarithm.

Contents

1 Divide-and-conquer Algorithms

We begin with the **divide-and-conquer** approach for solving problems. We take a big problem and see how we can break it down into smaller problems (of the same type). The key here is that the smaller problems (called **subproblems**) have to be smaller instances of the big problem. The actual algorithm has three steps: we figure out a method for splitting up the big problem, determine what to do when we reach the smallest possible subproblem, and then figure out how to glue our results back together.

1.1 Recurrence relations

In a divide-and-conquer algorithm, we take in a problem of size n, and split it into a subproblems that have size n/b. We then combine those problems in $\mathrm{O}(n^d)$ time. This lends itself to the following **recurrence relation:**

$$T(n) = aT(\lceil n/b \rceil) + \mathrm{O}(n^d).$$

Thankfully, we have a theorem that lets us evaluate this equation easily.

Master Theorem

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

Example: Binary Search
The classic example of divide-and-conquer is binary search, where we find a key k in a file containing a sorted list of keys by comparing k with the first $\lfloor k/2 \rfloor$ key, and recursing on either the lower or upper half of the list depending on whether the key is above or below the middle key. The recurrence is $T(n) = T(\lceil n/2 \rceil) + \mathrm{O}(1)$, or $\mathrm{O}(\log n)$ by the Master theorem.

1.2 Merge Sort

The divide-and-conquer approach lends itself nicely to a specific type of problem – **sorting a list**. How do we go about doing this? We realize that if a list is sorted, then every **subset** of that list must also be sorted! In that case, to sort the list, we'll split the list into halves, sort the halves (recursively, of course), and then merge the sorted components.

Pseudocode:

```
Input: An array of numbers a[1...n]
Output: A sorted version of a

procedure mergesort(a[1...n]):
    if n > 1:
        return merge(mergesort(a[1...n/2]),
                     mergesort(a[n/2+1...n]))
    else:
        return a

Input: Two sorted lists x[1...k], y[1...l]
Output: A merged version of those two lists

procedure merge(x[1...k], y[1...l]):
    if k=0: return y[1...l]
    if l=0: return x[1...k]
    if x[1] <= y[1]:
        return x[1] + merge(x[2...k], y[1...l])
    else:
        return y[1] + merge(x[1...k], y[2...l])
```

Since merges are linear, the recurrence relation for mergesort is $T(n) = 2T(\lceil n/2 \rceil) + \mathrm{O}(n) = \mathrm{O}(n \log n)$ by the Master theorem.

1.3 Medians

The median is the "middle" element of a set, where half the elements are lower in value and half the elements are larger. We **could** find the median of a set by sorting, but this take $\mathrm{O}(n \log n)$ time (which is way more work than we need to do! We only care about the median element, not whether or not the other elements are sorted). Sometimes when we're working with divide-and-conquer algorithms, it's easier to start with a **general** version of the problem. Here instead of looking for the median, we generalize to find the kth smallest element of the list (can you see why this is a more general form of the median problem?)

Given a list S of values, and some value v, we can split the list into 3 subsets – S_L, the list of values **less than** v, S_R, the list of values **greater than** v, and S_v, the list of values **equal to** v. This lets us come to an interesting realization: Say that $|S_L| = 2$, $|S_v| = 2$, and $|S_R| = 5$. Then we know **for a fact** that the 6th smallest element of the list must be the **2nd smallest element in** S_R! We've effectively reduced the difficulty of the problem.

kth Smallest Element

$$\text{selection}(S,k) = \begin{cases} \text{selection}(S_L,k) & \text{if } k \le |S_L| \\ v & \text{if } |S_L| < k \le |S_L| + |S_v| \\ \text{selection}(S_R, k - |S_L| + |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

If we always split our sublists **ideally**, i.e. right down the middle, our recurrence relation would be $T(n) = T(n/2) + \mathrm{O}(n)$. However, this isn't possible – splitting the sublists ideally each time would require knowing exactly what the median is! Instead, we pick v randomly. If we're really unlucky and consistently pick the highest or lowest elements of the list, our running time is $\Theta(n^2)$. However, this is pretty unlikely. We consider an input **good** if it's between the 25th and 75th percentile, which happens with probability $1/2$. Our new recurrence relation is then $T(n) \le T(3n/4) + \mathrm{O}(n)$, which is $\mathrm{O}(n)$ by the Master theorem.

1.4 Matrix Multiplication

The product of matrices $X, Y \in \mathbb{R}^{n \times n}$ is

$$Z_{i,j} = \sum_{k=1}^{n} X_{i,k} Y_{k,j}.$$

This is clearly in $\mathrm{O}(n^3)$. There is a divide-and-conquer approach that is significantly more efficient, however. We can break any $n \times n$ matrix into 4 $n/2 \times n/2$ matrices:

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

We then define the following:

$$\begin{aligned} P_1 &= A(F - H) \\ P_2 &= (A + B)H \\ P_3 &= (C + D)E \\ P_4 &= D(G - E) \\ P_5 &= (A + D)(E + H) \\ P_6 &= (B - D)(G + H) \\ P_7 &= (A - C)(E + F), \end{aligned}$$

then

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

which is $O(n^{\log_2 7})$ (this is less something you need to memorize and more something that's an interesting example of how we can find clever ways of speeding up problems that don't seem like they can be sped up).

1.5 The Fast Fourier Transform

So far we've seen how to multiply matrices. This begs the question – is there a way to efficiently multiply degree d polynomials? Let's define two polynomials and their product as follows:

$$A(x) = \sum_{i=0}^{d} a_i x^i \quad B(x) = \sum_{i=0}^{d} b_i x^i,$$

$$C(x) = A(x) \cdot B(x) = \sum_{i=0}^{2d} c_i x^i, \quad c_i = \sum_{j=0}^{i} a_j b_{i-j}.$$

This is obviously $\mathrm{O}(d^2)$.

We can describe polynomials in **another** way though. Remember that a degree d polynomial is uniquely characterized by its values at $d+1$ points. We can just multiply the $d+1$ values characterizing A and the $d+1$ values characterizing B, and get a polynomial describing C, which we can then interpolate to get the polynomial for C.

Now we have a strategy that we'd like to try. The next question is how to pick n points at which we evaluate the polynomial. We're trying to minimize computation, so we take advantage of the fact that for all even powers of x_i, $A(x_i) = A(-x_i)$ since $x_i^2 = (-x_i)^2$. We let A_e be the polynomial of degree $n/2-1$ with coefficients corresponding to the **even powers of** A. We let A_o be the polynomial of degree $n/2-1$ with coefficients corresponding to the **odd powers of** A. Then

$$A(x) = A_e(x^2) + xA_o(x^2).$$

Write down an example and make sure you understand why this makes sense. Since (as we previously established) we know that $A(x_i) = A(-x_i)$, we can evaluate $A(x)$ at n points by evaluating A_e and A_o at just $n/2$ points!

Our recurrence relation is then $T(n) = 2T(n/2) + O(n) = O(n \log n)$ by the Master theorem.

We can expand this to deeper layers of recursion by using complex numbers to allow for the fact that some of our squares will be negative, using the n^{th} roots of unity $1, \omega, \omega^2, ..., \omega^{n-1}$, where $\omega = \exp 2\pi i/n$. You have seen examples of the Fourier transform in classes like EE16B, and can see more examples of this algorithm in EE120. This transform is **invertible**, i.e. there is an inverse function converts from our "Fourier-ified" polynomial back to our "normal" polynomial. In more formal terms, we project our coefficients onto the Fourier basis, multiply the appropriate values, and use the inverse FFT to recover our coefficients (interpolation). The actual pseudocode for the FFT is omitted here (since it likely won't show up in the class). The important thing to understand is the intuition behind **how** we're splitting up these polynomials.

2 Decompositions of Graphs

One of the most important classes of problems we solve in computer science deals with **graphs**, mathematical objects that represent connections nad networks using **vertices** (representing objects) and **edges** (representing the relationships between those objects). We denote a graph with vertex set V and edge set E as $G(V, E)$. For each $e \in E = \{u, v\}$ $(u, v \in V)$, if $\{v, u\} \in E$ as well, then the graph is called **undirected**. If the relationship between two vertices is **not** symmetric, the graph is **directed**.

We typically represent a graph using an **adjacency matrix**, an $n \times n$ matrix where $A_{ij} = 1$ if $\{i, j\} \in E$ for $i, j \in V$ and 0 otherwise. Accessing an edge is a constant time operation, but the matrix has a space complexity of $O(n^2)$, which is poor for sparse graphs. We can also use an **adjacency list**, a set of linked lists corresponding to each vertex containing all the vertices that vertex is connected to. Then the whole data structure is $O(|E|)$, but accessing an element is no longer constant time.

2.1 Depth-First Search in Undirected Graphs

Question: What parts of the graph are reachable from a given vertex?

We can think of this problem as finding a path through a maze. Just like we can only see the paths in front of us until they hit a corner we can't see around, a computer can only get information about what vertices are immediately reachable from another vertex.

How would we walk through this maze? The simplest way is using a **chalk-and-string** method. We pick a path and go deep into the maze, unrolling a string behind us. When we reach a dead end, we follow the string back until the last "junction", and choose a different path; once all paths have been exhausted, we move to the previous "junction" etc.

Pseudocode:

```
Input: G(V, E), a graph, with v in V
Output: visited(u) is set to True for all u reachable from v
procedure explore(G, v):
    visited(v) = True
    previsit(v)
    for each edge (u, v) in E:
        if not visited(u): explore(u)
    postvisit(v)
```

The issue with EXPLORE is that we need to start it at every vertex (since

not every vertex is necessarily reachable from every other vertex). Note that we still only visit each vertex once; we maintain a VISITED array which keeps track of what vertices we've seen.

Pseudocode:

```
Inputs: G(V, E), a graph
Outputs: None
procedure dfs(G):
    for all v in V:
        visited(v) = False
    for all v in V:
        if not visited(v): explore(v)
```

Overall, this algorithm has a running time of $O(|V| + |E|)$, since each vertex is only explored once and each edge is explored twice.

A graph is **connected** if there is a path between any pair of vertices, i.e. DFS results in a single **search tree** (i.e. our chalk-and-string method hits every possible part of the maze). Say DFS results in multiple search trees. Each of these regions is then called a **connected component**. EXPLORE finds the entire connected component corresponding to vertex v.

We can get a bit more information still about how DFS visits vertices. The **previsit** number of a vertex corresponds to the order in which a vertex is first reached, while the **postvisit** corresponds to the order in which a vertex is last departed.

Property: For any **u** and **v**, the intervals [pre(u), post(u)] and [pre(v), post(v)] are either entirely disjoint or contained within each other.

2.2 Depth-First Search in Directed Graphs

Before moving to this variant of DFS, we must define some terms. A **tree edge** is any edge that was traversed as part of the DFS algorithm. A **forward edge** connect a vertex with a descendant in the DFS tree. A **cross edge** is any other edge (edges that **cross** branches) that connects two vertices that aren't descendants/ancestors.

DFS for directed graphs is identical to DFS for undirected graphs, except instead of exploring all neighbors of a vertex, we only explore along edges that **leave** our initial vertex.

Pre and Postvisit Orderings of Edges

Forward and Tree edges: $\text{pre}(\mathbf{u}) < \text{pre}(\mathbf{v}) < \text{post}(\mathbf{v}) < \text{post}(\mathbf{u})$.
Back edges: $\text{pre}(\mathbf{v}) < \text{pre}(\mathbf{u}) < \text{post}(\mathbf{u}) < \text{post}(\mathbf{v})$.
Cross edges: $\text{pre}(\mathbf{u}) < \text{post}(\mathbf{u}) < \text{pre}(\mathbf{v}) < \text{post}(\mathbf{v})$.

A graph has a **cycle** if there exists some path in the graph from v to v. We can find whether a graph has a cycle just by using DFS. **Directed acyclic graphs**, or **DAGs**, are a good way to model temporal, causal, or hierarchical relationships. All graphs can be **linearized**, meaning that we can "simplify" all graphs so that all edges in the graph point from left to right.

Properties of DAGs

1. In a DAG, every edge leads to a vertex with a lower **post** number.
2. Every DAG has at least one source and at least one sink.
3. A directed graph has a cycle if and only if its depth-first search reveals a back edge.

2.3 Strongly Connected Components

In an undirected graph, our previous definition for **connectivity** (every edge is reachable from every other edge) works just fine. However, when we restrict ourselves to **directed** graphs, we need to re-define our notion of connectivity. In a directed graph, two nodes u and v are connected if there is a path from u to v **and** a path from v to u. Just like we can partition undirected graphs into **connected components**, we can use this definition of connectivity to partition directed graphs into **strongly connected components**.

Each strongly connected component is the set of all vertices that are connected to every other vertex in that component. Naturally, this must mean that two vertices in a directed graph are part of a strongly connected component if and only if they are part of a cycle in the graph. If we represent each SCC by a **meta-node**, we're equivalently removing all cycles from the graph, resulting in a graph that is both **directed** (there are directed edges between connecting the SCCs) and **acyclic** – in other words, **every directed graph is a DAG of its strongly connected components**.

Properties of SCCs

1. If the EXPLORE subroutine is started at **v**, it will terminate only when all nodes reachable from **v** have been explored.
2. The node that receives the highest post number in a DFS must lie in a source SCC.
3. If C and C' are SCCs, and there is an edge from a node in C to a node in C', then the highest post number in C is higher than the highest post number in C'.

How do we find the strongly connected components of a graph? Based on property 1 above, we can conclude that running DFS in a **sink** SCC will discover all of the vertices in that SCC and no more (since DFS can't "leave" a sink SCC). Let the reverse graph G^R be defined as $G(V, E^R)$; in other words, G^R is a graph with the same vertex set as G, except all of its edges are **reversed**, meaning that if $(u, v) \in E$, then $(v, u) \in E^R$. By property 2, running DFS on G^R will reveal a source vertex (let's call it v) in G^R. A source vertex in G^R must be in a sink SCC in G (make sure you understand why). Then running DFS from v in G will reveal the entire sink SCC in G (property 1). We proceed by removing all the vertices in that sink from G and from G^R, and repeat until we've exhasuted the entire graph, at which point we've discovered all the SCCs in the graph.

3 Paths in Graphs

DFS is very good at telling us whether or not a path **exists** between two vertices in the graph; however, it's not very good at telling us what the **shortest** path is between two vertices. For this section, we define the **distance** between two vertices as the length of the shortest path between them.

3.1 Breadth-First Search

Since we're now concerned with minimizing distance, we try something different; we explore vertices in order of distance from the source; only once we've looked at all nodes a distance d from the start do we move on to the $d+1$ nodes. This reasoning is implemented in **breadth-first search** (BFS).

Pseudocode:

```
Input: Graph G = (V, E), vertex s in V
Output: for all u reachable from s,
        dist(u) = the distance from s to u

procedure bfs(G, s):
    for all u in V:
        dist(u) = infinity

    dist(s) = 0
    Q = [s] #queue containing just s
    while Q is not empty:
        u = eject(Q)
        for all edges (u, v) in E:
            if dist(v) = infinity
                inject(Q, v)
                dist(v) = dist(u) + 1
```

Unlike DFS, the BFS tree has the property that all paths from S are as short as possible – it is a **shortest paths tree.** This algorithm runs in $O(|V|+|E|)$ for the same reason as DFS; we add each vertex to the queue exactly once, and the inner loop sees each edge at most twice. DFS explores deeply into the graph, allowing us to search for loops, strongly connected components, etc. BFS, on the other hand, makes shallow explorations into the graph, allowing us to find the shortest path.

BFS treats all edges as having unit length. However, this is often not the case. We now approach a more general problem, where every $e \in E$ has an edge length l_e.

3.2 Dijkstra's Algorithm

Now that we've introduced the concept of **weighted** edges, let's see if we can adapt BFS to search a more general graph, where edge lengths l_e are positive integers.

The first idea that comes to mind is just filling each edge with $l_e - 1$ dummy vertices, such that all the edge lengths become 1 and then using BFS. This approach gets the job done, but it's not very efficient; we don't want BFS wasting time on dummy nodes.

Another strategy is implementing an "alarm clock." When we reach a vertex v from u, an alarm goes off. We set these alarms based on the following algorithm: we check whether or not going from v to u would be slower or faster than the current alarm set for u, with all alarms initialized to be infinite. If it is faster, we reset the alarm for u, and continue until all the alarms have gone off.

3.2.1 Djikstra's Algorithm

We use a **min-priority queue** implemented via a **heap** to keep track of what "alarm" will go off next. The functions of a heap are as follows:

1. **Insert:** Add a new element to the set
2. **Decrease-key:** Accommodate a decrease in the key value for na element
3. **Delete-min:** Return the element with smallest priority and remove it from the set
4. **Make-queue:** Build a PQ out of the given elements and keys

---

Pseudocode:

```
Inputs: Graph G=(V, E)
    positive edge lengths {l_e: e in E}
    vertex s in V
Outputs: For all vertices u reachable from s,
         dist(u) is the shortest path from s to u

procedure djikstra(G, l, s):
    for all u in V:
        dist(u) = infinity
        prev(u) = nil
    dist(s) = 0

    H = makequeue(V)
    while H is not empty:
        u = deletemin(H)
```

```
    for all edges (u, v) in E:
        if dist(v) > dist(u) + l(u, v):
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```

The efficiency of Dijkstra's varies by heap implementation. A naive array implementation gives us $O(|V|^2)$ while a binary heap gives us $O((|V| + |E|) \log |V|)$, and a Fibonacci heap gives us $O((|V| \log |V|) + |E|)$.

3.3 Shortest Paths in the Presence of Negative Edges

Djikstra's algorithm works because the shortest path from s to v must pass through nodes that are closer to v. How do we accommodate negative edge weights? A crucial invariant of Dijkstra's is that the values of DIST are either exactly correct or overestimates. In other words, the shortest distance from s to v that passes through u must be less than the distance from s to u plus l_{uv}. In other words, $dist(v) = \min\{dist(v), dist(u) + l_{uv}\}$.

If we consider Dijkstra's as a series of these updates, we realize that it doesn't actually matter what happens in the rest of the graph – using the above property, we can see that the distance will still be correctly computed! Since we don't know the shortest paths of the graph beforehand, we update **all** the edges $|V|-1$ times. This is known as the Bellman-Ford algorithm, and runs in $O(|V| \cdot |E|)$.

Pseudocode:

```
Inputs: Directed graph G = (V, E)
    edge lengths l = {l_e: e in E} with no negative cycles
    vertex s in V
Outputs: For all vertices u reachable from s, dist(u) is
    the distance from s to u
procedure bellman-ford(G, l, s):
    for all u in V:
        dist(u) = infinity
        prev(u) = nil
    dist(s) = 0
    repeat |V|-1 times:
        for all e in E:
            update(e)

procedure update(e in E):
    dist(v) = min{dist(v), dist(u) + l(u, v)}
```

This algorithm fails with negative cycles. It doesn't even make a lot of sense

to try and determine a shortest path in a graph with a negative cycle, as any path that enters the cycle will have a length of $-\infty$ (do you see why? We'd just be stuck in the negative cycle forever since we're trying to minimize the path length).

Fortunately, it is easy to detect these cycles. Instead of stopping after $|V| - 1$ iterations, we do one extra round. If we update some DIST value after this final round, we know there must be a negative cycle (since if there wasn't, the distances should be at their minimums at the end of $|V| - 1$ iterations).

3.4 Shortest Paths in DAGs

There are only two graphs where the possibility of having negative cycles is impossible. Trivially, they are graphs with no negative edges, and graphs without cycles. The key sources of efficiency is that in any path of a DAG, the vertices appear in increasing linearized order. It is enough to topologically sort the dag by DFS, and visit the vertices in sorted order. We can find the longest path in a DAG the same way, just by negating all edge lengths.

Pseudocode:

```
Inputs: DAG G = (V, E)
    edge lengths {l_e: e in E}
    vertex s in V
Outputs: For all vertices u reachable from s,
         dist(u) is the distance from s to u

procedure shortest-path-dag(G, l, s):
    for all u in V:
        dist(u) = infinity
        prev(u) = nil

    dist(s) = 0
    linearize G
    for each u in V in linearized order:
        for all edges (u, v) in E:
            update(u, v)
```

---

4 Greedy Algorithms

Greedy algorithms solve by problems by performing whichever action gives the most immediate benefit (hence the name *greedy* – they greedily do what gives the greatest immediate return). While greedy algorithms aren't always the best way to solve problems, there are certain cases where they will always yield optimal results.

4.1 Minimum Spanning Trees

Question: Suppose we have a selection of nodes connected by edges. Each edge has some cost associated with it. How do we connect all nodes while minimizing edge cost?

Minimum Spanning Tree

For an undirected graph $G(V, E)$ with w_e as the weight of edge $e \in E$, the **minimum spanning tree** $T(V, E')$, $E' \subseteq E$ is a tree that connects all the vertices in V with edges that minimize

$$\sum_{e \in E'} w_e.$$

Here we review some basic proeprties of undirected graphs:
Property 1: Removing an edge from a cycle from a graph does not disconnect the graph.
Property 2: A tree on n nodes has $n - 1$ edges.
Property 3: Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.
Property 4: An undirected graph is only a tree if there is exactly 1 path between any pair of nodes.

The Cut Property

If the vertices of $G(V, E)$ can be cut into S, $V - S$ then the lightest edge crossing S, $V - S$ must be in **every** MST of G.

4.1.1 Kruskal's Algorithm

Kruskal's Algorithm is one method for finding MSTs. We simply take an empty graph and add the lightest edge in E that does not result in a cycle. We repeat this until every single vertex is part of a single tree.

Pseudocode:

```
Input: Graph G(V, E) with edge weights w_e
```

```
Output: An MST on G, E'

procedure kruskal(G, w):

    for all v in V:
        create {v}
    E' = {}
    Sort E by increasing weight
    for each e = {u, v} in E:
        if u and v are in different sets:
            add e to E'
            merge the sets of u and v
    return E'
```

4.1.2 Data Structures for Disjoint Sets

We can store sets in a directed tree. We've already seen this in CS61B, but here's a brief overview. Nodes are elements, randomly arranged. Each node has a **parent** (sometimes written as $\pi(x)$) and **rank**, the height of that node's subtree.

The **name** or **root** is the topmost node, whose parent is itself and whose height is the height of the entire tree. Two elements are in the same set if their sets have the same root (found through the **find** operation).

Our **find** operation takes time proportional to the height of the tree. We build trees through the **union** operation, where we make the root element of the shorter tree r_{short} point to the root element of the taller tree r_{tall} to give us the best height and therefore fastest **find**.

Pseudocode:

```
procedure union(x, y):
    root_x, root_y := find(x), find(y)
    if root_x = root_y: return
    if rank(root_x) > rank(root_y): parent(root_y) = root_x
    else:
        parent(root_x) = root_y
        rank(root_y) += 1
```

Again, some brief properties of trees:

1. $\forall n$, $rank(\pi(n)) > rank(n)$

2. Any root node of rank k has at least 2^k nodes in its tree

3. If there are n elements overall, there can be at most $\frac{n}{2^k}$ nodes of rank k.

The last property implies that the maximum rank is $\log n$ which provides an upper bound on **find** and **union**.

Path Compression: With the above data structure, the runtime for Kruskal's algorithm is $O(|E| \log |V|)$ for sorting and $O(|E| \log |V|)$ for union and find. Suppose, however, that the edges are given to us in a sorted fashion or the weights are small (so sorting can be done in linear time). Is there a better way to optimize our **find**?

The solution is to modify **find** so that as we pass through parent nodes on the way to the root, we change the parent of each of those nodes to be the root of the entire set. Note that the above properties still hold, but the *rank*s no longer necessarily represent the height of the subtree. The **amortized cost** is then $O(1)$, faster than the earlier $O(\log n)$!

4.1.3 Prim's Algorithm

Prim's Algorithm is another way to find an MST of a graph. Here, we start with an empty graph. From a starting vertex, we then add the lightest edge with one vertex in the growing tree S and one vertex in $\bar{S}$.

This is extremeley similar to Dijkstra's Algorithm, the only difference being the key values by which entries in the priority queue are ordered. The runtime is still $O(|E| \log |V|)$.

Pseudocode:

```
Input: A connected undirected graph G(V, E) with weights w_e
Output: An MST on G, prev

procedure prim(G, w):
    for all v in V:
        cost(v), prev(v) := infty, None
    Pick a random v_0 --> cost(v_0) = 0
    PQ = makequeue(V) #Build a PriorityQueue
                      where the costs are keys
    while PQ is not Empty:
        u = del_min(PQ)
        for each e = {u, z} in E:
            if cost(z) > w_e:
                cost(z), prev(z) = w_e, u
                decrease_key(PQ, z)
    return prev
```

4.2 Huffman Encoding

Say we need to compress a large set of data by encoding each element of that data's alphabet. A naive approach is to evenly split up our encoding – for a 4-value alphabet, we give each letter 2 digits in binary (00, 01, 10, 11). Then we'd say that 00 represents A, 01 represnts B, etc. However, what happens when one letter of that alphabet appears much more frequently than another letter? It would make sense to try to modify our encoding so we can use fewer bits for more frequent letters and more bits for less frequent ones (a **variable-length encoding**).

An issue with variable-length encodings is that our result can be **ambiguous**. How would we be able to tell whether the string "0110" is {0}{110} or {011}{0}? The solution is to make sure our encoding is **prefix-free** (no codeword can be the prefix of another codeword). This way, we'll always know what "word" we're referring to at any point in our decoding process!

We can represent this by using a **full** binary tree (each node has either 0 or 2 children). When we traverse a string, we start at the root and move bit by bit until we reach a leaf node, at which point we return the decoded letter and return to the root for the next bit.

Question: How do we find the optimal coding tree given the frequencies of n symbols? In other words, how do we minimize the cost, where the cost of a tree is the sum of frequencies of all leaves and internal nodes?

We construct this tree **greedily**. Find the two symbols with the lowest frequency (e.g. frequencies f_1, f_2) and make them the children of a new node with frequency $f_1 + f_2$. Remove f_1, f_2 from the list of frequencies, add in $f_1 + f_2$ to the list of frequencies and repeat. This algorithm takes $O(n \log n)$ time with a binary heap.

Pseudocode:

```
Input: An array f[1,...,n] of frequencies
Output: An encoding tree with n leaves
procedure huffman(f):
    H := PriorityQueue of integers, ordered by f
    for i in (1, ..., n): insert(H, i)
    for k=n+1 to 2n-1:
        i = del_min(H), j = del_min(H)
        create a node k with children i, j
```

```
        f[k] = f[i] + f[j]
        insert(H, k)
```

4.3 Horn Formulas

A horn formula is a framework for expressing logical facts and then drawing conclusions. Horn formulas are composed of **boolean variables** with a truth value of either True or False. A **literal** is either a variable x or its negation $\bar{x}$. **Implications** are of the form "$f(x) \implies y$" ("if the logical statement on the LHS is true, then the single RHS literal is true") where both sides only have positive literals. A **singleton** is a degenerate implication "$\implies x$" where x is True. **Pure negative clauses** consist of an OR of only negative literals (i.e. $(\bar{x} \wedge \bar{y} \wedge \bar{z})$).

Question: Given a set of clauses consisting of implications and pure negative clauses, is there an assignment of true and false to the variables that satisfies all the clauses (i.e. is there a satisfying assignment)?

Our solution begins by initializing all variables as false. One by one, we begin to set them true if and only if an implication is violated otherwise. Then once all implications are satisfied, we check if all negative clauses are satisfied.

Pseudocode:

```
Input: A Horn Formula
Output: A satisfying assignment if one exists

def horn_solver(hf):
    set all variables to false
    while there is an implication that is not satisfied:
        set the RHS variable of that implication to True

    if all negative clauses are satisfied: return assignment
    else: return ‘‘no satisfying argument exists’’
```

4.4 Set Cover

Question: In a set B, how do we mark certain $b_i \in B$ such that $\forall\, b_j \in B$, $\exists\, b_i$ within a certain range of b_j?

This problem, known as **set cover**, is extremeley common. To find a greedy approximation, we let $S_i \subseteq B$ be the set of vertices "covered" by b_i (i.e. the set that fall within that "certain range" of b_i). The next step is simple and intuitive. While B is uncovered, pick the S_i which has the greatest number of uncovered elements. However, this solution may not aways be optimal. If B **contains** n **elements with an optimal cover of** k **sets, the greedy approach will use** $\leq k \ln n$ **sets**. There is likely no polynomial-time algorith with a faster **approximation factor**.

SECTIONS BELOW THIS ARE NOT UPDATED

5 Dynamic Programming

So far in this course, we've seen very specific answers to very specific questions (divide-and-conquer, graph traversal, greedy, etc). However, we now turn to more generalized methods of solving complex problems, albeit at the cost of some efficiency.

Typical subproblems:
i. The input is $x_1, ..., x_n$ and the subproblem is $x_1, ..., x_i$, therefore linear.
ii. The input is $x_1, ..., x_n$ and $y_1, ..., y_n$, and the subproblems are $x_1, ..., x_i$ and $y_1, ..., y_j$, therefore O(n^2)
iii. The input is a rooted tree. The subproblem is a rooted subtree.

5.1 Shortest Paths in DAGs

The most special property of DAGs is that they can be **linearized**, i.e. arranged in a line so that all edges point from left to right. This lends itself well to dynamic programming. To find the shortest path to a node N, we only need to find the minimum of {predecessors of N + distance from each predecessor to N}. This allows to find the shortest path in a single pass.

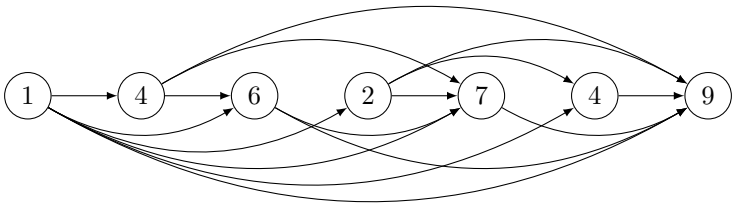
```
initialize all dist() values to infty
dist(s) = 0
for each v in V\{s}, in linearized order:
    dist(v) = min{dist(u) + l(u, v): u:(u, v) in E}
```

This algorithm solves a series of **subproblems**. Dynamic Programming procedures break a complex problem down into a series of subproblems, solving the smallest ones first as a way to solve progressively larger ones. We can think of each DP problem as a DAG – the nodes are subproblems, the edges are the dependencies between subproblems.

5.2 Longest Increasing Subsequence

Question: What is the the length of the longest increasing subsequence in a sequence of numbers, where the numbers are not necessarily adjacent?

This can be easily visualized by forming a linearized DAG out of the elements of the array, with edges connecting numbers that appear in increasing order. For instance, given the list [1, 4, 6, 2, 7, 4, 9] the linearized DAG would be:



Pseudocode:

```
Input: Directed Acyclic Graph G(V, E) with n vertices
Output: length of longest increasing subsequence

procedure longest_increasing(S):
    for j = 1, 2, ..., n:
        L(j) = 1 + max{L(i): (i, j) in E}
    return max{L(j)}
```

$L(j)$ is the length of the longest path ending at vertex j. The essence of dynamic programming is that there is some innate **ordering** to the way the subproblems are constructed, and therefore we must solve ths smaller subproblems and use them to solve the larger ones.

5.3 Edit Distance

Question: What is the minimum number of insertions, deletions, and replacements of characters needed to turn one string into another?

First we determine the subproblems that we need to solve. Instead of comparing all of $x[1, ..., n]$ and $y[1, ..., m]$, let's first see if we can find the edit distance of a prefix, $x[1, ..., i]$ and $y[1, ..., j]$. Let's call this subproblem $E(i, j)$, meaning our end goal is to find $E(n, m)$.

Examine the last character of each prefix. As stated above, there are 3 cases – either we can insert a character from one string into the other or perform a replacement of the two characters. In the first case, we get a cost of 1 and our subproblem becomes $E(i-1, j)$. In the second, our cost is 1 and we must solve $E(i, j-1)$. In the 3rd case, our cost is 1 if $x[i] \neq y[j]$ or 0 otherwise, and we must solve $E(i-1, j-1)$. We've split $E(i, j)$ into 3 smaller problems, but which one is best? We can try them all and find out. diff(i, j) is 1 if $x[i] \neq y[j]$

$$E(i, j) = min\{1 + E(i-1, j), 1 + E(i, j-1), \text{diff}(i, j) + E(i-1, j-1)\}$$

What order should we be solving these subproblems in? It doesn't matter as long as we solve $E(i-1, j), E(i, j-1), E(i-1, j-1)$ before we attempt $E(i, j)$.

Pseudocode:

```
for i in 0, ..., n:
    E(i, 0) = i
for j in 0, ..., m:
    E(0, j) = j
for i in 0, ..., n:
    for j in 0, ..., m:
        E(i, j) = min{E(i-1, j) + 1, E(i, j-1) + 1,
            E(i-1, j-1) + diff(i, j)}
return E(n, m)
```

5.4 Knapsack

Question: A robber wants to complete a heist with the most valuable total haul, but his knapsack has limited weight. Or more generally, say we have a series of tasks with a certain value and cost but we can only handle a certain total cost. How do we maximize our value given the cost of each task? This can be solved in $\mathbf{O}(nW)$ time where W is the weight limit and n is the number of items.

5.4.1 Knapsack with Repeated Items

If we're allowed repeated items, we can formulate our subproblems easily. If $K(w)$ is the maximum value we can get with a max weight of w, note that removing item i from the knapsack will give us the solution to $K(w - w_i)$. Effectively, this becomes

$$K(w) = \max_{i:w_i \leq w}\{K(w - w_i) + v_i\}.$$

Pseudocode:

```
K(0) = 0
for w=1, ..., W:
    K(w) = max{K(w-w_i)+v_i : w_i <= w}
return K(W)
```

5.4.2 Knapsack without Repeated Items

We can't use our subproblems from last time because it's impossible to know what items have already been used. To fix that, we have to keep track of the items being used as well. Let $K(w, j)$ be the maximum value of a knapsack with capacity w and items $1, ..., j$. We're looking for $K(W, n)$. Now, we realize that there are 2 cases: either we use item j or we don't! This is then simply

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_i, K(w, j - 1)\}.$$

Effectively, our reduction is $K(\cdot, j - 1)$.

Pseudocode:

```
for all j:
    K(0, j)=0
for all w:
    K(w, 0) = 0
for j in 1, ..., n:
    for 1 in 1, ..., W:
        if w_j > w: K(w, j) = K(w, j-1)
        else: K(w, j) = max{K(w, j-1), K(w-w_j, j-1) + v_i}
return K(W, n)
```

5.5 Chain Matrix Multiplication

Suppose we want to multiply a series of matrices. While matrix multiplication is not commutative (eg. $\mathbf{AB} \neq \mathbf{BA}$), it is associative (eg. $\mathbf{(AB)C} = \mathbf{A(BC)}$). The number of operations we have to complete in order to multiply a set of matrices **does** vary with our parenthesization of the matrices. A natural greedy approach fails in finding the optimal solution.

Question: How do we determine the optimal parenthesization for matrix multiplication given the dimensions of each matrix?

Immediately, we can recognize that we can represent this process in a binary tree! Leaf nodes are the matrices we are multiplying and the intermediate nodes represent their products. The root of the tree is then the final product! Note that based on typical subproblem (iii) under the section header, we can use DP using the subtrees as subproblems. For $1 \leq i \leq j \leq n$ define

$$C(i, j) = \text{minimum cost of } \prod_{k=i}^{j} A_i.$$

How do we solve the subproblem? Notice that we can split each subtree into its left and right branches, i.e. have one side be $A_i \times ... \times A_k$ and the right half be $A_{k+1} \times ... \times A_j$. The cost of the subtree is then the cost of each branch plus the cost of combining them. The overall computing time is simply $O(n^3)$.

$$C(i,j) = \min_{i \leq k < j}\{C(i,k) + C(k+1,j) + m_{i-1}m_k m_j\}.$$

Pseudocode:

```
for i in 1,...,n:
    C(i, i) = 0
for s in 1, ..., n-1:
    for i in 1, ..., n-s:
        j = i+s
        C(i, j) = min{C(i, k) + C(k+1, j) + m_{i-1}*m_k*m_j
            : i <= k < j}
return C(1, n)
```

5.6 Shortest Paths

5.6.1 Shortest Reliable Paths

Question: Suppose we have a graph $G(V,E)$ with edge lengths $l(u,v)$. What is the shortest path from s to t that uses at most k edges?

Let $\text{dist}(v,i)$, $v \in V$, $i \leq k$ to be the length of the shortest path from s to v with at most i edges. The update equation is then

$$\text{dist}(v,i) = \min_{(u,v)\in E}\{\text{dist}(u,i-1) + l(u,v)\}$$

5.6.2 All-pairs Shortest Paths

Question: Instead of just finding the shortest path from s to t, suppose we wanted the shortest path between all pairs of vertices in the graph.

Number all $v \in V$ as 1, 2, ..., n. Reduce the number of **intermediate nodes** between i and j to be 0. Then, the shortest path between i and j is ∞ if i and j are nonadjacent, or just the $l(i,j)$ otherwise. If we slowly relax our rule by allowing more and more intermediate nodes, we can find a concrete update equation. Let $\text{dist}(i,j,k)$ be the shortest path between i and j where only allow nodes 1, ..., k to be used as intermediate nodes.

$$\text{dist}(i,j,k) = \min\{\text{dist}(i,k,k-1)+\text{dist}(k,j,k-1),\ \text{dist}(i,j,k-1)\}.$$

This takes its form in the Floyd-Warshall algorithm, which solve this problem in $O(|V|^3)$ time.

Pseudocode:

```
for i in 1, ..., n:
    for j in 1, ..., n:
        dist(i, j, 0) = infinity
for all (i, j) in E:
    dist(i, j, 0) = l(i, j)
for k in 1, ..., n:
    for i in 1, ..., n:
        for j in 1, ..., n:
            dist(i, j, k) = min{dist(i, k, k-1)
                + dist(k, j, k-1), dist(i, j, k-1)}
```

5.6.3 The Traveling Salesman Problem

Question: Find the shortest path a salesman can take that visits every city exactly once. In other words, find the minimum weight Hamiltonian cycle in a graph.

This is an NP-complete problem! However, a DP solution will still perform much better than the brute force O($n!$) solution. Let $S \subseteq M$ where M is the total map, i.e. all the cities. Define $C(S, j)$ to be the length of the shortest path in S starting at 1 and ending at j. How do we reduce this further? Note that if we were to backtrack by 1 city (a distance of d_{ij}) , we have a different problem – **what is the optimum city to backtrack to?** This is reflected in the update equation:

$$C(S, j) = \min_{i \in S, i \neq j}\{C(S - \{j\}, i) + d_{ij}\}.$$

Pseudocode:

```
C({1}, 1) = 0
for s in 2, ..., n:
    for all subsets S in {1, 2, ..., n} of size s
            containing vertex 1:
        C(S, 1) = 0
        for all j in S, j != 1:
            C(S, j) = min{C(S-{j}, i) + d_ij : i in S, i != j}
```

There are a total of 2^n subsets, each linear to solve. Therefore the total runtime is O($n^2 2^n$).

5.7 Independent Sets in Trees

An independent set in G is a set of vertices with no edges connecting them. Finding the largest independent set in a general graph is intractable (like

knapsack or TSP) but certain variations on it are solvable!

Question: What is the largest independent set in a tree?

Let $I(u)$ be the largest independent set of the subtree of u. If we include u in the independent set, we can't evaluate its children but we **can** evaluate its grandchildren. Otherwise, we can immediately evaluate its children. The runtime is then just dependent on the number of vertices. Our algorithm is linear, running in $O(|V| + |E|)$.

$$I(u) = \max \left\{ 1 + \sum_{\text{grandchildren } w \text{ of } u} I(w), \sum_{\text{children } w \text{ of } u} I(w) \right\}.$$

6 Linear Programming and Reductions

Linear programming describes the process of maximizing or minimizing the output of some linear objective function given a set of linear constraints on the variables of the objective function.

6.1 Introduction

Say you have a function in $\mathbb{R}^2$. We can easily represent the constraints of the linear program as regions on the 2-d plane. If we take the boundaries of the intersection of all of these regions, we get a set of vertices. The optimum solution must lie at one of these vertices. This method fails if there are mutually exclusive constraints ($x \geq 1, x \leq 2$) or if there are unbounded constraints ($x \in \mathbb{R}^n$). **Dantzig's Simplex Method** starts at a vertex, and then travels to an adjacent vertex only if the value of the objective method is increased by doing so. Once it has reached an "optimum" vertex, it halts. This method can be expanded into n dimensions easily.

6.2 Reductions in LP Problems

A **reduction** is the act of taking a complex problem and showing how it can be restated as a simpler problem we already know the answer to. For example, the longest path in a DAG can be reduced to finding the shortest path in a DAG by negating all edge weights.

In linear programming, there are many degrees of freedom – we can have either a minimization or a maximization problem, constraints can be equations or inequalities, and variables can be unrestricted in sign. However, we can reduce each of these easily. We can switch between min and max problems by multiplying the objective function by -1. We can turn an inequality into an equality by introducing a **slack variable** that equals the disparity between the values on opposite sides of the comparison operator. Turn an equality into an introduction by introducing two tight inequalities ($= \implies \leq \vee \geq$). Replace a negative variable with the difference of two nonnegative variables.

Using these reductions, we can turn any LP problem into **standard form**, with nonnegative variables, all constraints are equations, and the objective function is to be minimized.

6.3 Flows in Networks

A **flow** is a shipping scheme on a graph from source s to sink t that tells us, given the maximum capacity of each edge, how much information we can send through the graph.

6.3.1 Maximizing Flows

We are looking at $G(V, E)$, $s, t \in V$, with capacities $c_e > 0 \forall e \in E$. f_e is the flow on edge e where $0 \leq f_e \leq c_e$ and $\forall u \in V \backslash \{s, t\}$ the amount of flow

leaving u equals the amount entering u. The size of the flow is the amount of information in all flows combined, equivalent to the amount of information leaving s.

This is clearly a linear program and can be solved by simplex extremely quickly! We simply find a path from s to t and gradually increase flow along that path and sum all such possible paths. What if simplex chooses the wrong initial path? We could definitely then reach a case where once we make a path, the next path seems to clash with our already determined one. Simplex avoids this issue by allowing flows to cancel out other flows. In other words, simplex looks for one of two types of edges:

i) (u, v) is in the original network and is not at full capacity.
ii) (v, u) is in the original network and there is some flow through it.

In the first case, if the current flow through (u, v) is f_{uv}, it can still handle $c_{uv} - f_{uv}$ flow. In the second case, we can cancel out all or some of the flow, giving us up to f_{uv} flow to work with. These two types of edges allow us to draw the **residual graph** G^f whose edges are the types of edges listed, with residual capacities $c^f_{uv} = c_{uv} - f_{uv}$ if the first condition is satisfied and $c^f_{uv} = f_{uv}$ if the second condition is satisfied. **We can think of simplex as choosing $s - t$ paths in G^f to find the max flow**. The solution generated by simplex **must also give the optimal solution.**

Max-flow min-cut theorem: The size of the maximum flow of a network equals the capacity of the smallest (s, t)-cut.

Each iteration of simplex is **efficient**, as we are essentially performing DFS or BFS to find an $s - t$ path in G^f, which is $O(|E|)$. By induction, since $\sum_e c_e = C \in \mathbb{Z}^+$, the maximum flow is $C|E|$ and therefore the number of iterations is the same. However, using BFS (ensuring the lowest number of edges) we can cap iterations at $O(|V||E|)$ and intelligently choosing our paths gives us $O(|V||E|^2)$ as the total time for finding the max flow.

6.4 Duality

In networks, flows are smaller than cuts – however, the min-cut and max-flow coincide! Every linear maximization problem has a dual minimization problem. In fact, the maximum of every primal must be the minimum of its dual. The generic form of the primal and the dual is shown:

$$\begin{array}{rlcl} & \max\ \mathbf{c}^\top \mathbf{x} & & \\ \textbf{Primal:} & \mathbf{A}\mathbf{x} & \leq & \mathbf{b} \\ & \mathbf{x} & \geq & 0 \end{array} \implies \begin{array}{rlcl} & \min\ \mathbf{y}^\top \mathbf{b} & & \\ \textbf{Dual:} & \mathbf{y}^\top \mathbf{A} & \geq & \mathbf{c}^\top \\ & \mathbf{y} & \geq & 0 \end{array}$$

Duality Theorem: If a linear program has a bounded optimum, then

so does its dual, and these two values coincide.

6.5 Zero-sum Games

The payoff in a game can be represented as a **matrix** of values where each entry is the **gain** of the row and the **loss** of the column. Row wants a strategy that maximizes gain while Column wants a strategy that minimizes loss. In a game like rock paper scissors, if we calculate the expected payoff (easily calculated using linearity of expectation) we see that Row forces an expected payoff of 0 and likewise Column forces and expected payoff of 0. The best either player can do given a random strategy is 0, as we would intuitively presume.

For more complex games, we can begin to see that the optimum strategy can be solved for using the concept of duals. If Row announces some strategy, she knows that there is some pure solution that Column can come up to "counter" her strategy. Since Row can also determine what Column's pure strategy would be, she can defensively select her moves to **maximize** the **minimum** payoff of Column's pure strategy. Similarly, if Column has to declare his strategy first, he should choose the strategy that minimizes his loss against Row's best response. These two linear programs are dual to each other and have the same optimum.

In summary, Row can guarantee an expected outcome V or higher regardless of Column's strategy. Similarly, Column can guarantee an expected loss of V or lower no matter what Row does. This can be generalized as the **min-max theorem**, a fundamental result of game theory.

min-max theorem:

$$\max_{\mathbf{x}} \min_{\mathbf{y}} \sum_{i,j} G_{ij} x_i y_j = \min_{\mathbf{y}} \max_{\mathbf{x}} \sum_{i,j} G_{ij} x_i y_j.$$

7 NP-Complete Problems

7.1 Search Problems

All the algorithms we've seen so far are **efficient**, meaning their time requirement grows as a polynomial function of the input size. Most problems we've been able to solve through DP, LP, graph traversal, and divide-and-conquer are able to efficiently reduce an exponential number of solutions to a polynomial time problem. Here we examine those problems for which there exists no efficient reduction to polynomial time.

7.1.1 Satisfiability

The SATISFIABILITY or SAT problem is a classic example of an NP-HARD problem. An instance of SAT looks like this:

$$(x \vee y \vee z)(x \vee \bar{y})(y \vee \bar{z})(z \vee \bar{x})(\bar{x} \vee \bar{y} \vee \bar{z})$$

This is a **boolean formula** in **conjunctive normal form** (CNF), a collection of clauses consisting of the disjunction of several literals. A truth assignment is a boolean assignment of TRUE or FALSE to each variable. The question is then, **is there a truth assignment that satisfies all clauses?** It's hard to do this in general, as there are 2^n possible assignments. SAT is a **search** problem, where we are given an instance **I** and asked to find a solution S. Formally, a search problem is specified by an algorithm $\mathcal{C}$ that takes two inputs, an instance I and a proposed solution S, and runs in time polynomial in $|I|$. S is only a solution to I iff $\mathcal{C}(I, S)$ is TRUE. The fastest algorithms for this problem are still exponential in nature.

There are certain cases where we have a method to find a solution though. If the instance has at most one positive literal, the formula is called a **Horn formula** an can be approached greedily. If all clauses have only two literals, we can use graph theory to find the strongly connected components of a graph constructed from the instance (2SAT).

7.1.2 The Traveling Salesman Problem

The traveling salesman problem (TSP) attempts to find a tour in a graph of total length b or less. We dress TSP as a search problem instead of an **optimization problem** (i.e. what is the shortest possible Hamiltonian circuit/Rudrata cycle) because we want to be able to compare problems (i.e. reduce problems to each other). Like SAT, TSP has no polynomial time solution.

7.1.3 Euler and Rudrata

The search problem EULER PATH attempts to find whether an Eulerian Path exists in a graph (where each edge is visited exactly once). This is true if and only if the graph is connected and every vertex (with the possible exception of the start and final vertices) has odd degree. The search problem RUDRATA

CYCLE instead attempts to (similar to TSP) deduce whether a cycle exists in a graph that visits each vertex exactly once. The RUDRATA PATH problem is similar to to the RUDRATA CYCLE problem, except now the start and end vertices don't have to be the same. There is a strong equivalence between both versions of the problem.

7.1.4 Cuts and Bisections

The MIN CUT problem is to find a cut in a graph **G** with at most **b** edges. This problem can be found in polynomial time with $n-1$ max-flow computations (find max flow from one vertex to every other vertex with edge weights with capacity 1. The smallest of these max flows (by the max-flow min-cut theorem) corresponds to the min cut of the graph. The BALANCED CUT problem attempts to partition a graph G into two sets S, T such that there are at most b edges between S and T and that S and T have a minimum size. This forms the essence for **clustering** problems.

7.1.5 Integer Linear Programming

There exists a polynomial algorithm for linear programming. A much more difficult problem arises when we constrain the solution to be $\in \mathbb{Z}$ Given $\mathbf{A}$ and $\mathbf{b}$ and a nonnegative integer vector $\mathbf{x}$ satisfying the inequalities $\mathbf{Ax} \leq \mathbf{b}$, or report that none exists.(INTEGER LINEAR PROGRAMMING). A special case of ILP is ZERO-ONE EQUATIONS, or ZOE. Here we attempt to to solve the linear equation $\mathbf{Ax} = \mathbf{1}$, where $a_{i,j} \in \{0, 1\}$.

7.1.6 Three-Dimensional Matching

We've already seen how to reduce BIPARTITE MATCHING to max-flow. However, a generalized form of bipartite matching, 3D MATCHING, where we have vertices representing triplets (i.e. **n** boys, **n** girls, and **n** pets) instead of pairs, has no known polynomial time algorithm.

7.1.7 Independent Set, Vertex Cover, Clique

The INDEPENDENT SET problem, where we aim to find sets of vertices with no shared edges, can efficiently be solved on trees. However, for general graphs no polynomial time algorithm is known. It is closely related to VERTEX COVER, a special case of SET COVER where we attempt to find b vertices that are collectively adjacent to every edge. Another closely related problem is CLIQUE, where given a graph and a goal **g** we attempt to find a set of **g** vertices such that all possible vertices between them are present (i.e. the induced subgraph is complete).

7.1.8 Longest Path

The SHORTEST PATH algorithm is well known and extensively studied, there is no known polynomial time solution to the LONGEST PATH problem, where we are asked to find a path from s to t with weight **at least** g.

7.1.9 Knapsack and Subset Sum

Recall the KNAPSACK problem from the Dynamic Programming section. While we were able to find a DP algorithm in $O(nW)$, this is still exponential (since we depend on W and not $\log W$). There is no known polynomial time algorithm for KNAPSACK. However, UNARY KNAPSACK, where we code all integers in **unary**, does have a polynomial-time solution. There is also a special case of KNAPSACK, SUBSET SUM, where we try to find a subset of a set of integers that sums to exactly W. Both problems are hard.

7.2 NP-Complete Problems

All NP-Complete problems have alternate versions which are easy to solve.

$$\text{3SAT} \iff \text{2SAT, HORN SAT}$$
$$\text{TSP} \iff \text{MST}$$
$$\text{LONGEST PATH} \iff \text{SHORTEST PATH}$$
$$\text{3D MATCHING} \iff \text{BIPARTITE MATCHING}$$
$$\text{KNAPSACK} \iff \text{UNARY KNAPSACK}$$
$$\text{independent set} \iff \text{INDEPENDENT SET on trees}$$
$$\text{ILP} \iff \text{LP}$$
$$\text{RUDRATA PATH} \iff \text{EULER PATH}$$
$$\text{BALANCED CUT} \iff \text{MIN CUT}$$

A crucial observation is that all the problems on the LHS are hard for the **same reason**, and as such we determine that **all NP-Complete problems can be reduced to each other.**

7.2.1 P and NP

All problems whose solution can be **verified** in polynomial time are in **NP**. The class of problems that can be **solved** in polynomial time are in **P**. All **NP**-Complete problems are exactly as hard as each other, if a polynomial time algorithm can solve one of them, the same algorithm can solve all of them.

7.3 The Reductions

7.3.1 Rudrata (s, t)-Path → Rudrata Cycle

7.3.2 3Sat → Independent Set

7.3.3 Sat → 3Sat

7.3.4 Independent Set → Vertex Cover

7.3.5 Independent Set → Clique

7.3.6 3Sat → 3D Matching

7.3.7 3D Matching → ZOE

7.3.8 ZOE → Subset Sum

7.3.9 ZOE → ILP

7.3.10 ZOE → Rudrata Cycle

7.3.11 Rudrata Cycle → Traveling Salesman Problem

7.3.12 Any Problem in NP → SAT

I can't summarize this chapter any better than the book can soooo the book is your best bet

8 Algorithms with Numbers

8.1 Modular Arithmetic

Modula arithmetic is a system for dealing with restricted ranges of integers. $x \equiv r \mod N$ iff $\exists q \in \mathbb{Z} | x = qN + r$. In other words, $x \equiv y \mod N$ iff $(x - y)$ divides N. Addition and multiplication remain well defined – if $x \equiv x' \mod N$ and $y \equiv y' \mod N$, then $x+y \equiv x'+y' \mod N$ and $xy \equiv x'y' \mod N$. Commutativity, distributivity, and associativity all hold under modulo operators. Addition is $O(n)$, multiplication is $O(n^2)$, and division is $O(n^3)$. The pseudocode for modular exponentiation is below.

Pseudocode:

```
Input: Two n-bit integers x and N, an integer exponent y
Output: x^y mod N
procedure modexp(x, y, N):
    if y=0: return 1
    z = modexp(x, floor(y/2), N)
    if y is even:
        return z^2 mod N
    else:
        return x * z^2 mod N
```

The essence of the modular exponentiation is that we can break down any y using binary and combine the product into our final answer. For instance, $x^{25} = x^{16} \times x^8 \times x^1$, which only requires computing $x^{2\times4} \times x^{2\times3} \times x$.

There is also a process for finding the **GCD** of two numbers. **Euclid's rule** states that if **x** and **y** are positive integers, with $x \geq y$, $\gcd(x, y) = \gcd(x \mod y, y)$. this allows us to write the **extended Euclid's algorithm**:

Pseudocode:

```
Inputs: Two positive integers a, b, with a >= b
Outputs: Integers x, y, d such that d = gcd(a, b) and ax + by = d

procedure extended-euclid(a, b):
    if b = 0: return (1, 0, a)
    (x', y', d') = extended-euclid(b, a mod b)
    return (y', x' - floor(a/b)y', d)
```

If **d** divides both **a** and **b**, and $\mathbf{d = ax + by}$, then $d = \gcd(a, b)$.

x is the **multiplicative inverse** of $a \mod N$ if $ax \equiv 1 \mod N$. For any $a \mod N$, a has a multiplicative inverse modulo N if and only if a is relatively prime to N. The inverse can be found in $O(n^3)$ using the Extended Euclid Algorithm.

8.2 Primality Testing

Fermat's Little Theorem states that if **p** is prime, then for every $a \in [1, p)$, $a^{p-1} \equiv 1 \mod p$.

Pseudocode:

```
Input: Positive Integer N
Output: Yes/No

procedure primaility(N):
    pick a random a < N
    if a^(N-1) == 1 mod N:
        return Yes
    else:
        return No
```

We can choose a randomly for our test since that if the primality test fails for some value a relatively prime to N, then it must fail for **at least half the possible values for** a**.** Instead of just once, if we repeat the primality test for k randomly sampled values, the probability that we make an incorrect call is $1/2^k$, which quickly becomes insignificantly small.

Since $\pi(n) = x/\ln(x)$, randomly picking numbers and running our primality test will generate a prime in (on average) $O(n)$ time.

8.3 Universal Hashing

We want a way to take a large set of data and store it in such a way that accessing a "bin" takes time proportional to the number of accesses needed on average but still has fast lookup times.

Hash tables offer a small "nickname" to each item and then group them by that nickname. This way we can easily access the bucket where the item is and hopefully take only a short amount of time to traverse the bucket to find the correct item. We assign these nicknames using a **hash function**.

Selecting a hash function is difficult – we must select one that is able to both spread around data among buckets while still having some consistent scheme behind it. A good way to ensure some degree of **randomness** is to use some of the properties of primality – select a prime number close to the desired nubmer of buckets, and assign to each item an item $\mod$ the prime. Each bucket would have on average $\frac{1}{n}$ items, meaning it has an equivalent spread to just randomly assigning values (but we have a method behind our hash function, which provides consistency).

More generally, we choose a prime n for the number of desired buckets. For performance, we normally select a value around twice as large as the number

of items. Assume the domain of all data items to be $N = n^k$. Then each data item is a k-tuple of integers modulo n, with $\mathcal{H} = \{h_a : a \in \{0, ..., n-1\}^k\}$.

9 Hashing, Streaming, and Multiplicative Weights

9.1 Hashing

We continue our discussion on hashing from the previous section. The goal is to create a table of size $m = O(n)$, call it $T[1, ..., m]$, and create a hash function $h : U \mapsto [m]$ that maps each item in the universe U to a bucket $T[h(x)]$. We've already discussed a simple universal hashing scheme which simulates the spread of a random assignment but has a distinct method behind it. The largest bucket in this case has size $\log n$.

We can modify this system by, instead of having a singular hash h_1, introducing another function h_2. Now, for each $x \in U$ we add x to the less full of $T[h_1(x)]$, $T[h_2(x)]$. Now instead of being $\log n$, the size of each bucket only has $\log \log n$ items.

9.2 Streaming

A **stream** is a collection of data where items are presented one at a time, and is especially prevalanet when the amount of data exceeds the total amount of available memory.