

INFO 6205 – Program Structure and Algorithms

Assignment 3

Student Name: Vidhi Bharat Patel

Professor: Nik Bear Brown

Instructions:

- Support your approach with algorithms and run through an example if possible.
- Write pseudo codes wherever possible.
- Your code should be a runnable Jupyter notebook.

Q1(10 Points)

Write a code with an algorithm explaining the solution for following problem:

There is a ball in a maze with empty spaces (represented as 'o') and walls (represented as 'x'). The ball can go through the empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the $m \times n$ maze, the ball's start position and the destination, where $\text{start} = [\text{start_row}, \text{start_col}]$ and $\text{destination} = [\text{destination_row}, \text{destination_col}]$, return true if the ball can stop at the destination, otherwise return false. [You may assume that the borders of the maze are all walls]

- To solve this problem, we can use the depth-first search algorithm. Starting from the start position of the ball and try to move it in each direction until it will hit a wall.
- At each new position, we have to call the recursive function again with the new position as the starting position.
- If the destination is reached, it will return true, otherwise we have to backtrack and try another direction.
- The function takes in the maze represented as a 2D list of characters, the starting position as a list of row and column indices, and the destination position as a list of rows and columns. The function returns True if the ball can reach the destination, otherwise False.
- The time complexity of this solution is $O(mn)$, since all the places in the maze needs to be explored. It is also due to the visited set.
- The code can be given as followed: -

```

def ballPath(maze, start, destination):
    m, n = len(maze), len(maze[0])
    visited = set()

    def dfs(x, y):
        if [x, y] == destination:
            return True
        if (x, y) in visited:
            return False
        visited.add((x, y))

        # roll up
        i = x
        while i > 0 and maze[i-1][y] != 'x':
            i -= 1
        if dfs(i, y):
            return True

        # roll down
        i = x
        while i < m-1 and maze[i+1][y] != 'x':
            i += 1
        if dfs(i, y):
            return True

        # roll left
        j = y
        while j > 0 and maze[x][j-1] != 'x':
            j -= 1
        if dfs(x, j):
            return True

        # roll right
        j = y
        while j < n-1 and maze[x][j+1] != 'x':
            j += 1
        if dfs(x, j):
            return True

        return False

    return dfs(start[0], start[1])

```

Q2 (10 points)

According to Wikipedia's article: "The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

The board is made up of an $m \times n$ grid of cells, where each cell has an initial state: live (represented by a 1) or dead (represented by a 0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

- Any live cell with fewer than two live neighbors dies as if caused by under-population. Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies, as if by over-population. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the $m \times n$ grid board, return the next state.

Example 1:

0	1	0		0	0	0
0	0	1		1	0	1
1	1	1	→	0	1	1
0	0	0		0	1	0

Input: board = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]
Output: [[0,0,0],[1,0,1],[0,1,1],[0,1,0]]

Write a code and explain your approach to solving the problem with the help of an algorithm.

- The problem could be solved by iterating through each cell of the board, then determine if it is dead or alive based on its current life status and the life status of its eight neighbors.
- But, this would require an additional board's worth of space. The reason to copy the board is so that we can know the current state of each cell which is impossible if the state is changed in place to a 1 or a 0.
- A different identifier for changed cells can be used to know the state it was when the game started and what would be the state when the game will end.
- The identifiers will remain 0 and 1 for unchanged cells, 2 for cells that were dead but now alive and -1 for cells that were alive but now dead.
- We can make one traversal through the board and change the state of each cell which is now dead or alive based on the current life status and also the status of its eight neighbors.
- Lastly, we can make a second traversal to decode those identifiers into a proper result.

```

class Solution:
    def gameOfLife(self, board: List[List[int]]) -> None:
        m, n = len(board), len(board[0])

        def in_bounds(row:int, col:int) -> bool:
            if row < 0 or col < 0 or row >= m or col >= n:
                return False
            return True

        def count_neighbors(row:int, col:int) -> int:
            neighbors = [(1,0),(0,1),(-1,0),(0,-1),(-1,-1),(-1,1),(1,-1),(1,1)]
            lives = 0
            for neighbor in neighbors:
                next_row, next_col = row+neighbor[0], col+neighbor[1]
                if in_bounds(next_row, next_col):
                    if abs(board[next_row][next_col]) == 1:
                        lives += 1
            return lives

        def dead_or_alive(row:int, col:int, lives:int, now:int) -> int:
            alive = board[row][col] == 1
            if alive and lives < 2:
                return -1
            elif alive and lives > 3:
                return -1
            elif not alive and lives == 3:
                return 2
            else:
                return now

        for row in range(m):
            for col in range(n):
                now = board[row][col]
                lives = count_neighbors(row,col)
                board[row][col] = dead_or_alive(row, col, lives, now)

        for row in range(m):
            for col in range(n):
                if board[row][col] > 0:
                    board[row][col] = 1
                else:
                    board[row][col] = 0

```

Q3 (10 points)

A (5 points): Give an algorithm to find the position of largest element in an array of n numbers using divide-and-conquer. Write down the recurrence relation and find out the asymptotic complexity for this algorithm.

- To find the position of the largest element in an array using divide-and-conquer, the following approach could be followed: -
 1. If the array has only one element, then return its position.
 2. Divide the array into two halves.
 3. Recursively find the position of the largest element in both halves.
 4. Then, compare the largest elements of the two halves.
 5. Return the position of the largest element among the two.
- The function will take an array, a starting index, and an ending index, and it will return the position of the largest element in the array.
- The recurrence relation for this algorithm is $T(n) = 2T(n/2) + O(1)$
- Since the array is divided into two halves and recursively solve each half, and then merge the results in $O(1)$ time.
- Through the Master Theorem, the asymptotic complexity of this algorithm is $O(\log n)$.
- The code is given below:-

```
def find_largest_pos(arr, start, end):  
    if start == end:  
        return start  
  
    mid = (start + end) // 2  
    left_largest_pos = find_largest_pos(arr, start, mid)  
    right_largest_pos = find_largest_pos(arr, mid+1, end)  
  
    if arr[left_largest_pos] >= arr[right_largest_pos]:  
        return left_largest_pos  
    else:  
        return right_largest_pos
```

B (5 points): Give a divide-and-conquer algorithm to find the largest and the smallest elements in an array of n numbers. Write down the recurrence relation and find out the asymptotic complexity for this algorithm.

- Using a divide-and-conquer algorithm to find the largest and smallest elements in an array of n numbers, can be given as followed:-
 1. If the array has only one element, then return that element as both the largest and smallest.
 2. If the array has two elements, then compare them and return the larger and smaller elements respectively.
 3. For n number of elements in the array, divide the array into two halves, then recursively find the largest and smallest elements in each half.
 4. Then compare the largest elements of the two halves and return the maximum as the largest.
 5. Compare the smallest elements of the two halves and return the minimum as the smallest.
- The function will take an array, starting index, and an ending index and then return a tuple containing the smallest and largest elements in the array.
- The recurrence relation for this algorithm is $T(n) = 2T(n/2) + O(1)$.
- The asymptotic complexity of this algorithm is $O(\log n)$.
- The code is given below:-

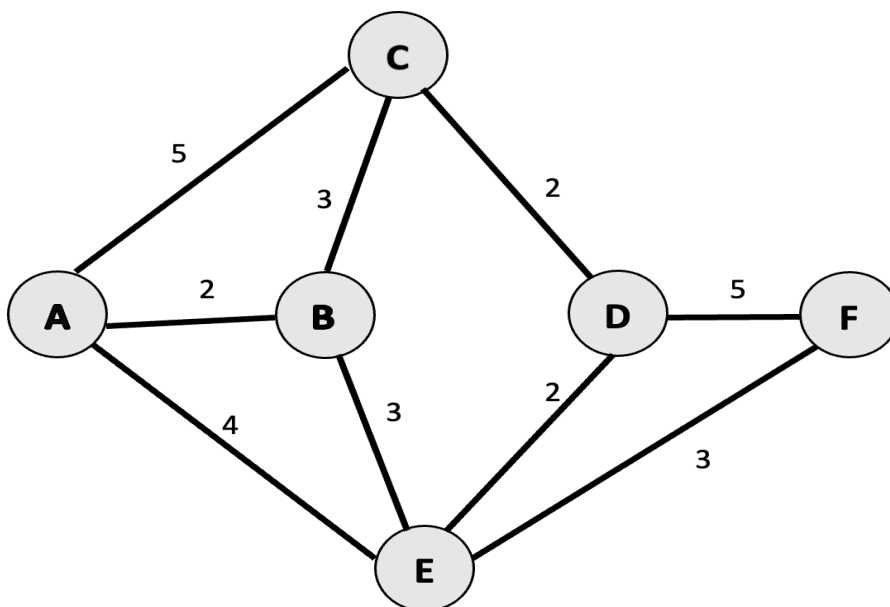
```

def find_min_max(arr, start, end):
    if start == end:
        return (arr[start], arr[start])
    elif start + 1 == end:
        if arr[start] < arr[end]:
            return (arr[start], arr[end])
        else:
            return (arr[end], arr[start])
    else:
        mid = (start + end) // 2
        left_min, left_max = find_min_max(arr, start, mid)
        right_min, right_max = find_min_max(arr, mid+1, end)
        return (min(left_min, right_min), max(left_max, right_max))

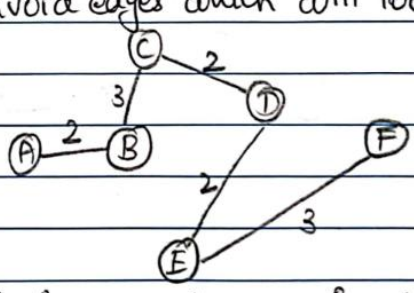
```

Q4 (5 Points)

Use Kruskal's algorithm to find a minimum spanning tree for the connected weighted graph below: What is the Time Complexity of Kruskal's algorithm?



- Kruskal's algorithm to find a minimum spanning tree for the above graph can be given as followed:-
 1. Arrange all the edges in an increasing order of the weights
 2. Then add the edges which has the least weightage
 3. It is important to eliminate the edges which will form a closed circuit or a loop.
- The time complexity of Kruskal's algorithm is $O(E \log E)$, where E is the number of edges in the graph. This is because the algorithm needs to sort all the edges in non-decreasing order of their weights, which takes $O(E \log E)$ time, and then perform a union-find operation on each edge to determine whether to include it in the minimum spanning tree or not.
- The union-find operation also takes $O(\log E)$ time in the worst case, so the total time complexity of Kruskal's algorithm is $O(E \log E + E \log E) = O(E \log E)$.

Q.4] Kruskal's Algorithm.		
Step 1	Arrange all edges in increasing order as	
weight	src	dest
2	A-B	Step 2: add edges which has least weight. avoid edges which will loop.
2	C-D	
2	D-E	
3	B-C	
X 3	B-E	
3	E-F	
X 4	A-E	
X 5	A-C	
X 5	D-F	
∴ This is the min. Spanning tree.		

```

from collections import defaultdict
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        x_root = self.find(x)
        y_root = self.find(y)
        if x_root == y_root:
            return False
        if self.rank[x_root] < self.rank[y_root]:
            self.parent[x_root] = y_root
        elif self.rank[x_root] > self.rank[y_root]:
            self.parent[y_root] = x_root
        else:
            self.parent[y_root] = x_root
            self.rank[x_root] += 1
        return True

    def kruskal(graph):
        vertex_map = {}
        idx = 0
        for u, v, w in graph:
            if u not in vertex_map:
                vertex_map[u] = idx
                idx += 1
            if v not in vertex_map:
                vertex_map[v] = idx
                idx += 1

```

```

# Convert vertices to indices in the graph representation
graph = [(vertex_map[u], vertex_map[v], w) for u, v, w in graph]
# Sort edges by weight
edges = sorted(graph, key=lambda x: x[2])
uf = UnionFind(len(graph))
mst = []
for edge in edges:
    u, v, w = edge
    # Check if edge creates a cycle
    if uf.union(u, v):
        mst.append((list(vertex_map.keys())[list(vertex_map.values()).index(u)],
                    list(vertex_map.keys())[list(vertex_map.values()).index(v)], w))
    if len(mst) == len(vertex_map) - 1:
        break
# total cost of the minimum spanning tree
total_cost = sum(w for u, v, w in mst)
return mst, total_cost

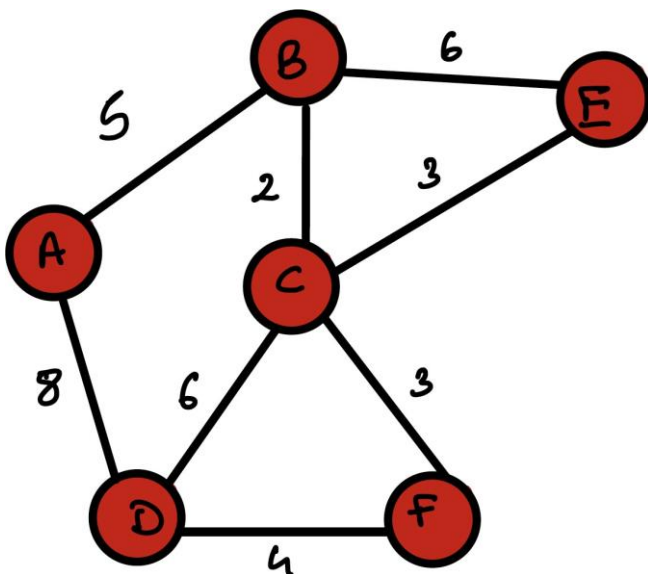
graph = [('A', 'B', 2), ('C', 'D', 2), ('D', 'E', 2),
        ('B', 'C', 3), ('B', 'E', 3), ('E', 'F', 3),
        ('A', 'E', 4), ('A', 'C', 5), ('D', 'F', 5)]
mst, total_cost = kruskal(graph)
print("Minimum Spanning Tree:")
for edge in mst:
    print(edge[0], "-", edge[1], ":", edge[2])
print("Total cost:", total_cost)

```

☞ Minimum Spanning Tree:
 A - B : 2
 C - D : 2
 D - E : 2
 B - C : 3
 E - F : 3
 Total cost: 12

Q5 (5 Points)

Use Prim's algorithm to find a minimum spanning tree for the connected weighted graph below. Show your work. What is the Time Complexity of Prim's algorithm?



- Prim's algorithm to find a minimum spanning tree for the connected weighted graph can be given as:-
 - Choose any vertex from the graph and then choose the shortest edge from it and add it. The vertex I choose is A, then I went ahead choosing B as it was shortest edge than D.
 - Then, choose the nearest edge and add, repeat this process such that the graph does not become a loop or a closed circuit.
 - Add all the weights which will give the cost of the minimum spanning tree.
- The time complexity of Prim's algorithm is $O(E \log V)$, where E is the number of edges and V is the number of vertices in the graph. This is because we use a priority queue (implemented as a heap) to

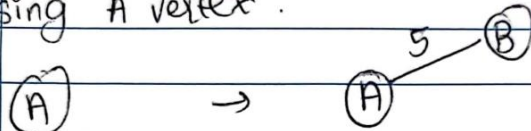
select the next minimum-weight edge to add to the minimum spanning tree, and each edge is added to the heap at most once.

- In a connected graph, the number of edges is at least $V-1$ and at most $V(V-1)/2$, so the time complexity of Prim's algorithm is between $O(V \log V)$ and $O(V^2 \log V)$.

Q.5] Prim's Algorithm.

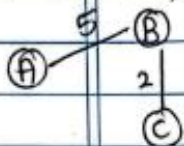
Step 1) Choose a vertex, then choose the shortest edge from it & add.

∴ Choosing A vertex.

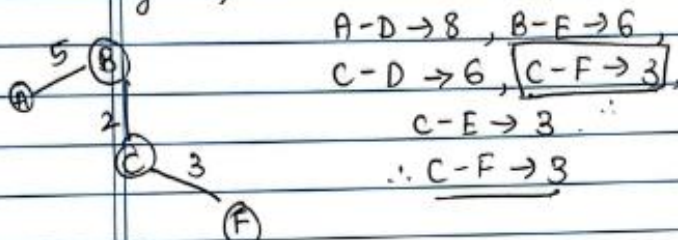


(Since it is shortest edge from A)
 $A-B \rightarrow 5$ ✓
 $A-D \rightarrow 8$

Step 2] choose the nearest edge & add it.
 $B-E \rightarrow 6$, $A-D \rightarrow 8$, ∴ $B-C \rightarrow 2$.

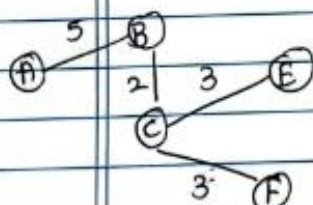


Step 3] choose the nearest vertex with min. weight, if there are multiple same, ^{then} any.



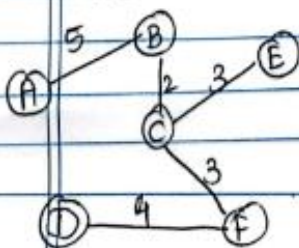
$A-D \rightarrow 8$, $B-E \rightarrow 6$,
 $C-D \rightarrow 6$, $C-F \rightarrow 3$,
 $C-E \rightarrow 3$. ∴
∴ $C-F \rightarrow 3$

Step 4] Repeat it.



$A-D \rightarrow 8$, $C-D \rightarrow 6$,
 $C-E \rightarrow 3$, $F-D \rightarrow 4$,
 $B-E \rightarrow 6$
∴ $C-E \rightarrow 3$

Step 5] Repeat it & select one which does not create a loop or circuit.



∴ $D-F \rightarrow 4$

∴ The total cost of
MST is 17


```

import heapq
def prim(graph, start):
    visited = set()
    mst = []
    total_cost = 0
    pq = [(0, None, start)]
    while pq:
        weight, u, v = heapq.heappop(pq)
        if v not in visited:
            visited.add(v)
            if u is not None:
                mst.append((u, v, weight))
                total_cost += weight
            for neighbor, weight in graph[v]:
                heapq.heappush(pq, (weight, v, neighbor))
    return mst, total_cost
graph = {
    'A': [('B', 5), ('D', 8)],
    'B': [('A', 5), ('C', 2), ('E', 6)],
    'C': [('B', 2), ('E', 3), ('D', 6), ('F', 3)],
    'D': [('A', 8), ('C', 6), ('F', 4)],
    'E': [('B', 6), ('C', 3)],
    'F': [('C', 3), ('D', 4)]
}
mst, total_cost = prim(graph, 'A')
print("Minimum Spanning Tree:")
for u, v, weight in mst:
    print(u, "--", v, ":", weight)
print("Total Cost:", total_cost)

```

Minimum Spanning Tree:

A -- B : 5

B -- C : 2

C -- E : 3

C -- F : 3

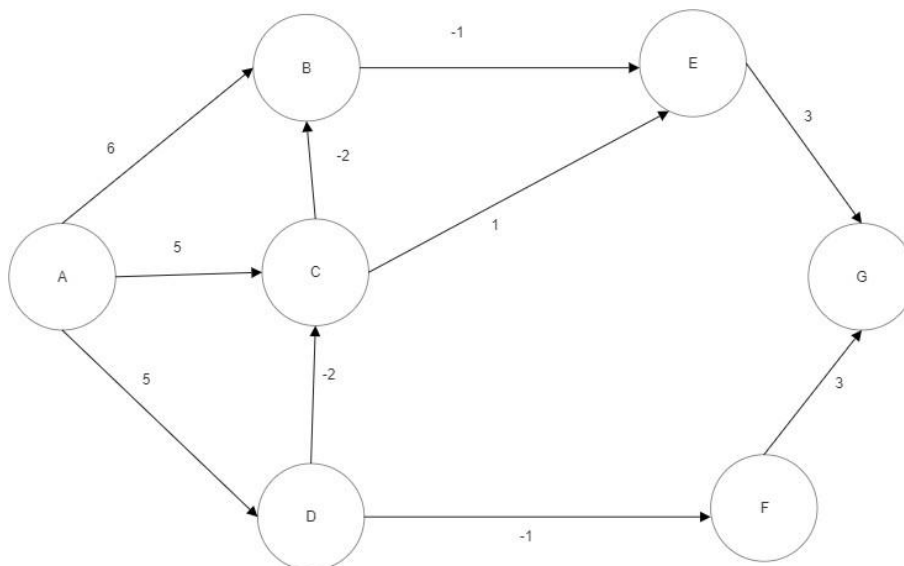
F -- D : 4

Total Cost: 17

Q6(5 Points)

Use the Bellman-Ford algorithm to find the shortest path from node A to G in the weighted directed graph below. Show your work.

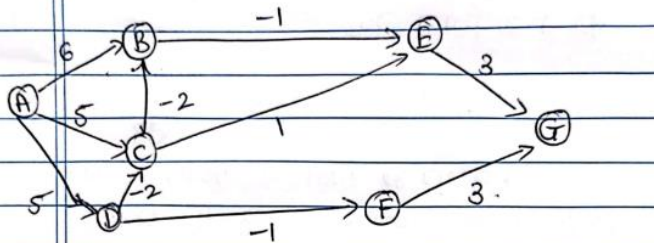
What is the time complexity of Bellman-Ford? And for what condition algorithm fail?



- Bellman Ford algorithm to find the shortest path from node A to G can be given as followed:-
 1. First initialize distances and predecessors
 2. Then relax the edges repeatedly
 3. The final step would be to check for the negative cycles.
- The time complexity of the Bellman-Ford algorithm is $O(|V||E|)$, where $|V|$ is the number of vertices (nodes) in the graph and $|E|$ is the number of edges. This is because in the worst case, we need to relax each edge in the graph $|V|-1$ times, and there can be at most $|V||E|$ edges in a directed graph.

- The algorithm will fail if there is a negative-weight cycle in the graph. A negative-weight cycle is a cycle in the graph where the sum of the weights of the edges is negative. In this case, there is no shortest path from the source node to any other node in the cycle, because we can keep going around the cycle to make the path shorter and shorter. The Bellman-Ford algorithm will continue to try to relax the edges in the cycle, causing the distances to keep decreasing without converging to a solution.

Q 6) Bellman - Ford



A	B	C	D	E	F	G
0	∞	∞	∞	∞	∞	∞
0	6	5	5	0	∞	∞
0	1	3	5	0	4	3
0	1	3	5	0	4	3

The path will be A-D-C-B-E-G

\therefore The shortest path = 3

```
def bellman_ford(graph, start):
    # Step 1: Initialize distances and predecessors
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    # Step 2: Relax edges repeatedly
    for _ in range(len(graph) - 1):
        for node in graph:
            for neighbor, weight in graph[node].items():
                new_distance = distances[node] + weight
                if new_distance < distances[neighbor]:
                    distances[neighbor] = new_distance

    # Step 3: Check for negative cycles
    for node in graph:
        for neighbor, weight in graph[node].items():
            if distances[node] + weight < distances[neighbor]:
                raise ValueError('Negative cycle detected')

    return distances, distances['G']

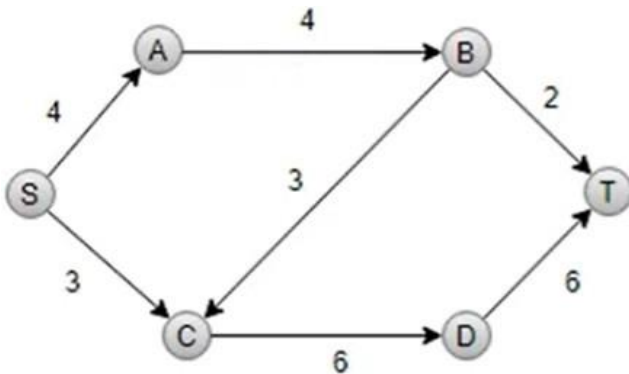
graph = {
    'A': {'B': 6, 'C': 5, 'D': 5, 'E': 0},
    'B': {'E': -1},
    'C': {'B': -2, 'E': 1},
    'D': {'C': -2, 'F': -1},
    'E': {'G': 3},
    'F': {'G': 3},
    'G': {}
}

distances = bellman_ford(graph, 'A')
print(distances)
```

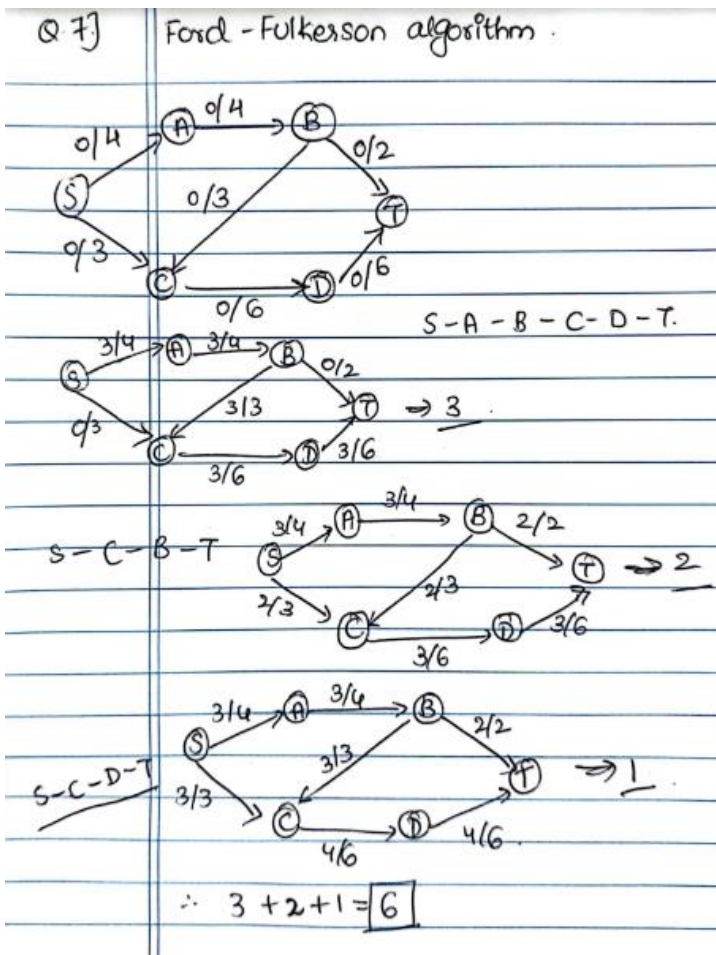
↳ ({'A': 0, 'B': 1, 'C': 3, 'D': 5, 'E': 0, 'F': 4, 'G': 3}, 3)

Q7(5 Points)

Use the Ford-Fulkerson algorithm to find the maximum flow from node S to T in the weighted directed graph below. Show your work.



- Ford-Fulkerson algorithm to find the maximum flow from node S to T in the weighted directed graph can be given as followed: -
 1. Initialize the flow through each edge to zero.
 2. Choose a path from S to T using any pathfinding algorithm. In this case, we can use a depth-first search algorithm.
 3. Find the minimum capacity of all the edges on the chosen path. This will be the maximum flow that can be pushed through the path.
 4. Update the flow through each edge on the chosen path by adding the minimum capacity found in step 3. Also, update the reverse edges on the path by subtracting the flow from the corresponding forward edges.
 5. Repeat steps 2 to 4 until no more paths can be found from S to T.
 6. The maximum flow from S to T will be the sum of the flows through all the edges leaving S.



Q8(10 Points)

Suppose you live with $n - 1$ other people, at a popular off-campus cooperative apartment, the Ice-Cream, and Rainbows Collective. Over the next n nights, each of you is supposed to cook dinner for the co-op exactly once, so that someone cooks on each of the nights. Of course, everyone has scheduling conflicts with some of the nights (e.g., algorithms exams, Miley concerts, etc.), so deciding who should cook on which night becomes a tricky task. For concreteness, let's label the people, $P = \{p_1, \dots, p_n\}$, the nights, $N = \{n_1, \dots, n_n\}$ and for person p_i , there's a set of nights $S_i = \{n_1, \dots, n_n\}$ when they are not able to cook. A person cannot leave S_i empty. If a person isn't doesn't get scheduled to cook in any of the n nights, they must pay \$200 to hire a cook.

A (5 Points). Express this problem as a maximum flow problem that schedules the maximum number of matches between the people and the nights.

- To express this problem as a maximum flow problem, we can create a graph where each person and each night is a vertex. We will have a source vertex s and a sink vertex t .
- For each person vertex p_i , we will connect an edge from s with capacity 1 (since each person can cook only once) and for each night vertex n_j , we will connect an edge to t with capacity 1 (since we need one person to cook on each night).
- For each person p_i , we will connect an edge to each night vertex n_j that is not in their set S_i (since they cannot cook on those nights). These edges will have capacity 1, since each person can only cook on one night.
- Then, we can run the maximum flow algorithm on this graph to find the maximum number of matches between the people and the nights. Each unit of flow in the graph represents one match between a person and a night.

B (5 Points) Can all n people always be matched with one of the n nights? Prove that it can or cannot.

- No, all n people cannot always be matched with one of the n nights.
- This can happen if there are more than n nights where no one is available to cook. In that case, it will be impossible to find a match for each person on a different night.
- To prove this, suppose that there are k nights where no one is available to cook, where k is greater than n . Then, there are $n-k$ nights available to cook.
- However, each person has a set S_i of nights when they cannot cook, and there are at most $n-1$ people available to cook on each night.
- Therefore, if k is greater than $n-1$, there will be at least one night where no one is available to cook, and not all n people can be matched with one of the n nights.

Q9(10 Points)

In a standard s - t Maximum-Flow Problem, we assume edges have capacities, and there is no limit on how much flow is allowed to pass through a node. In this problem, we consider the variant of the Maximum-Flow and Minimum-Cut problems with node capacities. Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and nonnegative node capacities $\{c_v \geq 0\}$ for each $v \in V$. Given a flow f in this graph, the flow through a node v is defined as $f_{in}(v)$. We say that a flow is feasible if it satisfies the usual flow-conservation constraints and the node-capacity constraints: $f_{in}(v) \leq c_v$ for all nodes. Give a polynomial-time algorithm to find an s - t maximum flow in such a node-capacitated network. Define an s - t cut for node-capacitated networks and show that the analog of the Max-Flow Min-Cut Theorem holds true.

- To find an s - t maximum flow in a node-capacitated network, we can use the following algorithm:
 1. Initialize the flow f to be zero.
 2. While there exists a feasible augmenting path P from s to t in the residual network G_f , do the following: a. Determine the bottleneck capacity $b(P)$ of path P . b. Augment the flow f along path P by $b(P)$.
 3. Return the maximum flow value $|f|$.
- In this algorithm, a feasible augmenting path is a path from s to t in the residual network G_f that satisfies the node-capacity constraints, in addition to the usual flow-conservation constraints. The bottleneck capacity of a path is the minimum capacity of an edge in the path. Augmenting the flow

along path P means adding $b(P)$ units of flow along each edge in the path, in the forward direction and subtracting $b(P)$ units of flow along each edge in the path, in the backward direction.

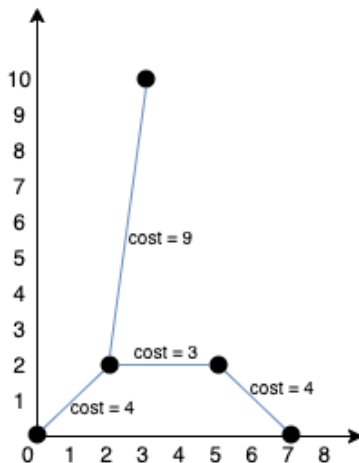
- To define an s - t cut for node-capacitated networks, we consider the cut of the graph induced by the set of nodes reachable from s in the residual network G_f . That is, the cut consists of the set of nodes $\{v \in V: \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$.
- We say that an s - t cut is feasible if it satisfies the node-capacity constraints, that is, the sum of capacities of the nodes on the source side of the cut is at least the value of the flow $|f|$.
- The analog of the Max-Flow Min-Cut Theorem holds true for node-capacitated networks. That is, the maximum value of a feasible flow in a node-capacitated network is equal to the minimum capacity of a feasible s - t cut. The proof of this theorem follows the same argument as in the standard Max-Flow Min-Cut Theorem, by showing that any feasible flow can be used to construct a feasible s - t cut of the same capacity, and any feasible s - t cut can be used to construct a feasible flow of the same value.

Q10(10 Points)

You are given an array of points representing integer coordinates of some points on a 2D-plane, where $\text{points}[i] = [x_i, y_i]$. The cost of connecting two points $[x_i, y_i]$ and $[x_j, y_j]$ is the manhattan distance between them: $|x_i - x_j| + |y_i - y_j|$, where $|val|$ denotes the absolute value of val . Return *the minimum cost to make all points connected*. All points are connected if there is exactly one simple path between any two points. Use an appropriate algorithm to find the minimum cost. Support solution using runnable code.(Language of your choice)

Input: `points = [[0,0],[2,2],[3,10],[5,2],[7,0]]`

Explanation:



- To solve the problem, we can use the Minimum Spanning Tree (MST) algorithm.
- The MST algorithm finds the minimum cost to connect all nodes of a graph.
- In this case, we can consider the given points as nodes of the graph, and the cost of connecting two points as the weight of the edge between them.
- Kruskal's algorithm is a way to find the MST.
 1. To use Kruskal's algorithm, we need to follow these steps:
 2. Sort the edges (pairs of points) by their weights (costs), in non-decreasing order.
 3. Create an empty set of edges called the MST.
 4. Go through the sorted edges and check if adding an edge would create a cycle. If it does not, add it to the MST.
 5. Stop when all nodes are connected.
 6. Calculate and return the sum of the weights of the edges in the MST.
- The code can be given as followed: -

```

def minCostConnectPoints(points):
    def manhattan_distance(p1, p2):
        return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

    edges = []
    for i in range(len(points)):
        for j in range(i+1, len(points)):
            edges.append((i, j, manhattan_distance(points[i], points[j])))
    edges.sort(key=lambda e: e[2])
    parent = list(range(len(points)))

    def find(node):
        if parent[node] != node:
            parent[node] = find(parent[node])
        return parent[node]

    def merge(u, v):
        parent[find(u)] = find(v)

    # Find the MST using Kruskal's algorithm
    mst = []
    for u, v, w in edges:
        if find(u) != find(v):
            mst.append((u, v, w))
            merge(u, v)
            if len(mst) == len(points) - 1:
                break
    return sum(w for u, v, w in mst)

points = [[0,0], [2,2], [3,10], [5,2], [7,0]]
print(minCostConnectPoints(points))

```

20

Q11(5 Points)

Consider using a simple linked list as a dictionary. Assume the client will never provide duplicate elements, so we can insert elements at the beginning of the list. Now assume the peculiar situation that the client may Perform any number of insert operations but will only ever perform atmost one lookup operation.

A. What is the worst-case running time of the operations performed on this data structure under the assumptions above? Briefly justify your answer.

- The worst-case running time of the operations performed on this linked list data structure is $O(n)$, where n is the number of elements in the list.
- This is because in the worst case, the lookup operation will have to traverse the entire list to find the desired element.
- Since there is no assumption about the order of elements in the list, we cannot use any optimization to reduce the worst-case time complexity.

B. What is the worst-case amortized running time of the operations performed on this data structure under the assumptions above? Briefly justify your answer.

- The worst-case amortized running time of the operations performed on this linked list data structure is $O(1)$ for insertions and $O(n)$ for lookup.
- This is because each insertion takes constant time $O(1)$ to add an element to the beginning of the list. However, the lookup operation can take up to $O(n)$ time in the worst-case scenario.
- Since there is only at most one lookup operation, the overall cost of this operation is distributed across all insertions, resulting in an amortized cost of $O(n)/n = O(1)$.
- Therefore, the amortized worst-case time complexity of both insertions and lookup is $O(1)$.