

## TOPIC 5 : GREEDY

1. There are  $3n$  piles of coins of varying size, you and your friends will take piles of coins as follows: In each step, you will choose any 3 piles of coins (not necessarily consecutive). Of your choice, Alice will pick the pile with the maximum number of coins. You will pick the next pile with the maximum number of coins. Your friend Bob will pick the last pile. Repeat until there are no more piles of coins. Given an array of integers `piles` where `piles[i]` is the number of coins in the  $i$ th pile. Return the maximum number of coins that you can have.

**Program:**

```
piles = [2,4,1,2,7,8]
```

```
piles.sort(reverse=True)
```

```
your_coins = 0
```

```
n = len(piles)
```

```
for i in range(n):
```

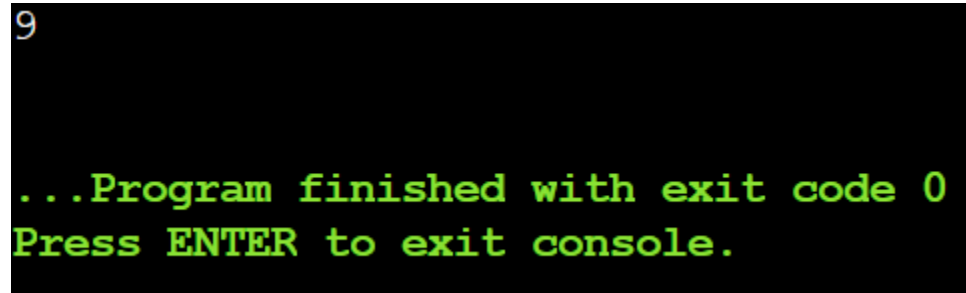
```
    your_coins += piles[2*i + 1]
```

```
print(your_coins)
```

**Sample Input:**

```
piles = [2,4,1,2,7,8]
```

**Output:**



```
9
...Program finished with exit code 0
Press ENTER to exit console.
```

2. You are given a 0-indexed integer array `coins`, representing the values of the coins available, and an integer `target`. An integer  $x$  is obtainable if there exists a subsequence of `coins` that sums to  $x$ . Return the minimum number of coins of any value that need to be added to the array so that every integer in the range  $[1, \text{target}]$  is obtainable. A subsequence of an array is a new non-empty array that is formed from the original array by deleting some (possibly none) of the elements without disturbing the relative positions of the remaining elements.

**Program:**

```
coins = [1, 4, 10]
```

```

target = 19

coins.sort()

miss = 1
added = 0
i = 0

while miss <= target:
    if i < len(coins) and coins[i] <= miss:
        miss += coins[i]
        i += 1
    else:
        miss += miss
        added += 1

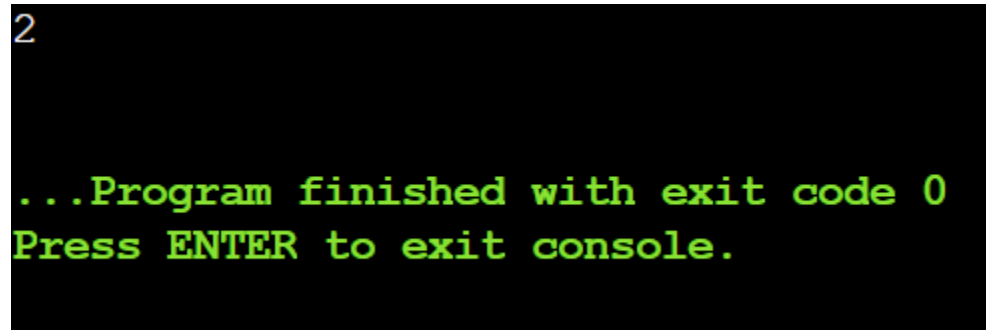
print(added)

```

**Sample Input:**

coins = [1,4,10], target = 19

**Output:**



```

2

...Program finished with exit code 0
Press ENTER to exit console.

```

**3.** You are given an integer array jobs, where jobs[i] is the amount of time it takes to complete the ith job. There are k workers that you can assign jobs to. Each job should be assigned to exactly one worker. The working time of a worker is the sum of the time it takes to complete all jobs assigned to them. Your goal is to devise an optimal assignment such that the maximum working time of any worker is minimized. Return the minimum possible maximum working time of any assignment.

**Program:**

```

jobs = [3, 2, 3]
k = 3

```

```

def can_assign(jobs, workers, max_time, index):
    if index == len(jobs):
        return True
    for i in range(len(workers)):
        if workers[i] + jobs[index] <= max_time:
            workers[i] += jobs[index]
            if can_assign(jobs, workers, max_time, index + 1):
                return True
            workers[i] -= jobs[index]
        if workers[i] == 0:
            break
    return False

left, right = max(jobs), sum(jobs)
while left < right:
    mid = (left + right)
    if can_assign(jobs, [0]*k, mid, 0):
        right = mid
    else:
        left = mid + 1

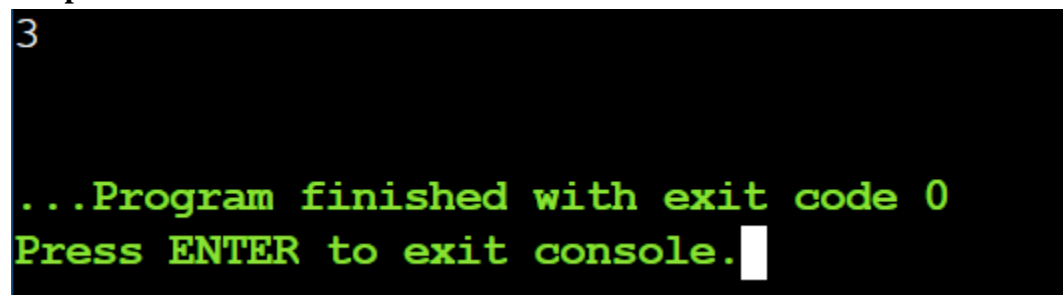
print(left)

```

**Sample Input:**

jobs = [3,2,3], k = 3

**Output:**



```

3

...Program finished with exit code 0
Press ENTER to exit console.

```

4. We have  $n$  jobs, where every job is scheduled to be done from  $startTime[i]$  to  $endTime[i]$ , obtaining a profit of  $profit[i]$ . You're given the  $startTime$ ,  $endTime$  and  $profit$  arrays, return the maximum profit you can take such that there are no two jobs in the subset with overlapping time range. If you choose a job that ends at time  $X$  you will be able to start another job that

**starts at time X.**

**Program:**

```
import bisect
```

```
startTime = [1,2,3,3]
```

```
endTime = [3,4,5,6]
```

```
profit = [50,10,40,70]
```

```
jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])
```

```
n = len(jobs)
```

```
dp = [0] * n
```

```
dp[0] = jobs[0][2]
```

```
end_times = [job[1] for job in jobs]
```

```
for i in range(1, n):
```

```
    incl_prof = jobs[i][2]
```

```
    idx = bisect.bisect_right(end_times, jobs[i][0]-1) - 1
```

```
    if idx != -1:
```

```
        incl_prof += dp[idx]
```

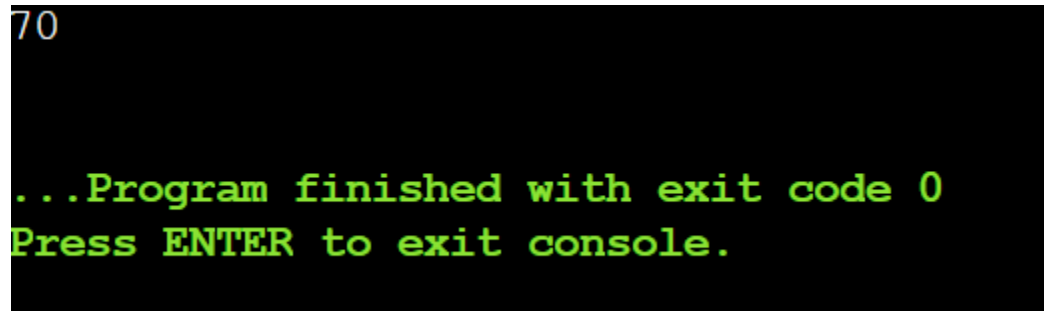
```
    dp[i] = max(incl_prof, dp[i-1])
```

```
print(dp[-1])
```

**Sample Input:**

```
startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70]
```

**Output:**



```
70
...Program finished with exit code 0
Press ENTER to exit console.
```

**5. Given a graph represented by an adjacency matrix, implement Dijkstra's**

**Algorithm to find the shortest path from a given source vertex to all other vertices in the graph. The graph is represented as an adjacency matrix where  $graph[i][j]$  denote the weight of the edge from vertex  $i$  to vertex  $j$ . If there is no edge between vertices  $i$  and  $j$ , the value is Infinity (or a very large number).**

**Program:**

```
import math

n = 5
graph = [
    [0, 10, 3, math.inf, math.inf],
    [math.inf, 0, 1, 2, math.inf],
    [math.inf, 4, 0, 8, 2],
    [math.inf, math.inf, math.inf, 0, 7],
    [math.inf, math.inf, math.inf, 9, 0]
]
source = 0

dist = [math.inf] * n
dist[source] = 0
visited = [False] * n

for _ in range(n):

    min_dist = math.inf
    u = -1
    for i in range(n):
        if not visited[i] and dist[i] < min_dist:
            min_dist = dist[i]
            u = i

    visited[u] = True

    for v in range(n):
        if not visited[v] and graph[u][v] != math.inf:
            if dist[v] > dist[u] + graph[u][v]:
                dist[v] = dist[u] + graph[u][v]
print(dist)
```

**Sample Input:**

```

n = 5
graph = [[0, 10, 3, Infinity, Infinity], [Infinity, 0, 1, 2, Infinity], [Infinity, 4, 0, 8,
2],
[Infinity, Infinity, Infinity, 0, 7], [Infinity, Infinity,
Infinity, 9, 0]]
source = 0

```

**Output:**

```

[0, 7, 3, 9, 5]

...Program finished with exit code 0
Press ENTER to exit console.

```

**6. Given a graph represented by an edge list, implement Dijkstra's Algorithm to find the shortest path from a given source vertex to a target vertex. The graph is represented as a list of edges where each edge is a tuple (u, v, w) representing an edge from vertex u to vertex v with weight w.**

**Program:**

```

import heapq

n = 6
edges = [
    (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)
]
source = 0
target = 4

graph = [[] for _ in range(n)]
for u, v, w in edges:
    graph[u].append((v, w))
    graph[v].append((u, w))

dist = [float('inf')] * n
dist[source] = 0
pq = [(0, source)]

```

```

while pq:
    curr_dist, u = heapq.heappop(pq)
    if curr_dist > dist[u]:
        continue
    for v, w in graph[u]:
        if dist[v] > dist[u] + w:
            dist[v] = dist[u] + w
            heapq.heappush(pq, (dist[v], v))

print(dist[target])

```

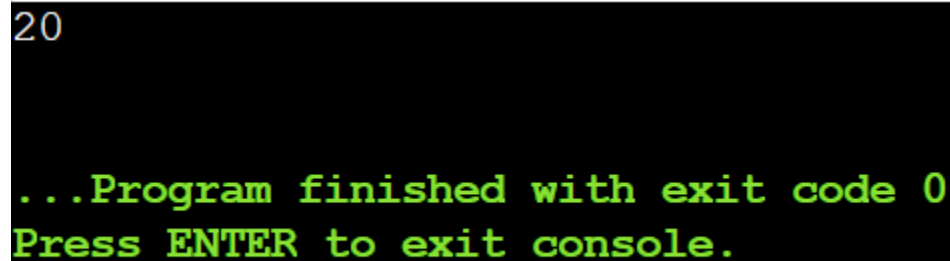
### Sample Input:

```

n = 6
edges = [(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) ]
source = 0
target = 4

```

### Output:



```

20

...Program finished with exit code 0
Press ENTER to exit console.

```

**7. Given a set of characters and their corresponding frequencies, construct the Huffman Tree and generate the Huffman Codes for each character.**

### Program:

```

import heapq

characters = ['f', 'e', 'd', 'c', 'b', 'a']
frequencies = [5, 9, 12, 13, 16, 45]

heap = []
for ch, freq in zip(characters, frequencies):
    heapq.heappush(heap, [freq, ch])

```

```

while len(heap) > 1:
    left = heapq.heappop(heap)
    right = heapq.heappop(heap)
    heapq.heappush(heap, [left[0] + right[0], left, right])

```

```

codes = { }
stack = [(heap[0], "")]

```

```

while stack:
    node, code = stack.pop()
    if isinstance(node[1], str):
        codes[node[1]] = code
    else:
        stack.append((node[2], code + "1"))
        stack.append((node[1], code + "0"))

```

```

result = [(ch, codes[ch]) for ch in sorted(codes)]
print(result)

```

### Sample Input:

```

n = 6
characters = ['f', 'e', 'd', 'c', 'b', 'a']
frequencies = [5, 9, 12, 13, 16, 45]

```

### Output:

```

[('a', '0'), ('b', '111'), ('c', '101'), ('d', '100'), ('e', '1101'), ('f', '1100')]

...Program finished with exit code 0
Press ENTER to exit console.

```

**8. Given a Huffman Tree and a Huffman encoded string, decode the string to get the original message.**

### Program:

```

import heapq
characters = ['a', 'b', 'c', 'd']
frequencies = [5, 9, 12, 13]
encoded_string = "1101100111110"

```

```

heap = []
for ch, freq in zip(characters, frequencies):
    heapq.heappush(heap, [freq, ch])

```



```

while len(heap) > 1:
    left = heapq.heappop(heap)
    right = heapq.heappop(heap)
    heapq.heappush(heap, [left[0] + right[0], left, right])

root = heap[0]

decoded = ""
node = root

for bit in encoded_string:
    if bit == '0':
        node = node[1]
    else:
        node = node[2]

    if isinstance(node[1], str): # Leaf node
        decoded += node[1]
        node = root

print(decoded)

```

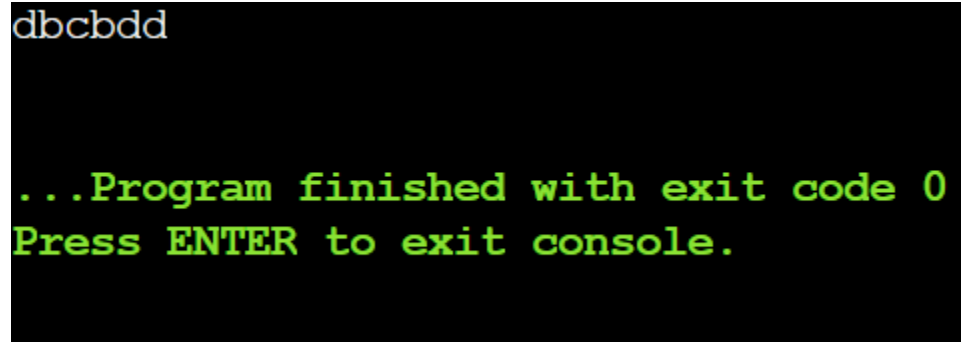
### Sample Input:

```

n = 4
characters = ['a', 'b', 'c', 'd']
frequencies = [5, 9, 12, 13]
encoded_string = '1101100111110'

```

### Output:



```

dbcbdd

...Program finished with exit code 0
Press ENTER to exit console.

```

**9. Given a list of item weights and the maximum capacity of a container, determine the maximum weight that can be loaded into the container using a greedy approach. The greedy approach should prioritize loading heavier**

**items first until the container reaches its capacity.**

**Program:**

```
weights = [5, 10, 15, 20, 25, 30]
```

```
max_capacity = 50
```

```
weights.sort(reverse=True)
```

```
total_weight = 0
```

```
for w in weights:
```

```
    if total_weight + w <= max_capacity:
```

```
        total_weight += w
```

```
print(total_weight)
```

**Sample Input:**

```
n = 6
```

```
weights = [5, 10, 15, 20, 25, 30]
```

```
max_capacity = 50
```

**Output:**

```
50
```

```
...Program finished with exit code 0  
Press ENTER to exit console. █
```

**10. Given a list of item weights and a maximum capacity for each container, determine the minimum number of containers required to load all items using a greedy approach. The greedy approach should prioritize loading items into the current container until it is full before moving to the next Container.**

**Program:**

```
weights = [5, 10, 15, 20, 25, 30, 35]
```

```
max_capacity = 50
```

```
weights.sort(reverse=True)
```

```

containers = 0
i = 0
n = len(weights)
while i < n:
    current_load = 0
    while i < n and current_load + weights[i] <= max_capacity:
        current_load += weights[i]
        i += 1
    containers += 1

print(containers)

```

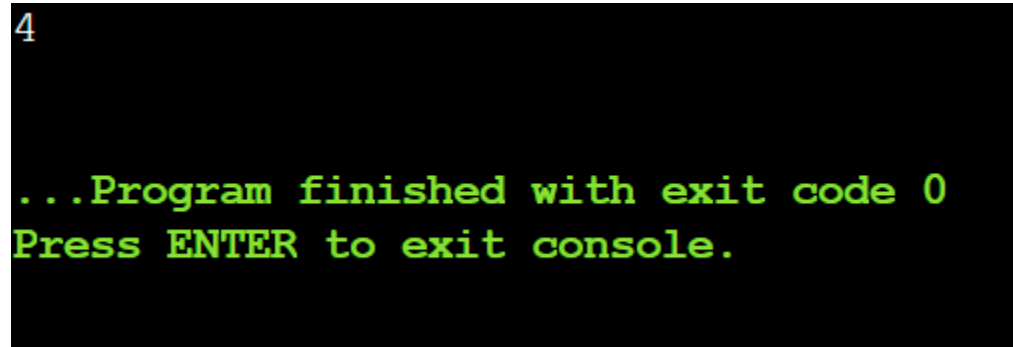
**Sample Input:**

```

n = 7
weights = [5, 10, 15, 20, 25, 30, 35]
max_capacity = 50

```

**Output:**



```

4
...Program finished with exit code 0
Press ENTER to exit console.

```

**11. Given a graph represented by an edge list, implement Kruskal's Algorithm to find the Minimum Spanning Tree (MST) and its total weight.**

**Program:**

```

n = 4
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]

edges.sort(key=lambda x: x[2])

parent = [i for i in range(n)]
rank = [0] * n

def find(u)
    while parent[u] != u:

```

```

        parent[u] = parent[parent[u]]
        u = parent[u]
    return u

def union(u, v):
    u_root = find(u)
    v_root = find(v)
    if u_root == v_root:
        return False
    if rank[u_root] < rank[v_root]:
        parent[u_root] = v_root
    elif rank[u_root] > rank[v_root]:
        parent[v_root] = u_root
    else:
        parent[v_root] = u_root
        rank[u_root] += 1
    return True

mst_edges = []
total_weight = 0

for u, v, w in edges:
    if union(u, v):
        mst_edges.append((u, v, w))
        total_weight += w

print("Edges in MST:", mst_edges)
print("Total weight of MST:", total_weight)

```

### **Sample Input:**

Input:

n = 4

m = 5

edges = [ (0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4) ]

### **Output:**

```
Edges in MST: [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
Total weight of MST: 19
```

```
...Program finished with exit code 0
Press ENTER to exit console.
```

**12.** Given a graph with weights and a potential Minimum Spanning Tree (MST), verify if the given MST is unique. If it is not unique, provide another possible MST.

**Program:**

```
n = 4
edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]
given_mst = [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
```

```
edges.sort(key=lambda x: x[2])
```

```
parent = [i for i in range(n)]
rank = [0] * n
```

```
def find(u):
    while parent[u] != u:
        parent[u] = parent[parent[u]]
        u = parent[u]
    return u
```

```
def union(u, v):
    u_root = find(u)
    v_root = find(v)
    if u_root == v_root:
        return False
    if rank[u_root] < rank[v_root]:
        parent[u_root] = v_root
    elif rank[u_root] > rank[v_root]:
        parent[v_root] = u_root
    else:
        parent[v_root] = u_root
        rank[u_root] += 1
    return True
```

```

mst_edges = []
for u, v, w in edges:
    if union(u, v):
        mst_edges.append((u, v, w))
unique = True
weights = [w for _, _, w in edges]
for i in range(len(edges)):
    for j in range(i+1, len(edges)):
        if edges[i][2] == edges[j][2] and edges[i] not in mst_edges and edges[j] not in mst_edges:
            unique = False
            break

print("Is the given MST unique?", unique)

```

### Sample Input:

```

n = 4
m = 5
edges = [ (0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4) ]
given_mst = [(2, 3, 4), (0, 3, 5), (0, 1, 10)]

```

### Output:

```

Is the given MST unique? True

...Program finished with exit code 0
Press ENTER to exit console.

```