CSE 101 Review

Taanish Reja and Kevin Jacob

Fall 2024

Table of Contents

- Minimum Spanning Tree
  - Cut Property
  - Kruskal's algorithm
    - Definition
    - Runtime
  - Prim's algorithm
    - Definition
    - Runtime

3. Greedy Algorithms
   - Exchange Argument
   - Greedy Stays Ahead
   - Greedy Achieves the Bound
4. Divide and Conquer
   - Master Theorem
     - Characteristic Polynomial
     - Recurrence
     - Examples
   - Selection and Sorting
5. Backtracking
6. Dynamic Programming
7. P and NP

1 CSE 21 Review

1.1 Asymptotic Classes

1.1.1 Big Theta

Definition: $f(n) \in \Theta(g(n))$ if there are constants $C, C',$ and k such that $f(n) \leq Cg(n)$ and $g(n) \leq C'f(n)$ for all $n \geq k$. In other words, Big Θ defines a tightly bound relationship (grows just as fast).
Limit definition: $f(n) \in \theta(g(n))$: $\lim_{x\to\infty} \frac{f(n)}{g(n)} = c$ where c is finite and $c \neq 0$

1.1.2 Big Omega

Definition: $f(n) \in \Omega(g(n))$ if there are constants C and k such that $f(n) \geq Cg(n)$ for all $n \geq k$. In other words, Big Ω defines a lower bound relationship.
Limit definition: $f(n) \in \Omega(g(n))$: $\lim_{x\to\infty} \frac{f(n)}{g(n)} = c$ where $c > 0$ or $c = \infty$

1.1.3 Little O

Definition: if $g(n)$ grows strictly faster than $f(n)$, then $f(n) \in o(g(n))$. Put another way, $f(n) \in o(g(n))$ and $f(n) \notin o(g(n))$
Limit definition: $f(n) \in o(g(n))$: $\lim_{x\to\infty} \frac{f(n)}{g(n)} = 0$

1.1.4 Big O

Definition: $f(n) \in O(g(n))$ if there are constants C and k such that $f(n) \leq Cg(n)$ for all $n \geq k$. In other words, $g(n)$ grows just as fast or faster than $f(n)$.
Limit definition: $f(n) \in O(g(n))$: $\lim_{x\to\infty} \frac{f(n)}{g(n)} = c$ where c is finite
Examples:

- $2^n \in O(n^2)$
  $\lim_(n \to \infty)\frac{2^n}{n^2} =$ (Apply L'Hopital's rule)
  $\lim_(n \to \infty)\frac{2^n \ln 2}{2n}$ (Apply L'Hopitals rule
  $\lim_(n \to \infty)\frac{2^n \ln 2^2}{2} = \infty$
  Thus, the statement is false.

- $F_n \in O(2^n)$ where F_n is a function representing the fibonacci sequence.
  $F_n = F_{n-1} + F_{n-2}, F_0 = 1, F_1 = 1$
  Claim: $F_n \leq 2^n$ for all $n \geq 0$
  Base cases:
  $F_0 = 1, 2^0 = 1$
  $F_1 = 1, 2^1 = 2$
  Induction step:
  Let k be an arbitrary integer such that $k > 1$.
  Assume that $F_m \leq 2^m$ for all m in the range $0 \leq m \leq k$.
  WTS $F_k \leq 2^k$:
  $F_k = F_{k-1} + F_{k-2} \leq 2^{k-1} + 2^{k-2} = 2^{k-2}(2+1) \leq 2^{k-2}(4) = 2^k$
  Thus, the statement is true.

1.1.5 Big O Class Properties

Domination: If $f(n) \leq g(n)$ for all n then $f(n) \in O(g(n))$
Transitivity: If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$ then $f(n) \in O(h(n))$
Additivity/ Multiplicativity: If $f(n) \in O(g(n))$ and $h(n)$ is non-negative then
$f(n) + h(n) \in O(g(n) + h(n))$ and $f(n) \cdot h(n) \in O(g(n) \cdot h(n))$
Sum is maximum: $f(n) + g(n) \in O(\max\{f(n), g(n)\})$
Ignoring constants: For any constant c, $c \cdot f(n) \in O(f(n))$

2 Graphs

2.1 Depth First Search

2.1.1 Definition

Finds if there is a path between the start vertex and the other vertices in a graph.
Pseudocode:

```
procedure GraphSearch($G$: Graph, $s$: vertex)
Initialize $X$ = empty, $F = s$, $U = V - F$
while $F$ is not empty:
        Pick $w$ in $F$
        for every $(w, y) \in E$:
            If y is not in $X$ or $F$, then move $U$ to $F$
        Move $w$ from $F$ to $X$
```

Explored (X) contains a list of every vertex we have explored. This prevents us from checking already explored vertices.
Frontier (F) contains a list of the vertices we have reached but not explored. We add vertices in ascending order.
Unexplored (U) contains a list of vertices we have not explored.
For low level implementations, we prefer the use of a stack, as it allows us to explore a branch from one vertex (as in deepest paths) instead of waves like a BFS. The Frontier and Unexplored lists replicate this behavior.

2.1.2 Correctness

In order to prove correctness, we must prove the following (bidirectional proof)

- If $v \in X$ then there is a path from s to v (Induction Proof)
  Base Case: Before going through the loop, X is empty and $F = \{s\}$
  Loop Invariant: After the ith iteration of the while loop, every element of X or F is reachable from s in G

  1. You pick a vertex v in F
  2. We move all neighbors of v into F it they are in U
     If there is a path from s to v and an edge (v, u) then there is a path from s to u

3. We move v from F to X
   By the inductive hypothesis, we know that there is a path from s to v

- If $v \notin X$ then there is not a path from s to v (Contradiction Proof)

  1. Suppose that by contradiction that there is a vertex v reachable from s that is not in X.Then there is a path from s to v. Let z be the last vertex in the path that is in X and w be the next vertex after z in the path (vertices all in a line).
  2. Then z must have been in F at some point. And when z was picked from F, w must have been moved from F to X which contradicts the assumption that w is no in X

2.1.3 Runtime

Analyzing pseudocode:

```
procedure GraphSearch($G$: Graph, $s$: vertex)
Initialize $X$ = empty, $F = s$, $U = V - F$
while $F$ is not empty: $\rightarrow$ Runs at most once per vertex
        Pick $w$ in $F$ $\rightarrow$ Constant time $c$
        for every $(w, y) \in E$: $\rightarrow$ outdeg($w$) iterations
            If y is not in $X$ or $F$, then move $U$ to $F$ $\rightarrow$ Constant time $c'$
        Move $w$ from $F$ to $X$ $\rightarrow$ Constant time $c''$
```

Notes:
outdeg(w) $\rightarrow$ Degree of each vertex
Total runtime:
$\sum_{w \in V}(c) + \sum_{w \in V}(c'') + \sum_{w \in V}(c'\text{outdeg}(w) =$
$(\sum_{w \in V}(c) + \sum_{w \in V}(c'')) + (\sum_{w \in V}(c'\text{outdeg}(w)) =$
$O(|V|) + O(|E|) =$
$O(|V| + |E|)$
The simplifications occur because summing a constant over all vertices is equivalent to counting the vertices, and checking all edges of each vertex is equivalent to counting all the edges
Thus, the runtime is $O(|V| + |E|)$

2.1.4 Reduction Example

Problem: For a given graph $G(V, E)$, determine if there is vertex v such that for all vertices $u \in V$, there exists a path between u and v.
Algorithm:

```
1. Run DFS on $G$.s
2. Identify $h$ as the vertex with the highest post number from the previous step.
3. Run explore on $G$ starting from $h$.
4. if all vertices are visited then
5.     return TRUE
```

```
6. else
7.     return FALSE
```

Runtime analysis:
DFS and Explore will each take $O(|V + E|)$ steps. Finding if all vertices are visited will take $O(|V|)$ steps. Thus, the runtime of this algorithm is $O(V + E)$. **Correctness proof:**
Forward direction
The algorithm returns true when there is a vertex in the graph that is connected to all other vertices by a path
Assume there is a vertex H in the end of the graph that is connected to all other vertices in the graph. After running DFS on G, the vertex with the highest post number will be H, since it is at the end of the graph. If explore is ran on G starting from H, then all vertices will be visited, since H is connected to all the other vertices in the graph. Therefore, the algorithm correctly returns true when there is a vertex in the graph that is connected to all other vertices by a path. QED.
Backwards direction
The algorithm returns false when there is not a vertex in the graph that is connected to all other vertices by a path.
The algorithm returns false when performing explore from the vertex with the highest post order. The vertex with the highest post order is the vertex that can reach all corresponding vertices in its chain. If the DFS only has one chain then the highest post order vertex will be the starting vertex and it can reach all vertices. If the DFS needs multiple chains, we can perform explore from the vertex with the highest post order and the vertices in that vertex's chain are only reachable from that vertex. Thus, we would have to start explore from that vertex in order to reach the vertices in the chain and if explore doesn't visit all the vertices the algorithm returns False thus there isn't a vertex that satisfies the desired property.

2.2 Directed Acyclic Graphs

2.2.1 Definition

A directed graph without a cycle

2.2.2 Linearization of DAGs

```
procedure linearize(a DAG $G = V, E$)
    run$DFS(G)$        return list of vertices in decreasing order of post numbers
```

2.2.3 Properties

- Every edge in DAG goes from a higher post number to a lower post number
  Proof: Suppose (u, v) is an edge in a dag. Then, it can't be a back edge, therefore it can only be aforward edge/tree eddge or a cross edge, both of which have the property that $post(v) < post(u)$

- Every Directed graph is a DAG of its strongly connected components. Some SCCs are sink SCCs and some ar esource SCCs

Sink scc: A SCC with no outgoing edges. If explore is performed on a sink SCC, only vertices from that SCC will be visited. There is no way to identify a source SCC from post numbers.

Source scc: A SCC with no incoming edges. The vertex with the greatest post number in any DFS output tree belongs to a source SCC.

Proof: If C and C' are strongly connected components and there is an edge from a vertex in C to a vertex in C' then the highest post number in C is greater than the highest post number in C'.

Case 1: DFS searches C before C'

Then at some point dfs will cross into C' and visit every edge in C' then retrace its steps until it gets back to the first node in C it started with and assign it the highest post number.

Case 2: DFS searches C' before C.

Then DFS will visit all vertices of C' before getting struck and assign a post number to all vertices of C'. Then it will visit some vertex of C later and assign post numbers to those vertices.

2.2.4 Construction Example

Algorithm

```
Input: Graph $G = (V, E)$, $s, t \in V$
    Create a new graph $G' = (V', E')$ as follows:
    For every vertex $v$ in $V$, we will split into two copies $v_e$ and $v_o$.
    For every edge $(v, u)$ of $E$:
        If $w((v, u)) = 1$, add edges $(v_e, u_o)$ and $(v_o, u_e)$.
        If $w((v, u)) = 0$, add edges $(v_e, u_e)$ and $(v_o, v_o)$.
    Run explore on $G'$ starting from $s_e$. If $t_e$ is visited, then return true. Otherwise, return false.
```

Runtime analysis:

Graph construction will take $(|2V| + |2E|)$, since we are making an even and odd copy of each vertex. Thus, the runtime of graph construction is $O(|V| + |E|)$. Meanwhile, running explore will take $O(|V| + |E|)$. Since these steps are sequential, this algorithm will take $O(|V| + |E|)$ time

Correctness proof:

Forward direction If there is a path in G from s to t such that the sum of the edge weights is even, then the algorithm returns true:

Assume there is a path in G from s to t such that the sum of the edge weights is even. For this to happen when our edge weights are only 0 or 1, there needs to be an even amount of edges with edge weight one. If there is an even amount of edges with edge weight one, and we start from a even vertex in G', we will switch an even amount of times, and end up in another even vertex of G'. Thus, if there is a path from s to t such that the sum of the edge weights is even, there will be path from s_e to t_e. Our algorithm returns true if explore(G', s_e) returns true, and by the correctness of explore, we can conclude that the algorithm returns true.

Backward direction If there is not a path in G from s to t such that the sum of the edge weights is even, then the algorithm returns false.

We will prove the contrapositive: If the algorithm returns true, then there is a walk from s to t with an even sum of edge weights

If the algorithm returns true, then there is a walk from s to t with an even sum of edge weights. Assume the algorithm returns true, this means that DFS found a path s_e to t_e within G'. When constructing G', every vertex in G is split into v_e and v_o, representing walks of even and odd weight when arriving at that vertex. An edge in G of 1 will switch our parity/weight (v_ie goes to v_io and so on). if the traversal reaches t_e, it implies that there exists a sequence of edges from s_e to t_e, which corresponds to a walk from s to t in the original graph with an even sum of edge weights. Since the traversal reached t_e, the parity of the sum of the edge weights on this walk must be even. Therefore, if the algorithm returns true, it implies that a walk from s to t with an even sum of edge weights exists in the original graph G.

2.3 Connected Components

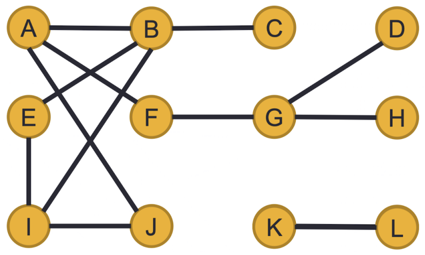
2.3.1 Explore Pre/Post Numbers

explore is our basic graphsearch algorithm that works by starting *explore* on a starting node and performing *explore* on all the starting nodes neighbors. We go through the neighbors for each node in alphabetical order and *explore* works like a DFS. We can also add pre/post numbers to each vertex to keep track of when a vertex enters and leaves the stack.

```
procedure explore(G = (V,E),s):
1. visited(s) = true
2. previsit(s)
3. for each edge (s,u):
4.     if not visited(u):
5.         prev(u) = s
6.         explore(G,u)
7. postvisit(s)

procedure previsit(v):
1. pre(v)=clock
2. clock++

procedure postvisit(v):
1. post(v)=clock
2. clock++

prodecure DFS(G):
1. cc = 0
2. clock = 1
3. for each vertex V:
4.     visited(v) = false
```

```
5. for each vertex V:
6.     if not visited(u):
7.         cc++
8.         explore(G,v)
```

2.3.2 Edges on Undirected Graphs

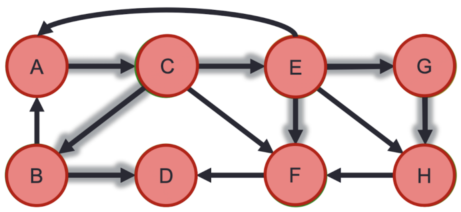


vertex	pre	post	prev	cc
A	1	20	∅	1
B	2	11	A	1
C	3	4	B	1
D	14	15	G	1
E	5	10	B	1
F	12	19	A	1
G	13	18	F	1
H	16	17	G	1
I	6	9	E	1
J	7	8	I	1
K	21	24	∅	2
L	22	23	K	2

Back Edges: Back edges in an undirected graph G that has been explored are edges in G that are not in the DFS tree of G. Each back edge corresponds to a cycle and we can check if an edge is a back edge by looking at previous pointers to see if edge exist in output tree.
Cycle Edges: Removing an edge that is in a cycle will not disconnect an undirected graph and removing an edge that is not in a cycle will disconnect an undirected graph.

An undirected connected graph G has a cycle if and only if it's DFS output tree has a back edge
Connected Undirected Graphs: An undirected graph is connects if for every pair of vertices u,v in G, there exists a path from u to v. *explore* only reaches one connected component of the graph (the set of vertices reachable from s)

2.3.3 Edges on Directed Graphs



vertex	pre	post	prev	comp
A	1	16	∅	1
B	3	6	C	1
C	2	15	A	1
D	4	5	B	1
E	7	14	C	1
F	8	9	E	1
G	10	13	E	1
H	11	12	G	1

DFS Alphabetical Order

vertex	pre	post	prev	comp
A	10	15	E	3
B	12	13	C	3
C	11	14	A	3
D	3	4	F	1
E	9	16	∅	3
F	2	5	H	1
G	7	8	∅	2
H	1	6	∅	1

DFS Reverse Alphabetical Order

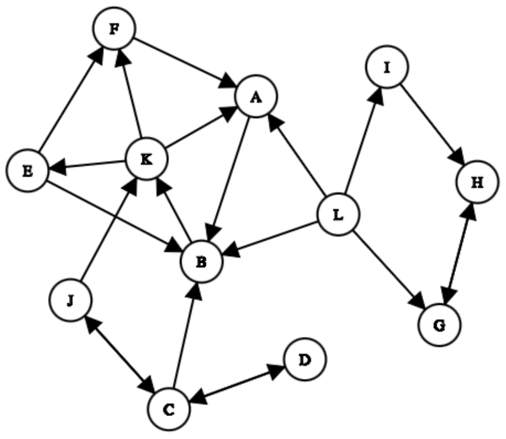
Tree edge: Solid edge included included in the DFS output tree
Back edge: Leads to ancestor (A directed graph G has a cycle if and only if its DFS output)
Forward edge: Leads to a descendant
Corss edge: Leads to neither ancestor or descendant

1. (u, v) is a tree/forward edge then
   $pre(u) < pre(v) < post(v) < post(u)$
2. (u, v) is a back edge
   $pre(v) < pre(u) < post(u) < post(v)$
3. (u, v) is a cross edge then
   $pre(v) < post(v) < pre(u) < post(u)$

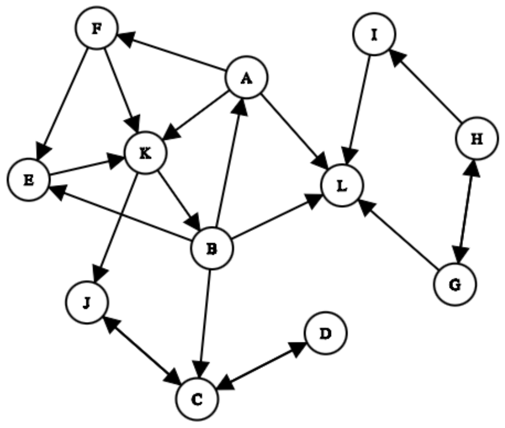
2.3.4 Strongly Connected Components

- - Two vertices u and v in a directed graph are *strongly connected* if there exists a path from u to v and a path from v to u.
  - The maximal set of strongly connected vertices is called a *strongly connected component*
  - Every directed graph is a DAG of its strongly connected components
  - **Sink SCC:** If explore is performed on a vertex that is a sink SCC, then only vertices from that SCC will be visited. There is not a direct way to find a sink SCC.
  - **Source SCC:** The vertex with the greatest post number in a DFS output tree belongs to a source SCC
- SCC Algorithm
  1. Given a graph G, let G^R be the reverse graph of G. The sources of G^R are the sinks of G.
  2. Run DFS on G^R and keep track of the post numbers
  3. Run DFS on G and order the vertices in decreasing order ot the post numbers from the previous step
  4. Each time DFS increments **cc**, you have found a new SCC
- Runtime
  It is linear time for each step of the SCC algorithm so it is linear time in general, $O(|V| + |E|)$

- Example



Original Graph



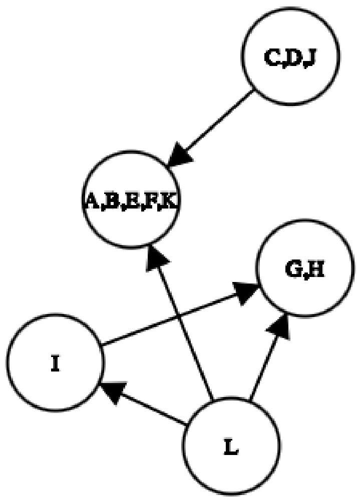
Reverse Graph

Node	Pre	Post	Prev
A	1	18	∅
B	5	14	K
C	6	11	B
D	7	8	C
E	3	16	F
F	2	17	A
G	19	24	∅
H	20	23	G
I	21	22	H
J	9	10	C
K	4	15	E
L	12	13	B

Reverse Graph DFS

Node	Pre	Post	Prev	CC
A	7	16	∅	3
B	8	15	A	3
C	19	24	∅	5
D	22	23	C	5
E	12	13	K	3
F	10	11	K	3
G	1	4	∅	1
H	2	3	G	1
I	5	6	∅	2
J	20	21	C	5
K	9	14	B	3
L	17	18	∅	4

Traversal on original graph



Meta graph

2.4 Reduction and Bidirectional Proofs

2.4.1 Algorithm Modification

Modify an existing algorithm. Requires induction.

2.4.2 Reduction

Use an existing algorithm (DFS) as a subroutine (possibly changing the input). Usually proved with a bidirectional proof.

2.4.3 Max Bandwidth

Uses reduction for more concise correctness proof:
Algorithm:

```
MaxBandDecision($G, s, t, M$):
    Construct $G_m$ (by removing all edges less than $M$ from $G$)
    Run graphsearch($G_m, s$)
    if t is visited:
        return TRUE
    else
```

```
        **return** FALSE
```

Proof of correctness:
Case 1: If there is a path in G from s to t with bandwidth at least M then **return** TRUE
Proof: Suppose there is a path p in G from s to t with bandwidth at least M. Then every edge in p has weight $\geq M$. Therefore p is a path in G_m. So the algorithm will output TRUE.
Case 2: If there is not a path from s to t with bandwidth at least M then return FALSE
Proof (contrapositive): If the algorithm returns TRUE then there is a path from s to t in G with bandwidth at least M.
Suppose the algorithm returns TRUE. Then there is a path p in G_m from s to t. Then p is a path in G such that all edge weights are $\geq M$. Runtime:
Graph creation: $O(|V| + |E|)$
Graphsearch: $O(|V| + |E|)$
Total time: $O(|V| + |E|)$

2.5 Breadth First Search

2.5.1 Definition

Finds the shortest path (number of edges) to every vertex from a specific vertex. If a vertex cannot be reached, its distance is infinite.
Pseudocode:

```
**procedure** BFS
Input: $G(V, E)$, vertex $s$ in V
**for** each vertex $u$ in $V$:
    dist($u$) $= \infty$
$dist(s) = 0$
$Q = [s]$ (queue containing s)
**while** $Q$ is not empty:
    $u =$ eject($Q$)
    **for** all edges $(u, v)$ in $E$:
        **if** dist($v$) $= \infty$ then:
            inject($Q, v$)
            dist($v$) =dist($u$) $+ 1$
```

2.5.2 Corrctness

For each vertex V, we want to show that dist(v) is the minimum distance of all paths from s to v.
Claim: For each distance value $d = 0, 1, 2$:

- all vertices at distance $\leq d$ from s have their distance values set correctly

- all other vertices (distance ¿ d from s) have distances set to ∞

- The queue contains exactly the nodes at distance d

Proof:
Base case: (for d = 0)

- dist(s) = 0 is the correct distance value

- all other vertices have distances set to ∞

- The queue contains only s which is the only vertex at distance 0

Induction step:
Let k be an arbitrary integer such that $k \geq 0$. Assume that the above 3 statements are true for when $d = k + 1$
All vertices have distance $\leq k$ have been set and the queue only contains vertices at distance $= k$
Suppose v is the next vertex to be popped from the queue and let u be a neighbor of v.

- if dist(u) $\neq \infty$, then by IH, dist(u) has been set correctly and it is not updated

- if dist(u) $= \infty$, then dist(u) = dist(v) + 1 = $k + 1$

All new vertices added to the queue have distance $k + 1$ and are set correctly
All vertices of distance $k + 1$ have been added to the queue
The queue only contains the nodes at distance $k + 1$

2.5.3 Runtime

Each vertex enters the queue at most one time. Similarly to DFS, we also look at each edge of each vertex. Thus, BFS runs in $O(|V| + |E|)$ time.

2.5.4 Reduction Example

Problem statement: You are managing a network of cell-phones with a total of n cell-phones: $C_1, \ldots, C_n$. You have a record of m cellphone message timestamps in the form (C_i, C_j, t) which means that cellphone i sent a message to cellphone j at time t (you can assume all of the times are different.). You are informed that cellphone 1 was infected by a virus at time t_0. If cellphone i has a virus and sends a message to cellphone j then cellphone j will contract the virus (the message will be sent immediately and cellphone j will get the virus whether or not the message is opened.)
Graph Construction: Create a new graph G. For each message, create a new node for the sender and receiver where each node is a tuple containing the cell phone and timestamp and an edge between the sender and receiver nodes, where the weight of these edges is 1. Additionally, create a node with cellphone 1 as

the label and the infection time as the timestamp and an edge with weight 0 to the message node with the timestamp directly after the infection time. Sort the node list by timestamp. Then, iterate through the node list, adding nodes from the same phone to a separate list for each cell phone. Then iterate through each cell phone list, creating an edge between adjacent nodes with weight 0. The number of nodes created will be $O(2m+1) = O(m)$, since for each message we create two nodes and we also create a node for the infection time. The number of edges will be $O(2m+1) = O(m)$, since in the worst case each message accounts for edge, nodes with the same phone label are connected, and there is an edge from the infection node to the node denoting the infected phone's first message.

Algorithm Pseudocode:

```
procedure isInfected:
input: G(V,E), s (starting node), k (target label)
    Run BFS on $G(V, E)$ starting from $S$
    for node in nodes:
        if (dist(node) < $\infty$ && node.label == k):
            return node.timestamp
    return false
```

Runtime analysis:

Running BFS takes $O(|V|+|E|)$, which in this case is $O((2m+1)+(2m+1)) = O(m)$. Then, we iterate over all the nodes and check their distance and label. In the worst case, this takes $O(m)$. Thus, the algorithm takes $O(m)$ time to run.

Correctness Proof:

Forward direction:

If a cellphone is infected, then the algorithm returns the earliest infection time.

In the construction of G, any time an arbitrary message with label i is infected, it will transmit its virus to all other message nodes with label i after the message's timestamp. Thus, messages with label i sent after the first message with label i was infected will be capable of spreading the virus. Thus, the graph correctly models the transmission of the virus. If cellphone i is infected, then $\text{dist}(node) < \infty$ for each node labeled with cell phone i because of the correctness of BFS. Since the node list is sorted by timestamp in graph construction, when we loop over all the nodes after running BFS and check their distance and label, the first node with the label provided as input and distance less than ∞ will be the node that was infected earliest. Thus, if a cell phone is infected, the algorithm correctly returns the earliest possible infection time.

Backward direction: If a cellphone is not infected, then the algorithm returns ∞
Contrapositive: If the algorithm returns a timestamp, that timestamp was the cellphone's earliest infection time.
The algorithm begins by running BFS from vertex s, which represents the first infected cellphone. BFS guarantees that for any vertex v, the value dist[v] represents the shortest path (in terms of the number of edges) from s to v. Therefore, if dist[v] is not infinity (the algorithm returned a timestamp) there exists a path from s to v, meaning the infection could have potentially spread to v from s. Finally suppose the algorithm returns timestamp t for vertex f. If there existed an earlier timestamp t' (where t' ¡ t) such that cellphone f could have been infected, then f would have been found in the iteration at timestamp t'. However, since the list is sorted and the algorithm did not find f earlier, we can conclude that no such earlier infection time exists.

2.6 Dijkstra's Algorithm

2.6.1 Definition

BFS only works to find shortest distances when every edge has the same weight. To find the shortest path on weighted graphs, we can use Dijkstra's. Dijkstra's uses a priority queue to get the lowest dist(v) value.
Pseudocode:

```
procedure GraphSearch(Dijkstra's):
Input: $G(V, E)$, $s$
Initialize $X$ = empty, $F = \{s\}$
Initialize dist($v$) = $\infty$ for all $v$
Initialize dist($s$) = 0
while $F$ is not empty:
    Pick the $v$ in $F$ that has the lowest dist($v$) value
    for each neighbor $u$ of $v$:
    if dist($u$) > dist($v$) + $l(u, v)$
        move $u$ to $F$
        set dist($u$) = dist($v$) + $l(u, v)$
    Move $v$ from $F$ to $X$
```

2.6.2 Correctness

Claim: After dijkstra is done, dist(v) is the length of the shortest path from s to v.
Suppose for some vertex v, there is a path p from s to v such that length(p) ¡ dist(v).
After BFS is done, dist(v) is the length of the shortest path from s to v for all vertices $v \in V$
Suppose for some vertex v, that there is a path p from s to v such that length(p) $<$ dist(v)
(Let $d(v)$ be the actual length of the shortest path from s to v)
Then let z be the last vertex in the path s.t. dist(z) $= d(z)$ and let w be the first vertex in the path s.t. dist(w) $> d(w)$
Case 1: $d(w) = d(z) + l(z, w)$. Then in the algorithm, when z is chosen from F, we will find that $d(w) > d(z) + l(z, w)$. So reset dist(w) $= d(z) + l(z, w)$. So dist(w) $= d(w)$

Case 2:
$d(w) < d(z) + l(z,w)$. Then the path p is actually not the shortest path (because i can use the shortcut through w)

2.6.3 Implementing Priority Queue: Heap vs. HashTable

Hash Table
Indexed by vertex, giving key value (O(B)=2)
deletemin: (O(V)) (need to find minimum)
decreasekey: (O(1))
Total runtime:
$|V| \cdot \text{deletemin} + |E| \cdot \text{decreasekey}$
$|V| \cdot O(V) + |E| \cdot O(1) = O(|V^2|)$
Binary Heap:
Represented with an array. Swap root/rightmost child, then delete root (pop its value) and trickle new root down (same heap we've been working with).
deletemin: $O(\log(|V|))$
decreasekey: $O(\log(V))$
Total runtime:
$O(\log(|V|)) \cdot |V| + O(\log(V)) \cdot |E|$
Runtime Comparison:

	Array $(O(|V|^2))$	Priority Queue $(O(|V| + |E|)log(|V|))$
Sparse Graphs $|E| = \Theta(|V|)$	$O(|V^2|)$	$O(|v|log(v))$
Dense Graphs $|E| = \Theta(|V^2|)$	$O(V^2)$	$O(|V|^2 \log |V|)$

2.6.4 Reduction Example

Problem Statement: You are the manager of a train station network (connections connect two stations and trains travel both ways). Some stations have been broken. You only have the budget to fix one. Since the route from uptown (Station A) and downtown (Station B) is the most popular route, you wish to fix the station that results in the shortest journey time between stations A and B. How do you find this station?

Psuedocode:

```
procedure findPath
Input: $G(V,E), s$ (starting station), $t$ (ending station)
    Create a graph G' as follows. For each functioning station, create two copies $f_1$ and $f_2$. For each broken station, create a vertex $d_1$.
    for every edge $(v,u)$ of $E$:
        if v is functioning and u is functioning, add edges $(v_1, u_1)$ and $(v_2, u_2)$.
        if v is damaged and u is functioning, add edges $(v_1, u_2)$
    Run GraphSearch(Dijkstra's) on G' starting from $s_1$.
    if $dist(t_2) > dist(t_1)$
        return prev($t_1$)
    else
        currPrev := prev($t_1$)
```

```
while currPrev is not broken:
    currPrev = prev(currPrev)
return currPrev
```

Correctness Proof

Case 1: Fixing a single broken station will improve travel time between station A and B, so we return the broken station

Fixing a single broken station will improve travel time if the shortest path distance to b_2 (the station b node in the second section of G') is shorter than the path to b_1 (the station b node in the first section of G'). The path that goes to b_2 contains only one broken station since in our graph construction, there are only outgoing edges from broken stations to the second part of our graph. This path only has 1 broken station since the second section only contains functional station nodes and we ignore all edges between 2 broken stations. Thus, if we need to fix a single broken station to improve travel itme between station A and B, our algorithm correctly returns the broken station.

Case 2: Fixing a broken station will not improve travel time between station A and B so we return a random station

Contrapositive: If we return the broken station, then fixing a single broken station will improve the travel time between A and B.

If we return the broken station, then the algorithm has determined that the shortest path distance to b_2 is shorter than the shortest path distance to b_1. Dijkstra's will correctly return the shortest path distances to all nodes reachable from a_1. The only way dist($b_2 < \infty$) is if there is a path from A to B such that there is a broken station K with functional nodes from the first part of the graph in the path between A and K and functional nodes from the second part of the graph in the path between K and B. The only way dist($b_1 < \infty$) is if there is a path from A to B such that there is only functional nodes from the first part graph in the path between A and B. If $dist(b_2) < dist(b_1)$, then the path where one broken station K is fixed is shorter than the path through only functional stations to b_1. If there is one broken station K between A and B, then by the correctness of dijkastra's, we can follow the prev pointers starting from b_2 to find some station K which is after a_1. Thus, if the algorithm returns the broken station, then the shortest path distance is to b_2, not b_1, which means that fixing a single broken station will improve the travel time between A and B.

Runtime Analysis

Graph creation has a worst case of $O(2|V|+2|E|)$=$O(V+E)$. Djkstra's will take $O(2|V|+2|E|) = O(V+E)$ worst case. Lastly, the worst case for returning the broken station is $O(v-1) = O(v)$. Thus, the algorithm's runtime is $O(|V| + |E|)$

2.7 Minimum Spanning Tree

Spanning Trees: A spanning tree of an undirected graph $G = (V, E)$ is a subgraph $G'(V, E')$ such that G' is a tree and all vertices in V are connected.

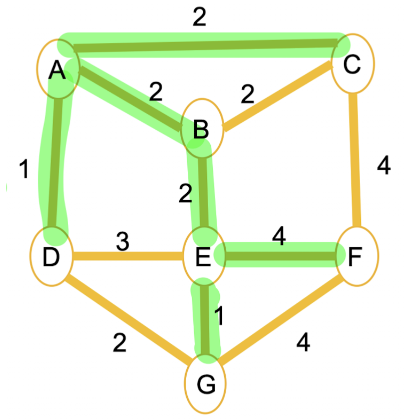
2.7.1 Cut Property

- Suppose $G` = (V, T)$ is a MST of $G = (V, E)$ and suppose that $X \subseteq T$
- Pick any subset of vertices $S \subseteq V$ such that there is no path from S to $V - S$ using edges from X
- Let $e \in E$ be the lightest edge that connect S to $V - S$
- Then $X \cup \{e\}$ is part of some MST

Proof of cut property

- **Case 1:** $e \in T$
  Then, $X \cup \{e\} \subseteq T$ and we assumed that T is the set of edges of an MST so $X \cup \{e\}$ is part of that MST
- **Case 2:** $e \notin T$
  Consider $T \cup \{e\}$. Since T is the edge set of a connected tree, $T \cup \{e\}$ is the edgeset of a graph that has a cycle which contains e. There must be another edge $e' \in T$ that connects S to $V - S$
  - $w(e) \leq w(e')$
  - $cost(T \cup \{e\} - \{e'\}) = cost(T) + w(e) - w(e')$
  - $cost(T \cup \{e\} - \{e'\}) \leq cost(T)$
  - T is the edgeset of an MST so it is minimal, therefore $cost(T \cup \{e\} - \{e'\})$ must also be minimal
  - $cost(T \cup \{e\} - \{e'\}) = cost(T)$ so $T \cup \{e\} - \{e'\}$ is the edgeset of an MST
  - $X \cup \{e\} \subseteq T \cup \{e\} - \{e'\}$

2.7.2 Kruskal's Algorithm



- Definition
  1. Start with a graph with only vertices
  2. Repeatedly add the next lightest edge that does not form a cycle

  DSDS: Disjoint Sets Data Structure manages partitioning the set into disjoint subsets

  - Makeset(S): puts each element of S into a set by itself

- Find(u): returns the name of the subset containing u
- Union(u,v): unions the set containing u with the set containing v

Kruskal's algorithm using DSDS

```
procedure Kruskal(G,w):
Input: undirected graph $G$ with edge weights $w$
Output: A set of edges $X$ that define the MST for all $v \in V$:
        Makeset(v)
$X = \{\}$
sort the set of edges $E$ in increasing order by weight
for all edges $(u, v) \in E$ until $|X| = |V| - 1$
        if find(u) $\neq$ find(v)
            add($u$,$v$) to $X$
            union($u$,$v$)
```

DSDS Version 1

- Keep an array Leader(U) indexed by element
- In each array position, keep the leader of its set
- Makeset(v): Initialize to self, O(1)
- Find(u): return Leader(u), O(1)
- union(u,v): For each array position, if it's currently Leader(v) change it to Leader(u), O(—V—)

Sorted Edges: (A,D)=1,(E,G)=1,(A,B)=2,(A,C)=2,(B,C)=2,(B,E)=2,(D,G)=2,(D,E)=3,(E,F)=4,(F,G)=4

A	B	C	D	E	F	G
A	B	C	D	E	F	G
D	B	C	D	E	F	G
D	B	C	D	G	F	G
B	B	C	B	G	F	G
C	C	C	C	G	F	G
G	G	G	G	G	F	G
F	F	F	F	F	F	F

Union-By-Rank Version 2 Subroutines:

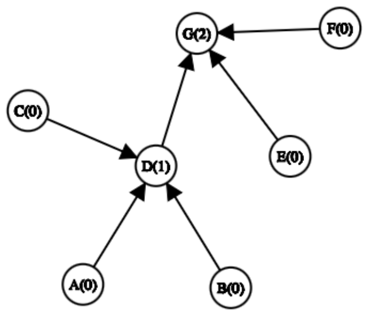
```
procedure makeset(x):
runtime: O(1)
$\pi(x) := x$
$rank(x) := 0$
```

```
procedure find(x):
runtime: O(height of tree containing x)
while $x \neq \pi(x)$:
        $x := \pi(x)$
```

```
return $x$

procedure union(x,y):
runtime: O(find)
$rx := find(x)$
$ry := find(y)$
if $rx = ry$ then return
if $rank(rx) > rank(ry)$ then
        $\pi(ry) := rx$
else
        $\pi(rx) := ry$
            if $rank(rx) = rank(ry)$ then
                $rank(ry) := rank(rx) + 1$
```

- Each set is a rooted tree, with the vertices of the tree labelled with the elements of the set and the root the leader of the set
- Only need to go up to leader, so just need parent pointer
- The root points to itself
- The root is a representation of the set containing it and all of its children
- Information in the data structure is stores in two arrays
  * $\pi(v)$: the parent pointer
  * rank(v): the height of the tree hanging from v

Union-By-Rank Graph



Graph Notes:

- If a vertex has rank k, the height of its tree is k

    - Any root of rank k has at least 2^k vertices in its tree
    - Maximum height of tree with n vertices is $log(n)$ since vertex with rank $log(n)$ has at least n vertices

- **Runtime of Kruskal's:** $|V|makeset + 2|E|find + (|V|-1)union + sort(|E|)$
    - **Version 1:** $|V|O(1) + 2|E|O(1) + (|V|-1)O(|V|) + sort(|E|) = O(|V|^2 + |E|\log|E|)$
    - **Version 2:** $|V|O(1) + 2|E|O(log(|V|)) + (|V|-1)O(log(|V|)) + |E|log(|E|) = O(|E|log(|V|))$

2.7.3 Prim's Algorithm

- Definition Based on graphsearch. Creates a MST by picking the next lightest edge that doesn't create a cycle.

  Pseudocode:

```
procedure Prim's:
Pick a random vertex $s$
Initialize $X$ = empty, $F = \{s\}$
Initialize cost($v$) = $\infty$ for all v
Initialize prev($s$) = null
Initialize output = null
while F is not empty:
        Pick the $v$ that has the lowest cost($v$) value
        for each neighbor $u$ of $v$:
            if cost($u$) > $l(v, u)$
                move $u$ to $F$
               set cost($u$) = $l(v, u)$
               set prev($u$) = $v$
        Move $v$ from $F$ to $X$
        Move $(v, prev(v))$ into output.
```

- Runtime Worst case: same as dijkstra's, but since the input must be a connected graph, $|E| = \Omega(|V|)$

3 Greedy Algorithms

3.1 Exchange Argument

3.1.1 Exchange Argument Template

ExArg: Let g be the first greedy decision. Let OS be any legal solution that does not pick g. Then there is a solution OS' that does pick g and S is at least as good as OS.
Prove by strong induction on instance size the GS is optimal
Induction Step:

1. Let g be the first greedy decision. Let I' be the rest of problem given g

2. $GS = g + GS(I')$

3. OS is an legal solution
4. OS' is defined from OS by the exchange argument
5. $OS' = g +$ some solution on I'
6. Induction: $GS(I')$ at least as good as any solution on I'
7. GS is at least as good as OS', which is at least as good as OS

3.1.2 Exchange Argument Example

Suppose you have a pile of n socks that are all different shades of gray (potentially ranging from 0% meaning pure white up to 100% meaning pure black.)
A pair of socks is considered passable if their colors differ by a small percentage threshold $T\%$
Input: List $L = (c_1, \dots, c_n)$ of n greyscale values and a positive threshold T
Solution Format: Set S of pairs chosen from L
Constraints: Each pair (c_1, c_2) must have the property that $|c_1 - c_2| \leq T$
Objective: Maximize the number of pairs
Greedy Strategy: Sort the socks in increasing order by color: $(c_1, c_2, \dots, c_n)$

- If $|c_1 - c_2| \leq T$ then pair (c_1, c_2) and recurse on the rest of the socks $(c_3, \dots, c_n)$
- If $|c_1 - c_2| > T$ then throw away c_1 and recurse on the rest of the socks $(c_2, \dots, c_n)$

ExArg Claim: For some input $I = (c_1, \dots, c_n)$, assuming that $c_2 - c_1 \leq T$, let OS be some solution that doesn't pair (c_1, c_2). Then there is a solution OS' that does pair (c_1, c_2) and has at least as many pairs as OS
Proof of ExArg: Suppose that OS is a solution of I that doesn't pair (c_1, c_2). Consider the following 3 cases.

1. c_1 and c_2 are both unpaired in OS. Then create OS' by adding the pair (c_1, c_2) to OS.
   OS' is valid since OS was valid and did not pair c_1,c_2 and $c_2 - c_1 \leq T$, then (c_1, c_2) is a valid pair so OS' is valid. OS' is just as good as OS since there is one more pair in OS' than OS so $|OS'| > |OS|$
2. c_1 is paired with c_j and c_2 is unpaired (or c_2 is paired with c_j and c_1 is unpaired). Then create OS' from OS by removing the pair (c_1, c_j) and replacing it with (c_1, c_2)
   OS' is valid since OS was valid and $c_2 - c_1 \leq T$, then (c_1, c_2) is a valid pair so OS' is valid. OS' is just as good as OS since there is the same number of pairs in OS' as OS so $|OS'| = |OS|$
3. c_1 is paired with c_j and c_2 is paired with c_i. Then create OS' from OS by removing the pairs $(c_1, c_j), (c_2, c_i)$ and replacing them with $(c_1, c_2), (c_i, c_j)$
   OS' is valid since $c_2 - c_1 \leq T$, then (c_1, c_2) is a valid pair and we can consider the two subcases for the next pair
   $c_i \leq c_j$: $c_j - c_i \leq c_j - c_1 \leq T$
   $c_i > c_j$: $c_i - c_j \leq c_i - c_2 \leq T$
   OS' is just as good as OS since there is the same number of pairs in OS' as OS so $|OS'| = |OS|$

Induction Part:
Claim: For any input of any size $n \geq 2$, the greedy solution is optimal
Base Case: For $n = 2$, then if the two socks can be paired, the greedy strategy pairs them
Induction Hypothesis: Suppose that for some $n > 2$ the greedy is optimal for all inputs of size k such that $2 \leq k \leq n-1$
Induction Step: Suppose $I = (c_1, \ldots, c_n)$ is an input of size n ordered by color then let OS be any solution OS' is just as good as OS since there is the same number of pairs in OS' as OS so $|OS'| = |OS|$
Case 1: suppose that $c_2 - c_1 > T$, then if c_2 can't pair with c_1, then no sock can pair with c_1 because all other socks are equal to or darker than c_2 so we can consider the subproblem without c_1 with $n-1$ socks. By the inductive hypothesis since there are fewer than n socks, the greedy strategy is optimal.
Case 2: suppose that $c_2 - c_1 \leq T$, then let OS be any solution. Then by the exchange argument, there exists a solution OS' that includes (c_1, c_2) and has at lest as many pairs as OS. Therefore we have that $|OS(I)| \leq |OS'(I)| = |\{(c_1, c_2)\} \cup S(I')| \leq |\{(c_1, c_2)\} \cup GS(I')| = |GS(I)|$

3.2 Greedy Stays Ahead

Methodology:

1. Start with arbitrary solution OS and compare all of greedy solution GS to all of OS (instead of just the greedy choice)
2. Induct on greedy choices instead of size

Example:

1. Consider an input I with n with n events
2. Let $OS(I)$ be some arbitrary set of non-conflicting events ($OS(I) = [J_1, J_2, \ldots, J_k]$
3. Let $GS(I)$ be the outcome of the greedy strategy $GS(I) = [G_1, G_2, \ldots, G_l]$
4. WTS $k \leq L$
5. Proof:
   (a) **Claim:** GS "stays ahead" of OS
   (b) Finish(G_1) $\leq$ Finish(J_i) for all $i \geq 1$ assume that Finish(G_i) $\leq$ Finish(J_i)
   (c) Inductive step: want to show that Finish(G_{i+1}) $\leq$ Finish($J_i + 1$)
   Proof by contradiction:
   Suppose that $k > l$ where $|OS| = k, |GS| = l$ (OS has more greedy choices than GS)
   Then G_l is the last greedy choice so that there are no events that start after G_l finishes
   Then Finish(G_l) $\leq$ Finish(J_l) and Finish(J_l) $\leq$ Start(J_{l+1}). This implies there is an event J_{l+1}

that starts after the last greedy choice G_l, which is impossible

3.3 Greedy Achieves the Bound

Greedy achieves the bound algorithms are useful for when we want to show the cost of a greedy solution is lower than an abritrary one (we show the greedy solution reveals a lower bound). **Note this means they only work for certain types of algorithms**
Methodology:
Example:
Problem: Let t be a certain time at the conference and $B(t)$ be the set of all events happening at t. Let R be the number of rooms in a valid schedule. **WTS** $R \geq |B(T)|$ for all t.

1. **Lower bound:** Let $L = \max_t |B(t)|$

2. **Proof:**
   Claim: At some point $k = |B(t)|$
   Let t be the starting time of the first event to be scheduled in room k.
   Then room k was the minimum room number available at the time.
   This means at time t, there were events going on in rooms $1, 2, \ldots, k-1$ plus the event in room k
   Therefore $|B(t)| = k$ at some point t
   Therefore, $k \leq \max_t(|B(t)|) = L$
   And the greedy solution achieves the optimal bound L.

3. **Conclusion:**
   Let GS be the greedy solution, $k = Cost(GS)$ the number of rooms used in the greedy solution
   Let OS be any schedule. $R = Cost(OS)$ the number of rooms used in OS
   By bounding lemma, $R \geq \max_t(|B(t)|) = L$
   By the achieve the bounds lemma, $k = |B(t)| \leq L$ for some t.
   Thus, $Cost(GS) = k \leq L \leq R \leq Cost(OS)$, and the greedy solution is optimal

4 Divide and Conquer

4.1 Master Theorem

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

- **a**: The number of recursive calls
- **b**: Fraction of the original input size of recursive calls
- **d**: Degree of polynomial of the number of non recursive part

$O(n^d)$	if $a < b^d$
$O(n^d logn$	if $a = b^d$
$O(n^{log_b a}$	if $a > b^d$

4.1.1 Characteristic Polynomial

Guess the polynomial: $DT(n) = Ar^n$
Eliminate:
If recurrence relation is $DT(n) = DT(n-1) + DT(n-2)$, we replace the recurring terms with a polynomial such that $Ar^n = Ar^{n-1} + Ar^{n-2}$, then simplify so that it become $r^2 = r + 1$
Solve the roots:
In this case, the roots are $\frac{1\pm\sqrt{5}}{2}$.
Write a characteristic polynomial:
We can rewrite the recurrence relation as $DT(n) = A_1 x_1^n + \cdots + A_k x_k^n$ when the polynomial has k roots.

4.1.2 Solving Recurrences

1. **Guess and Check**: Start with small values of n and look for a pattern. Confirm guess and check with proof by induction.

2. **Unravel**: Start with the general recurrence and keep replacing n with smaller input values. Keep unraveling until you reach the base case.

3. **Characteristic Polynomial**: If the recursion is of a certain form, you can guess that the closed form is Cw^n and solve for w by finding the root of the polynomial. (Note: better for finding a bound rather than closed form)

```
MultiplyKS function multiplyKS(x,y):        Input: n-bit integers x and y
        Output: the product xy
        $x_l$, $x_r$, and $y_L$, $y_r$ are the left-most and right-most $\frac{n}{2}$ bits of a $x$ and $y$ respectively
        $R_1 = multiplyKS(x_l, y_l)$
        $R_2 = multiplyKS(x_r, y_r)$
        $R_3 = multiplyKS((x_l + x_r)(y_l + y_r))$
return $R_1 \cdot 2^n + (R_3 - R_1 - R_2) * 2^{\frac{n}{2}} + R_2$
```

Let $T_{KS}(n)$ be the runtime
Then $T_{KS}(n) = 3T_K S(\frac{n}{2}) + O(n)$
Runtime analysis:
$a = 3$, $b = 2$, $d = 1$, so $a > b^d$ and $T_k(n) = O(n^{\log_2(3)}) = O(n^{1.585})$

```
Binary Search function Binary Search
Input: $(a_1, \ldots, a_n; x)$
        $lo = 1$
        $hi = n$
       while($lo \leq hi$):
          $m = \lceil \frac{lo+hi}{2} \rceil$
           if $x == a_m$ :
              return m
        **if** $x < a_m$ :
            $h = m - 1$
        **if** $x > a_m$ :
            $lo = m + 1$
    **return** 0
```

Runtime: $T(n) = T(\frac{n}{2}) + O(1)$
$a = 1, b = 2, d = 0$
$a = b^d$
$T(n) = O(\log_2(n))$

4.2 Selection and Sorting

4.2.1 Selection (quickselect)

Input: list of integeres and integer k
Output: the kth smallest number in the set of integers

```
function QuickSelect($a[1 \ldots n], k$)
  if $n == 1$:
    return $a[1]$
  pick a random integer $v$ from the list
  split the list into sets $S_L$,$S_V$,$S_R$
  if $k \leq |S_L|$:
    return QuickSelect($S_L$,$k$)
  if $k \leq |S_L| + |S_V|$:
    return $v$
  else:
    return QuickSelect($S_R$,$k - |S_L| - |S_V|$)
```

Runtime:
Splitting list: cn
Recursive step: $T(L)$ or $T(R)$
If we're lucky and always choose the median $|SR| \approx |SL| \approx \frac{n}{2}$
$T(n) = T(\frac{n}{2}) + O(n)$, and by master theorem $T(n) = O(n)$

4.2.2 Selection(median of medians)

Split the list into sets of 5 and find the medians of all those sets then find the median of the medians using a recursive call $T(n/5)$. Then partition the set like quickselect and recurse on S_R or S_L just like in quickselect.

```
function MofM($L, k$)
  If $L$ has 10 or fewer elements:
```

```
  Sort(L) and return the $kth$ element
Partition $L$ into sublists $S[i]$ of five elements each
For $i = 1, \dots \frac{n}{5}$
  $m[i] = MofM(S[i], 3)$
$M = MofM([m[1], \dots, m[\frac{n}{5}]], \frac{n}{10})$
Split the list into sets $S_L$,$S_M$,$S_R$
if $k \leq |S_L|$:
  return MofM($S_L, k$)
if $k \leq |S_L| + |S_M|$:
  return M
else:
  return MofM($S_R, k - |S_L| - |S_M|$)
```

Runtime:
$|SR| < \frac{7n}{10}$ and $|SL| < \frac{7n}{10}$. Can't use master theorem, so have to use induction. In the end, this algorithm is $O(n)$.

4.2.3 Sorting Lower Bound Runtime

Any sorting algorithm that relies on comparisons between elements runs in $\Omega(log(n!))$ and $log(n!) = \theta(nlogn)$.

4.2.4 Sorting (Mergesort)

```
function mergesort($a[1 \dots n]$)
  if $n > 1$:
    $ML$ = mergesort($a[1 \dots \lfloor \frac{n}{2} \rfloor]$)
    $MR$ = mergesort($a[\lfloor \frac{n}{2} \rfloor + 1 \dots n]$)
    return merge($ML, MR$)
  else:
    return $a$
```

MergeSort Correctness
Base Case: $n = 1$, mergessort returns the original array a which is trivally sorted.
Inductive Hypothesis: Suppose that for some $n > 1$, mergesort($a[1 \dots k]$) outputs the elements of a in sorted order on all inputs of size k where $1 \leq k < n$.
WTS: mergesort works for inputs of size n
Inductive Step: Since $n > 1$, mergesort($a[1 \dots n]$) returns merge(ML, MR) where ML = mergesort($a[1 \dots \lfloor \frac{n}{2} \rfloor]$) and MR = mergesort($a[\lfloor \frac{n}{2} \rfloor + 1 \dots n]$). Since $\lfloor \frac{n}{2} \rfloor < n$, the inductive hypothesis ensures that ML and MR are sorted and merge combines two sorted lists so the algorithm returns the elements in sorted order.
Runtime:
Each recursive call runs in $T(\frac{n}{2})$ and merge runs in $O(n)$ time so the recurrence relation can be defined by $T(n) = 2T(\frac{n}{2}) + O(n)$. $T(n) = O(nlogn)$ by the master theorem.

4.2.5 Sorting (Quicksort)

```
function quicksort($a[1 \dots n]$):
  if $n == 1$:
    return $a_1$
  pick a random index $1 \leq i \leq n$
  partition the list into $S_L$,$S_V$,$S_R$ based on $a_i$
  $L$ = Quicksort($S_L$)
  $R$ = Quicksort($S_R$)
  return $L \circ S_V \circ R$
```

Runtime:

Partition takes $O(n)$ and we have two recursve calls of size $|S_L|$ and $|S_R|$ and the final step of combining the sets takes $O(n)$.

Our recurrence relation can be defined by $T(n) = T(S_L) + T(S_R) + O(n)$ which can't be computed using the master theorem so we need to find the expected runtime which is $O(nlogn)$.

4.2.6 Deterministic vs Randomized

	Deterministic	Randomized
Sorting	**Mergesort:**$O(nlogn)$	**Quicksort:**avg case:$O(nlogn)$/worst case: $O(n^2)$
Selection	**Median of Medians:**$O(n)$	**Quickselect:**avg case:$O(n)$/worst case:$O(n^2)$

5 Backtracking

5.1 Setup

Goal: Reduce problem into smaller constrained set of problems (subproblems which we have deduced we need to solve in order to solve the problem), which we then solve recursively. Backtracking, unlike divide and conquer, just reduces the problem by a constant difference (its like divide and conquer, but we eliminate problems we don't need to solve)

5.2 Maximal independent set

Problem: Given an undirected graph with nodes representing people, and there is an edge between A and B if A and B are enemies, find the largest set such that no people are enemies (largest set of vertices such that no two vertices are connected with an edge)

Setup Local decisions: Picking vertex A

Possible answers: Yes or No

What happens if we pick A?

Recurse on subgraph $GR - \{A \cup \text{A's neighbors}\}$

If we dont pick A recurse on subgraph $GR - A$

Pseudocode:

```
MIS1($G$):
    if $|V| = 0$
        return $\emptyset$
    pick a vertex $v$
    In = MIS1($G - \{v$and v's neighbors$) \cup \{g\}$
    Out = MIS1($G - \{v\}$)
    If|In| > |Out|
        return In
    else
        return Out
```

Correctness proof: **Strong Induction**

Base Case:

$n = 0$. MIS1 correctly returns the empty set

Inductive hypothesis:

Let $n > 0$. MIS1 correctly returns the maximal independent setof any graph with k vertices where $1 \leq k < n$.

In is the max index set containing v

Out is the max index set without v

Better of the two is MIS in g

Runtime analysis:

In and Out have worst cases of $T(n-1)$ (worst case of In is that v has no neighbors)

So, runtime is:

$T(n) \leq 2T(n-1) + O(n)$

$T(n) = O(2^n)$

If vertex v has no neighbors, then In case is always better (or well, its the same), so we dont actually need to consider the out case.

Optimized Pseudocode:

```
MIS1($G$):
    if $|V| = 0$
        return $\emptyset$
    pick a vertex $v$
    In = MIS1($G - \{v$and v's neighbors$) \cup \{g\}$
    If deg($v$) == 0
        return In
    Out = MIS1($G - \{v\}$)
    If|In| > |Out|
        return In
    else
        return Out
```

New runtime

$T(n) = T(n-1) + T(n-2) + O(n)$

Since in certain cases we only consider the In case, the new worst case occurs when v has one neighbor. In this case, we must consider both the In and Out case and only remove 2 vertices from the In subproblem

We arrive at a runtime O($n^{1.62}$)

We can continue to remove worst cases, and further optimize runtime

5.3 Weighted Event Scheduling

- Instance: $[(s_1, f_1, v_1) + \cdots + (s_n, f_n, v_n)]$
- Solution: $s \subseteq \{1, \ldots, n\}$
- Constraints: Any two elements in s don't conflict
- Objective: $\sum_{i \subseteq s} v_i$ (Maximize)

Approach

- Sort the event by end time
- Pick last event I_n which is not necessarily good to include
- Try 2 cases
  - Case 1 (out): We exclude I_n and recurse on $[I_1, \ldots, I_n]$
  - Case 2 (in): We include I_n and recurse on the set of all intervals that don't conflict with I_n

Algorithm

```
**BTWES**($I_1, \ldots, I_n$)(sorted by end times)
    if $n = 0$:
        return 0
    if $n = 1$:
        return $V_1$
    OUT=**BTWES**($I_1, \ldots, I_{n-1}$)
    *Let $I_k$ be the last event to end before $I_n$ starts*
    IN=**BTWES**($I_1, \ldots, I_k$)+value($I_n$)
    return max(OUT,IN)
```

Time Complexity

Worst Case: $T(n) = 2T(n-1) + O(n) = O(2^n)$

This is no better than exhaustive search

Improvement

- We make up to 2^n recursive calls in our algorithm
- Each recursive call has the form $I_1 \ldots I_k$
- There are at most $n+1$ calls throughout
- Memoization: Store and reuse the ansers using a hasmap, don't recompute (Dynamic Programming)

6 Dynamic Programing

6.1 Dynamic Programming Steps

1. Define sub-problems and corresponding array
2. What are the base cases
3. Give recursion for sub problems (case analysis)
4. Order the subproblems
5. What is the final output
6. Put it all together into an iterative algorithm that fills in the array step by step
   For Analysis
7. correctness proof
8. runtime analysis

6.2 Weighted Event Scheduling DP

1. **Original Problem:** Find the max value among all valid schedules of $(I_1, \ldots, I_n)$
   Sub-problem: Let $A[k]$ be the max value among all valid schedules of $(I_1, \ldots, I_k)$
2. $A[0] = 0$ because you can't earn any value from the empty set of events
3. **Case 1:** I_k is not part of the max value schedule ($A[k] = A[k-1]$)
   Case 2: I_k is a part of the max value schedule ($V[k] + A[j-1]$) where I_{j-1} is the last event to end before I_k starts
   $A[k] = max(A[k-1], V[k] + A[j-1])$
4. Since each subproblem is dependent on sub problems of smaller index, we can order sub problems from 0 up to n
5. $A[n]$ is the final output
6. Algorithm

```
**MaxSubset**$(I_1, \ldots, I_n; V_1, \ldots, V_n)$ ordered by end times:
    $A[0] = 0$
    for $k = 1 \ldots n$:
        $j = 1$
        while $End(I_j) \leq start(I_k)$:
            j=j+1
        $IN = V_k + A[j-1]$
        $OUT = A[k-1]$
        $A[k] = max(IN, OUT)$
    return $A[n]$
```

7. Correctness Proof
   Claim: $A[k]$ is the max value out of all valid schedules of $(I_1, \ldots, I_k)$
   Base Case:$A[0] = 0$
   Inductive Hypothesis: $A[k]$ is set correctly for all $0 \leq k < n$ for some $n > 0$
   Inductive Step:
   Case 1: I_n is not in the max value schedule ($A[n] = A[n-1]$)
   Case 2: I_n is in the max value schedule ($A[n] = v[n] + A[j-1]$) where all intervals $I_j \ldots I_{n-1}$ conflict with I_n

8. Runtime Analysis
   Algorithm runs in $O(n^2)$ since the step to find j takes $O(n)$ and the loop runs n times

6.3 String reconstruction

Problem: Given a string of letters with no spaces or punctuation, how would you figure out how to separate the words?

- Subproblems
  $$s[k] = \begin{cases} T & \text{if} x_1, \ldots, x_k \text{ can be separated into english words} \\ F & \text{otherwise} \end{cases}$$

- Base case:
  $s[0] = 0$ (0 isn't a letter)

- Recursion:
  $S[k]$ is true if there exists some $1 \leq j << k$ such that $s(j-1)$ is true and $x[j...k]$ is a word

- Order the problems:
  0 to n

- Output:
  $S[n]$

Psuedoode:

```
StringReconstruction($x[1, ..., n]$)
    Initialize all items in $s$ to be false and all prev to be nil
    $s[0] = T$
    for $k$ from 1...$n$
        $j = k - 1$
        while not $s[k]$ and $j > 0$
            if $s[j] = T$ and $x[j+1, \ldots, k]$
                $s[k] = T$
                prev(k)=$j$
            else
                $j = j - 1$
    if $S(n)$
        $p = n$
        while $p > 0$
            $print(p)$
            $p = prev(p)$
```
Runtime: $O(n^2)$

6.4 Knapsack

x Problem: What is the maximum value you can from a list of items $a[1], \dots, a[n]$ where each item has value $v[i]$ and a weight $w[i]$ given you cannot carry more than weight c?
Make this a decision problem. Did I choose item n or not?

- Define subproblems
  Let $KS(j, b)$ be the maximum value you can fit in a b-capacity knapsack using only items $1, \dots, j$

- Base cases:
  $KS(0, b) = 0$ (no items to steal)
  $KS(j, 0) = 0$ (no capacity)

- Recursion:
  Case 1: item j is a part of the max value knapsack
  $KS[j, b] = v[j] + KS[j, b - w[j]]$
  Case 2: item j is not a part of the max value knapsack
  $KS[j, b] = KS[j - 1, b]$
  We don't know if its better to keep orn ot keep item j, so $KS[j, b]$ =max$(v[j] + KS[j, b - w[j]], KS[j - 1, b])$

- Order of subproblems:
  Start from top-left. Fill rows from left to right, then go down

- Final output:
  $KS[n, c]$ (we will store the max in here)

Psuedocode:
```
$knapsack(w[1, \dots, n], v[1, \dots, n], C)$
$KS[j, 0] = 0$ for all j
$KS[0, b] = 0$ for all b
for $j$ from $1 \dots n$
for $b$ from $1 \dots c$
    if $w[j] > b$:
        $KS[j,b] = KS[j-1,b]$
    else:
        $IN = v[j] + KS[j, b - w[j]]$
        $OUT = KS[j-1,b]$
        $KS[j,b] = max(IN, OUT)$
return $KS[n,c]$
```

6.5 Edit Distance

Problem: How can we keep track of how many changes we need to change one word into another? (e.g. Pelican and Ostrich)
Can brute force or... use DP:
Find them minimum cost to transform word $x[1 \dots n]$ to $y[1 \dots m]$

- define subproblems:
  let $E[i,j]$ be the minimum cost to transform $x[1 \dots i]$ to $y[1 \dots j]$

- base cases:
  $E[0,j] = j$ (if the first word is empty, we just type the word to get x)
  $E[i,0] = i$ (if the second word is empty, we just delete the word to get y)

- Recursion:
  Case 1: $i > j$, so $x[i]$ has a letter and $y[i]$ does not. Then delete $x[i]$. $E[i,j] = E[i-1,j] + 1$
  Case 2: $j > i$, so $x[i]$ has no letter and $y[i]$ does. Then write to $x[i]$. $E[i,j] = E[i,j-1] + 1$
  Case 3: $j == i$, so $x[i]$ has a letter and so does $y[i]$.
  Case 3a: $x[i] == y[j]$. Then $E[i,j] = E[i-1,j-1] + 0$ (no edit required)
  Case 3b: $x[i] \neq y[j]$. Then $E[i,j] = E[i-1,j-1] + 1$ (edit required)

- Ordering: Left to right, top to bottom

- Final output:
  $E[n,m]$

Pseudocode:

```
EditDist($x[1, \dots, n], y[1, \dots, m]$)
    Initialize for $i$ from 1 to $n$, $E[i,0] = 0$ and for $j$ from 0 to $m$, $E[0,j] = 0$
    for $i$ from 1 to $n$
        for $j$ from 1 to $m$
            if $x[i] == x[j]$
            $E[i,j]$ =min$(1 + E[i,j-1], 1 + E[i-1,j], 0 + E[i-1,j-1])$
        if $x[i] \neq x[j]$
            $E[i,j]$ =min$(1 + E[i,j-1], 1 + E[i-1,j], 1 + E[i-1,j-1])$
    Return $[n,m]$
```

6.6 Bookshelf problem

You are a librarian with n books such that book i has a height of $h[i]$ and a width of $w[i]$. You must keep the books in the order they are given in the input. (You cannot assume that they are ordered by height or width.)

You are going to build a bookshelf that has width W. You wish to build the shortest bookshelf of width W that can hold all n books in the order they are given. (assume that $w[i] \leq W$ for all i.)

(the height of a shelf must be at least as tall as the tallest book in that shelf and you can stack as many shelves as you wish.)

(12 points)

Design a DP tabulation algorithm for this problem

1: Define the subproblems:
Let BS(i) be the minimum height bookshelf for i books.

2: Define and evaluate the base cases
BS(0)=0
3: Establish the recurrence for the tabulation.

1. Add ith book to a new shelf
   $BS[i] = BS[i-1] + h[i]$
2. Add ith book to a shelf with previous books
   $BS[i] = BS[j-1] + max(h[j], \ldots, h[i])$ where $j < i$ and $w[j] + \ldots w[i] \leq W$ (look back at each possible value of j)

$BS[i] = min(BS[i-1] + h[i], BS[j-1] + max(h[j], \ldots, h[i]))$

4: Determine the order of subproblems:
Given our base case, we start with the first book (first index). Let i be the current shelf and let k be the number of books (the ones after the last book we have add) we can add until we have filled i. Then, we can either add $0, 1, \ldots, k$ books to i or add the next book to the next shelf (next index).

5: Final form of output. The final result is BS_n, where BS is an array $[BS_0, \ldots, BS_n]$ such that BS_i (for $0 \leq i \leq n$) stores the height of the bookshelf up to shelf i.

6: Put it all together as pseudocode

```
**MinHeight**(widths,heights,W):
    BS = [$\infty$ for i in range(len(widths))]
    BS[0]=0
```

```
for i in range(len(widths)):
    rowWidth = 0
    shelfHeight = 0
    prevIncluded = 0
    while (rowWidth $\leq$ W):
        rowWidth += widths[i-prevIncluded]
        shelfHeight = max(shelfHeight,heights[i-prevIncluded])
        BS[i+1] = min(BS[i+1],BS[i-prevIncluded]+shelfHeight)
return BS[len(widths)]
```

7: Runtime analysis
The runtime is $O(n^2)$ since for each book, we iterate up to n times, and since there are n books, we get a runtime of $O(n * n) = O(n^2)$

6.7 Mario

You are playing a Super Mario board game where there is a sequence of spaces numbered from 0 to n.

You start as Mario-1.

Mario-1 can grow to Mario-k by eating powerups. Mario-k can walk to the next space (space i to space $i+1$) or jump ahead exactly k spaces (space i to space $i+k$).

Each space either has a powerup or a spike.

If Mario-k is on a space with a powerup, you can choose one of three actions: you can stay on the same space and eat a powerup, you can advance one space forward or you can jump k spaces forward.

If you are on a space with a spike, then your Mario shrinks back down to Mario-1. If you land on a spike as Mario-1 then you die.

You start the game as Mario-1 on space 0 and you wish to land exactly on space n in the fewest number of "moves" (one move consists of either a walk or a jump or eating a powerup.)

(12 points) Design a DP-tabulation algorithm that returns the minimum number of moves to get from space 0 as Mario-1 to space n. (you can assume that the input is a binary array $A[0], \ldots, A[n]$ such that $A[i] = 0$ means that the space is a powerup and $A[i] = 1$ means that the space has a spike. Assume that $A[0] = 0$ and $A[n] = 0$. Return ∞ if it is impossible to get to space n.)

1: Define the subproblems:

Let moves[i][k] be the minimum number of moves to get from space 0 to space i as Mario-k for some k greater than 0 but less than n.

2: Define and evaluate the base cases

The base case is when i = 0 and $k = 0$, since it takes 0 moves to get from $i, k = 0$ to $i, k = 0$.

3: Establish the recurrence for the tabulation.

There are two cases, which are when we are at level 0 and level k where $k > 0$. The reason for this is because if mario hits a spike on level 0, mario dies, but if he hits a spike on level k, then he goes back to level 0.

- Case 1: Mario is at level 0
  $\text{moves}[i][0] = \min(\text{moves}[i-1][0]+1, \text{moves}[i-1][k]+1, \text{moves}[i-k][k]+1)$

- Case 2: Mario is at level k, where $k > 0$
  $\text{moves}[i][k] = \min(\text{moves}[i-1][k]+1, \text{moves}[i-k][k]+1, \text{moves[i][k-1]})$

4: Determine the order of subproblems:
Because each index is dependent on smaller indices, we start at index 0 and work up to index n.

5: Final form of output.
$\min(\text{moves}[n])$

6: Put it all together as pseudocode:

```
minSteps(A[0, ..., n]):
    moves = [0, ..., n][0, ..., n] where moves[i,j] = ∞ for all $i, j$
    moves[0][0] = 0
    for $i$ in $1, \ldots, n$:
        for $k$ in $1, \ldots, n$:
            if A[i][k] == 1
                moves[i][0] = ∞
            if k == 0
                if $i > 0$
                    moves[i][0] = min(moves[$i$][0], moves[$i-1$][0] + 1)
            else
                if $i > 0$
                    moves[i][k] = min(moves[i][k], moves[i][k-1]+1)
                    moves[i][k] = min(moves[i][k], moves[i-1][k]+1)
                if $i - k \geq 0$
                    if $A[i-k] == 1$
                        moves[i-k][0] = min(moves[i][0], moves[i-k][k]+1)
                    else
                        moves[i][k] = min(moves[i][k], moves[i-k][k]+1)
                if $i - 1 \geq 0$
                    if $A[i-1] == 1$
                        moves[i][0] = min(moves[i][0], moves[i-1][k]+1)
                    else
                        moves[i][k] = min(moves[i][k], moves[i-1][k]+1)
    minMoves = ∞
    for i in $1, \ldots, n$:
        minMoves = min(minMoves, moves[n][i])
    return minMoves
```

7: Runtime analysis
Because we have to traverse and fill an $n \times n$ array, the runtime is n^2

6.8 Shortest Path With Negative Edge Weights

1. Let $B[i, t]$ be the shortest path from v_0 to v_i using at most t edges
2. $B[0, t] = 0$ and $B[i, 0] = \infty$ (assuming no negative cycles)
3. Recursion
   Case 0: v_0 is the second to last vertex in the path from v_0 to v_i using at most t edges
   Case 1: v_1 is the second to last vertex . . .
   Case 2: v_2 is the second to last vertex . . .

$\vdots$

Case n-1: v_{n-1} is the second to last vertex ...

$$B[i,t] = min \begin{cases} B[0,t-1] + w(v_0, v_i) \\ B[1,t-1] + w(v_1, v_i) \\ \vdots \\ B[n-1,t-1] + w(v_{n-1}, v_i) \end{cases}$$

We can say that $w(u,v) = \infty$ if the edge doesn't exist

4. We can order the t parameter from $0, 1, \dots, n-1$

Preventing Infinite Recursion: If there are no negative cycles, the array values will never improve after t gets bigger than $n-1$

Bellman Ford Algorithm: Given a graph with n vertices labeled $v_0, \dots, v_{n-1}$ (with starting vertex v_0) with (possibly negative) edge weights, return "Negative Cycle" if there is a negative cycle. Otherwise, return the shortest distance from v_0 to all vertices.

Code

```
BFDP($G, v_0$) (Graph with edge weights):
    $B[0,0] = 0$
    $B[i,0] = \infty$ for all $i \neq 0$
    for $t = 1, \dots, n$:
        for $i = 0, \dots, n-1$:
            $B[i,t] = min_{(v_j, v_i) \in E}[B[j,t-1] + w(v_j, v_i)]$
    for $i = 0, \dots, n-1$:
        if $B[i, n-1] \neq B[i,n]$:
            return "Negative Cycle"
    return $[B(0,n), B(1,n), \dots, B(n-1,n)]$
```

Runtime: For a graph G with n vertices and m edges, the algorithm will run n $O(n(n+m))$

Currency Arbitrage: Vertices will represent currencies, edges will represent bids for trade, and edge weight represents exchange rate bid. We would like to find a cycle that has a product bigger than 1. We can map this problem to the shortest path problem by setting each edge weight to $-log(e)$ of itself. The smallest sum of edges in our new graph is the largest product of the original graph.

6.9 Dynamic Programming Trees (Maximum Independent Set)

Find the maximum weight of an independent set of vertices. (no two vertices in a set are connected by an edge)

1. Subproblems
   $M[k] = (IN, OUT)$ where IN is the weight of the maximal independent set of the subtree hanging from k including vertex k and OUT is the weight of the maximal independent set of the sub tree hanging from vertex k excluding k

2. Base Cases
   If v is a leaf then $M[v] = (w(v), 0)$

3. Recursion
   In order to compute $M[k] = (IN, OUT)$
   $IN : w(k) + \sum_i M[c_i(k)].out$
   $OUT : \sum_i (max(M[c_i(k)].IN, M[c_i(k)], OUT))$
4. Order Subproblems: Order subproblems by layers starting from the bottom up
5. Output: $max(M[r].IN, M[r].OUT)$

7 Network Flow

7.1 Linear Programming

Description
Solves optimization problems
Sets constraints and and objective function
Works when all constraints and objective function are all linear equations or inequalities
Constraints limit solution space to polygon or multidimensional polyhedron. Since objective function is linear, then there are no local optimums so global optimums occur at corners

7.2 Network flow

- Instance: directed graph with non-negative edge weights, two verticies (s: source, t: sink)
- Solution format: an assignment of non-negative values to each edge
- At any vertex except s, t total flowing in = total flowing out. Flow along edge cannot exceed capacity of edge
- maximize total flow out of s = total flow into t

Let $f(e)$ be the flow of an edge and $c(e)$ be the capacity of an edge
Then we can change this problem into maximizing the flow of the residual $(ce - f(e)$. We can assign this residual to the reverse edge
Idea: For any flow, keep increasing residual until we cannot maximize it any more. End when there is no path from s to t.
Does FF always terminate? no (e.g. irrational numbers). If all integers from 1 to W, and maintain the invariant that all flows are integers, we should end
Pseudocode:

```
FF(G,c,s,t)
        Repeat until there is no path in the residual graph
            Find a path in the residual graph (e.g. DFS) from s to t
            Augment the flow along every edge in this path
```

Create new residual graph

8 P and NP

8.1 Definition

NP: The set of "easily verifiable" **decision** problems. Class of decision problems where there is an efficient verifier R. R takes in an input the instance x and the possible solution y and returns true if y is a valid solution to x and false otherwise.
P: Class of **decision** problems that have polynomial time algorithms that can solve it
P vs NP: P is a subset of NP but the question on wether $P = NP$ is unknown to everyone except myself.
NP Complete: A decision problem A in NP is said to be NP-Complete if all problems in NP reduce to A (SAT is NP Complete)

- SAT
- 3SAT
- Hamiltonian Path
- Traveling Salesperson (decision)
- Independent Set/Coloring
- 3-coloring

NP Hard: Hardest problems in NP. $NP - Hard \cap NP = NP = Complete$. $NP - Complete \subseteq NP$ and $NP - hard \nsubseteq NP$

8.2 Reductions

- We can view a decision problem as the set if all instances for which the result should be True
- A reduction from decision problem A to decision problem B is a polynomial time function F from the set of instances of A to the set of instances of B so that $x \in A \leftrightarrow F(x) \in B$
- If we have an algorithm Alg_b that can solve B, we can build an algorithm $Alg_a(x) := Alg_b(F(x))$
- If this function F exists, we say that A reduces to B ($A \leq_r B$)

8.3 SAT

Decision problem based on the problem of satisfiability. Given a boolean problem in conjunctive normal form, determine if there exists an assignment of variables that makes the formula true.
$(x \vee y \vee \bar{z}) \wedge (\bar{x} \vee y) \wedge (\bar{y} \vee \bar{z})$
$SAT \in NP$ since verifying an assignment takes polynomial time.

8.4 2SAT

Each clause has 2 literals
$2SAT \in P$
2SAT reduces to SCC

- Create a graph G
- For each literal x create 2 vertices: one for x and one for $\bar{x}$
- For each clause $(x \vee y)$ create two edges $(\bar{x} \rightarrow y)$ and $(\bar{y} \rightarrow x)$
- there is no chance of an assignment that makes the formula true if there is a contradiction
- There is a contradiction if $\bar{x} \rightarrow x$ and $\bar{x} \rightarrow x$
- Run SCC algorithm on this graph. For each variable x is x and $\bar{x}$ are in the same SCC return False, Otherwise return True

$SCC \in P$

8.5 Vertex Cover

Optimization: Given an undirected graph G, find the smallest subset of vertices that cover the edges. In other words, find the smallest subset of vertices such that at least on endpoint of each edge is in the set.
Decision: Given an undirected graph G, is there a vertex cover of G that uses B or fewer vertices?
$(3SAT) \leq_r (VC)$

- Given an instance I of 3SAT, we can turn it into an instance $F(I)$ of vertex cover such that I has a solution in (3SAT) iff $F(I)$ has a solution in (VC)
- Create G
  1. Create an edge $\{x, \bar{x}\}$ for each variable
  2. Create a triangle $\{a, b\}, \{b, c\}, \{c, a\}$ for each clause $(a \vee b \vee c)$
  3. For each literal x in step 1, create an edge to each x in step 2
- Ask if G has a VC with B or fewer vertices where $B = l + 2m$ where l is the number of literals and m is the number of clauses

Vertex Cover $\in NP - Complete$