DAY 9

1.
```
def maxCoins(piles):
    # Sort the piles array in ascending order
    piles.sort()

    # Initialize the variable to keep track of your total coins
    total_coins = 0

    # Since the total number of piles is 3n, we can process them in chunks of 3.
    # We will pick the second largest pile from each triplet
    for i in range(len(piles) // 3):
        total_coins += piles[-2 - 2*i]  # This is the second largest in the triplet

    return total_coins
```

2.
```
def minPatches(coins, target):
    coins.sort()  # Sort the coins to make the greedy strategy work
    current_sum = 0  # This tracks the maximum sum we can form so far
    patches = 0  # The number of new coins we need to add

    i = 0  # Pointer to the coins array
    while current_sum < target:
        # If the current coin can be used to form the sum `current_sum + 1`
        if i < len(coins) and coins[i] <= current_sum + 1:
            current_sum += coins[i]
            i += 1
        else:
            # Otherwise, we need to add a coin of value `current_sum + 1`
            current_sum += current_sum + 1
            patches += 1

    return patches
```

3.
```
def minimumTimeRequired(jobs, k):
    def canAssignJobs(maxTime):
        # Function to check if it's possible to assign jobs such that no worker has more than maxTime work
```

```
        workers = [0] * k  # Initialize all workers' workloads to 0

        def backtrack(i):
            if i == len(jobs):  # All jobs have been assigned
                return True
            for j in range(k):
                if workers[j] + jobs[i] <= maxTime:
                    workers[j] += jobs[i]  # Assign job to worker
                    if backtrack(i + 1):  # Recur to assign the next job
                        return True
                    workers[j] -= jobs[i]  # Undo assignment if it didn't work
                # If a worker is idle (workers[j] == 0) and we already tried this, skip to avoid duplicate work
                if workers[j] == 0:
                    break
            return False

        return backtrack(0)

    # Binary search for the minimum possible maximum working time
    low, high = max(jobs), sum(jobs)
    while low < high:
        mid = (low + high) // 2
        if canAssignJobs(mid):
            high = mid  # Try to minimize the max working time
        else:
            low = mid + 1  # Increase the max working time

    return low
```

4.

```
import bisect

def jobScheduling(startTime, endTime, profit):
    # Create a list of jobs with start, end, and profit
    jobs = list(zip(startTime, endTime, profit))

    # Sort jobs by their end time
    jobs.sort(key=lambda x: x[1])

    # dp[i] will store the maximum profit considering the first i jobs
    n = len(jobs)
    dp = [0] * (n + 1)

    # Helper function to find the rightmost job that doesn't overlap with job i
    def findLastNonConflictingJob(i):
        # We need to find the job whose end time is <= the current job's start time
        # Binary search on the end times
        low, high = 0, i - 1
        while low <= high:
            mid = (low + high) // 2
            if jobs[mid][1] <= jobs[i][0]:
```

```
                if jobs[mid + 1][1] <= jobs[i][0]:
                    low = mid + 1
                else:
                    return mid
            else:
                high = mid - 1
        return -1

    # Fill dp array
    for i in range(1, n + 1):
        # Option 1: Do not take this job, profit remains same as previous job
        dp[i] = dp[i - 1]

        # Option 2: Take this job
        prevJobIndex = findLastNonConflictingJob(i - 1)
        profitIncludingCurrent = jobs[i - 1][2] + (dp[prevJobIndex + 1] if prevJobIndex != -1 else 0)

        # Maximize profit by taking or not taking the job
        dp[i] = max(dp[i], profitIncludingCurrent)

    # The answer will be in dp[n], the maximum profit
    return dp[n]

# Example 1:
startTime1 = [1, 2, 3, 3]
endTime1 = [3, 4, 5, 6]
profit1 = [50, 10, 40, 70]
print(jobScheduling(startTime1, endTime1, profit1))  # Output: 120

# Example 2:
startTime2 = [1, 2, 3, 4, 6]
endTime2 = [3, 5, 10, 6, 9]
profit2 = [20, 20, 100, 70, 60]
print(jobScheduling(startTime2, endTime2, profit2))  # Output: 150
```

5.

```
import heapq

def dijkstra(n, graph, source):
    # Initialize distances as infinity for all vertices except the source
    dist = [float('inf')] * n
    dist[source] = 0

    # Priority queue to store (distance, vertex) pairs
    pq = [(0, source)]  # (distance, vertex)

    # While there are vertices to process
    while pq:
        # Get the vertex with the smallest tentative distance
        current_dist, u = heapq.heappop(pq)
```

```
        # If the current distance is greater than the stored distance, continue
        if current_dist > dist[u]:
            continue

        # Explore neighbors of vertex u
        for v in range(n):
            if graph[u][v] != float('inf'):  # Check if there's an edge
                weight = graph[u][v]
                # If the new calculated distance is shorter, update it
                if dist[u] + weight < dist[v]:
                    dist[v] = dist[u] + weight
                    heapq.heappush(pq, (dist[v], v))

    return dist

# Test Case 1
n1 = 5
graph1 = [
    [0, 10, 3, float('inf'), float('inf')],
    [float('inf'), 0, 1, 2, float('inf')],
    [float('inf'), 4, 0, 8, 2],
    [float('inf'), float('inf'), float('inf'), 0, 7],
    [float('inf'), float('inf'), float('inf'), 9, 0]
]
source1 = 0
print(dijkstra(n1, graph1, source1))  # Output: [0, 7, 3, 9, 5]

# Test Case 2
n2 = 4
graph2 = [
    [0, 5, float('inf'), 10],
    [float('inf'), 0, 3, float('inf')],
    [float('inf'), float('inf'), 0, 1],
    [float('inf'), float('inf'), float('inf'), 0]
]
source2 = 0
print(dijkstra(n2, graph2, source2))  # Output: [0, 5, 8, 9]
```

6.

```
import heapq

def dijkstra(n, edges, source, target):
    # Step 1: Convert the edge list to an adjacency list
    graph = {i: [] for i in range(n)}
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))  # For undirected graph, add reverse edge too
```

```
    # Step 2: Initialize the distance array and priority queue
    dist = [float('inf')] * n
    dist[source] = 0
    pq = [(0, source)]  # Priority queue stores (distance, vertex)

    # Step 3: Process the vertices
    while pq:
        current_dist, u = heapq.heappop(pq)

        # If we reached the target, return the distance
        if u == target:
            return current_dist

        # If the current distance is greater than the known distance, skip it
        if current_dist > dist[u]:
            continue

        # Explore the neighbors of the current vertex
        for v, weight in graph[u]:
            new_dist = current_dist + weight
            if new_dist < dist[v]:
                dist[v] = new_dist
                heapq.heappush(pq, (new_dist, v))

    # If we reach here, no path exists to the target
    return -1

# Test Case 1
n1 = 6
edges1 = [
    (0, 1, 7), (0, 2, 9), (0, 5, 14), (1, 2, 10), (1, 3, 15),
    (2, 3, 11), (2, 5, 2), (3, 4, 6), (4, 5, 9)
]
source1 = 0
target1 = 4
print(dijkstra(n1, edges1, source1, target1))  # Output: 20

# Test Case 2
n2 = 5
edges2 = [
    (0, 1, 10), (0, 4, 3), (1, 2, 2), (1, 4, 4), (2, 3, 9),
    (3, 2, 7), (4, 1, 1), (4, 2, 8), (4, 3, 2)
]
source2 = 0
target2 = 3
print(dijkstra(n2, edges2, source2, target2))  # Output: 8
```

7.

```
import heapq
```

```python
class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    # To make the node comparable based on frequency
    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(characters, frequencies):
    # Step 1: Create a priority queue (min-heap) and insert all the nodes
    priority_queue = []
    for char, freq in zip(characters, frequencies):
        heapq.heappush(priority_queue, Node(char, freq))

    # Step 2: Build the Huffman Tree
    while len(priority_queue) > 1:
        # Extract the two nodes with the lowest frequency
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)

        # Create a new internal node with the sum of frequencies
        merged_node = Node(None, left.freq + right.freq)
        merged_node.left = left
        merged_node.right = right

        # Insert the new internal node back into the priority queue
        heapq.heappush(priority_queue, merged_node)

    # The remaining node is the root of the Huffman tree
    return priority_queue[0]

def generate_huffman_codes(root, current_code=""):
    # Step 3: Generate the Huffman codes by traversing the tree
    if root is None:
        return {}

    if root.char is not None:  # It's a leaf node
        return {root.char: current_code}

    # Recursively get the codes for the left and right subtrees
    codes = {}
    codes.update(generate_huffman_codes(root.left, current_code + "0"))
    codes.update(generate_huffman_codes(root.right, current_code + "1"))

    return codes

def huffman_coding(characters, frequencies):
    # Build the Huffman Tree
    root = build_huffman_tree(characters, frequencies)
```

```
    # Generate the Huffman Codes
    codes = generate_huffman_codes(root)

    # Return the codes sorted by the characters
    return sorted(codes.items())

# Test Case 1
characters1 = ['a', 'b', 'c', 'd']
frequencies1 = [5, 9, 12, 13]
print(huffman_coding(characters1, frequencies1))
# Expected Output: [('a', '110'), ('b', '10'), ('c', '0'), ('d', '111')]

# Test Case 2
characters2 = ['f', 'e', 'd', 'c', 'b', 'a']
frequencies2 = [5, 9, 12, 13, 16, 45]
print(huffman_coding(characters2, frequencies2))
# Expected Output: [('a', '0'), ('b', '101'), ('c', '100'), ('d', '111'), ('e', '1101'), ('f', '1100')]
```

8.

```
import heapq

class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    # To make the node comparable based on frequency
    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(characters, frequencies):
    # Step 1: Create a priority queue (min-heap) and insert all the nodes
    priority_queue = []
    for char, freq in zip(characters, frequencies):
        heapq.heappush(priority_queue, Node(char, freq))

    # Step 2: Build the Huffman Tree
    while len(priority_queue) > 1:
        # Extract the two nodes with the lowest frequency
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)

        # Create a new internal node with the sum of frequencies
        merged_node = Node(None, left.freq + right.freq)
        merged_node.left = left
        merged_node.right = right

        # Insert the new internal node back into the priority queue
```

```
        heapq.heappush(priority_queue, merged_node)

    # The remaining node is the root of the Huffman tree
    return priority_queue[0]

def decode_huffman_tree(root, encoded_string):
    # Step 3: Decode the encoded string
    decoded_message = []
    node = root  # Start at the root of the Huffman tree

    for bit in encoded_string:
        # Traverse the tree based on the bit: '0' means left, '1' means right
        if bit == '0':
            node = node.left
        else:
            node = node.right

        # If we reach a leaf node, add the character to the result and reset to the root
        if node.char is not None:
            decoded_message.append(node.char)
            node = root  # Reset to the root

    # Join the decoded characters to form the decoded message
    return ''.join(decoded_message)

def huffman_decoding(characters, frequencies, encoded_string):
    # Build the Huffman Tree
    root = build_huffman_tree(characters, frequencies)

    # Decode the string using the Huffman Tree
    decoded_message = decode_huffman_tree(root, encoded_string)

    return decoded_message

# Test Case 1
characters1 = ['a', 'b', 'c', 'd']
frequencies1 = [5, 9, 12, 13]
encoded_string1 = '1101100111110'
print(huffman_decoding(characters1, frequencies1, encoded_string1))
# Expected Output: "abacd"

# Test Case 2
characters2 = ['f', 'e', 'd', 'c', 'b', 'a']
frequencies2 = [5, 9, 12, 13, 16, 45]
encoded_string2 = '110011011100101111001011'
print(huffman_decoding(characters2, frequencies2, encoded_string2))
# Expected Output: "fcbade"
```

9.

```
def max_weight_loaded(weights, max_capacity):
```

```
    # Step 1: Sort the weights in descending order
    sorted_weights = sorted(weights, reverse=True)

    # Step 2: Initialize the total weight loaded into the container
    total_weight = 0

    # Step 3: Try to add items to the container
    for weight in sorted_weights:
        if total_weight + weight <= max_capacity:
            total_weight += weight
        else:
            break  # Stop if adding this item exceeds the capacity

    return total_weight

# Test Case 1
weights1 = [10, 20, 30, 40, 50]
max_capacity1 = 60
print(max_weight_loaded(weights1, max_capacity1))  # Expected Output: 50

# Test Case 2
weights2 = [5, 10, 15, 20, 25, 30]
max_capacity2 = 50
print(max_weight_loaded(weights2, max_capacity2))  # Expected Output: 50
```

10.

```
def min_containers(weights, max_capacity):
    # Step 1: Sort the weights in descending order
    weights.sort(reverse=True)

    # Step 2: Initialize the number of containers and current container's remaining capacity
    container_count = 0
    current_capacity = 0

    # Step 3: Try to fit items into containers
    for weight in weights:
        if current_capacity + weight <= max_capacity:
            # Add the item to the current container
            current_capacity += weight
        else:
            # The current container can't fit this item, so use a new container
            container_count += 1
            current_capacity = weight  # Start a new container with this item

    # Account for the last container being used
    if current_capacity > 0:
        container_count += 1

    return container_count
```

```
# Test Case 1
weights1 = [5, 10, 15, 20, 25, 30, 35]
max_capacity1 = 50
print(min_containers(weights1, max_capacity1))  # Expected Output: 4

# Test Case 2
weights2 = [10, 20, 30, 40, 50, 60, 70, 80]
max_capacity2 = 100
print(min_containers(weights2, max_capacity2))  # Expected Output: 6
```

11.

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

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])  # Path compression
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)

        if root_u != root_v:
            # Union by rank
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1
            return True
        return False

def kruskal(n, edges):
    # Step 1: Sort the edges by weight
    edges.sort(key=lambda x: x[2])  # Sort by the third element in each tuple (weight)

    uf = UnionFind(n)
    mst = []
    mst_weight = 0

    # Step 2: Iterate through sorted edges
    for u, v, weight in edges:
        if uf.union(u, v):  # If u and v were not connected
            mst.append((u, v, weight))
```

```
            mst_weight += weight

        # If we've added n-1 edges, we can stop
        if len(mst) == n - 1:
            break

    return mst, mst_weight

# Test Case 1
n1 = 4
m1 = 5
edges1 = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
mst1, mst_weight1 = kruskal(n1, edges1)
print("Edges in MST:", mst1)
print("Total weight of MST:", mst_weight1)

# Test Case 2
n2 = 5
m2 = 7
edges2 = [(0, 1, 2), (0, 3, 6), (1, 2, 3), (1, 3, 8), (1, 4, 5), (2, 4, 7), (3, 4, 9)]
mst2, mst_weight2 = kruskal(n2, edges2)
print("Edges in MST:", mst2)
print("Total weight of MST:", mst_weight2)
```

12.

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

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])  # Path compression
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)

        if root_u != root_v:
            # Union by rank
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1
            return True
        return False
```

```
def kruskal(n, edges):
    # Step 1: Sort the edges by weight
    edges.sort(key=lambda x: x[2])  # Sort by the third element in each tuple (weight)

    uf = UnionFind(n)
    mst = []
    mst_weight = 0

    # Step 2: Iterate through sorted edges
    for u, v, weight in edges:
        if uf.union(u, v):  # If u and v were not connected
            mst.append((u, v, weight))
            mst_weight += weight

        # If we've added n-1 edges, we can stop
        if len(mst) == n - 1:
            break

    return mst, mst_weight

def is_mst_unique(n, edges, given_mst):
    # Step 1: Get the MST from Kruskal's algorithm
    mst, mst_weight = kruskal(n, edges)

    # Step 2: Check if the given MST matches the one found by Kruskal's
    given_mst_weight = sum(weight for u, v, weight in given_mst)
    if given_mst_weight != mst_weight:
        return True, None  # If the weights are not equal, the given MST is incorrect

    # Step 3: Check for alternative MSTs
    # Since the MST weight is the same, check if we can swap any edges without increasing the total weight
    # We will attempt to find another valid MST by considering edges with the same weight.

    # We will try to construct a different MST
    uf = UnionFind(n)
    alternative_mst = []

    # Start with the edges in the original MST
    mst_set = set(given_mst)

    # Add edges to the alternative MST from the sorted list
    for u, v, weight in edges:
        if uf.union(u, v) and (u, v, weight) not in mst_set:
            alternative_mst.append((u, v, weight))

        # If we have found a valid alternative MST, break
        if len(alternative_mst) == n - 1:
            break

    # If we find a valid alternative MST, then the given MST is not unique
    if len(alternative_mst) == n - 1:
        return False, alternative_mst
    else:
```

```
        return True, None

# Test Case 1
n1 = 4
m1 = 5
edges1 = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
given_mst1 = [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
is_unique1, alt_mst1 = is_mst_unique(n1, edges1, given_mst1)
if is_unique1:
    print("Is the given MST unique?", True)
else:
    print("Is the given MST unique?", False)
    print("Another possible MST:", alt_mst1)

# Test Case 2
n2 = 5
m2 = 6
edges2 = [(0, 1, 1), (0, 2, 1), (1, 3, 2), (2, 3, 2), (3, 4, 3), (4, 2, 3)]
given_mst2 = [(0, 1, 1), (0, 2, 1), (1, 3, 2), (3, 4, 3)]
is_unique2, alt_mst2 = is_mst_unique(n2, edges2, given_mst2)
if is_unique2:
    print("Is the given MST unique?", True)
else:
    print("Is the given MST unique?", False)
    print("Another possible MST:", alt_mst2)
```