

TOPIC-5: GREEDY

1. Maximum of Coins

Aim: To finds the maximum number of coins you can collect from $3n$ piles of coins.

Algorithm:

1. Start the program.
2. Read the number of piles n (must be a multiple of 3).
3. Input all the pile values (number of coins in each pile).
4. Sort the pile values in ascending order.
5. Initialize a variable result = 0 to store your total coins
6. Stop the program.

Input & Output: -

The screenshot shows two windows side-by-side. On the left is a code editor window titled 'tpoi 6.1.py - C:/Users/Reddy/AppData/Local/Programs/Python/Python313/tpoi 6.1.py (3.13.5)'. It contains Python code for a greedy algorithm to solve the '3n Piles of Coins Problem'. On the right is an 'IDLE Shell 3.13.5' window showing the execution of the program. The shell window has a title bar 'IDLE Shell 3.13.5' and a menu bar 'File Edit Shell Debug Options Window Help'. It displays the Python version and path, followed by three runs of the program. In each run, it prompts for the number of piles (must be a multiple of 3), enters 3, asks for 3 pile values, sorts them, and prints the total maximum coins collected.

```
tpoi 6.1.py - C:/Users/Reddy/AppData/Local/Programs/Python/Python313/tpoi 6.1.py (3.13.5)
File Edit Format Run Options Window Help
def maxCoins(piles):
    piles.sort()
    n = len(piles) // 3
    result = 0
    for i in range(n, len(piles), 2):
        result += piles[i]
    return result
print("==>>> 3n Piles of Coins Problem ===")
num = int(input("Enter number of piles (must be multiple of 3): "))
if num % 3 != 0:
    print("Please enter a number that is a multiple of 3.")
else:
    piles = []
    print(f"Enter {num} pile values:")
    for i in range(num):
        val = int(input(f"File {i+1}: "))
        piles.append(val)
    print("\nFiles you entered:", piles)
    print("Maximum coins you can collect:", maxCoins(piles))

>>>
IDLE Shell 3.13.5
File Edit Shell Debug Options Window Help
Python 3.13.5 (tags/v3.13.5:6ccb0aa2, Jun 11 2025, 16:15:46) [MSC v.1943 64
bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
= RESTART: C:/Users/Reddy/AppData/Local/Programs/Python/Python313/tpoi 6.1
.py =
==>>> 3n Piles of Coins Problem ===
Enter number of piles (must be multiple of 3): 3
Enter 3 pile values:
File 1: 3
File 2: 9
File 3: 18
Piles you entered: [3, 9, 18]
Maximum coins you can collect: 9
>>>
= RESTART: C:/Users/Reddy/AppData/Local/Programs/Python/Python313/tpoi 6.1
.py =
==>>> 3n Piles of Coins Problem ===
Enter number of piles (must be multiple of 3): 6
Enter 6 pile values:
File 1: 3
File 2: 6
File 3: 9
File 4: 12
File 5: 18
File 6: 21
Piles you entered: [3, 6, 9, 12, 18, 21]
Maximum coins you can collect: 27
>>>
```

2. Minimum of Coins

Aim: To write a Python program that finds the minimum number of coins

Algorithm:

1. Start the program.
2. Input the list of existing coin values and the target value.
3. Sort the list of coins in ascending order.
4. Traverse through the coin list:
5. Continue this process until $\text{reachable} > \text{target}$.
6. Print the value of added coins as the minimum number of coins to be added.
7. Stop the program.

Input & Output: -

The image shows two windows side-by-side. The left window is titled 'topic 6.2.py' and contains the Python code for the coin problem. The right window is titled 'IDLE Shell 3.13.5' and shows the command-line interaction where the user enters coin values and a target value, and the program outputs the minimum number of coins needed.

```

topic 6.2.py - C:/Users/Reddy/AppData/Local/Programs/Python/Python313/topic 6.2...
File Edit Format Run Options Window Help
def minPatches(coins, target):
    coins.sort() # Sort coins in ascending order
    added_coins = 0
    reachable = 1
    i = 0

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

    return added_coins

# ---- Main Program ---
print("==== Minimum Coins to Cover Range [1, target] ====")

# User input
coins = list(map(int, input("Enter coin values separated by spaces: ").split))
target = int(input("Enter target value: "))

# Output result
print("Minimum coins needed to be added:", minPatches(coins, target))

IDLE Shell 3.13.5
File Edit Shell Debug Options Window Help
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
= RESTART: C:/Users/Reddy/AppData/Local/Programs/Python/Python313/topic 6.2.py =
==== Minimum Coins to Cover Range [1, target] ====
Enter coin values separated by spaces: 3 4 6 8 9
Enter target value: 2
Minimum coins needed to be added: 2

```

3. Maximum Working time

Aim:

To write a Python program that assigns n jobs among k each job is assigned to exactly one worker.

Algorithm:

1. Start the program.
2. Input the list of job times and the number of workers (k).
3. Define a helper function.
4. Use Binary Search on the answer.
5. For each mid-value between low and high:

6. Stop the program.

Input & Output: -

```
main.py | Run | Output
1 - def minimumTimeRequired(jobs, k):
2 -     jobs.sort(reverse=True)
3 -     workloads = [0] * k
4 -     n = len(jobs)
5 -     def backtrack(i):
6 -         if i == n:
7 -             return True
8 -         for w in range(k):
9 -             if workloads[w] + jobs[i] <= mid:
10 -                 workloads[w] += jobs[i]
11 -                 if backtrack(i + 1):
12 -                     return True
13 -                 workloads[w] -= jobs[i]
14 -             if workloads[w] == 0:
15 -                 break
16 -         return False
17 -     left, right = max(jobs), sum(jobs)
18 -     while left < right:
19 -         global mid
```

== Minimum Possible Maximum Working Time ==
Enter job times separated by spaces: 5 6 7 8 9
Enter number of workers: 15
Minimum possible maximum working time: 9
== Code Execution Successful ==

4. Maximum Profit

Aim: To find the maximum profit subset of jobs such that no two jobs overlap.

Algorithm

1. Start the program.
2. Input arrays start Time, end Time, and profit.
3. Combine jobs into tuples and sort by end Time.
4. Initialize a DP array and an end times list.
5. Stop the program.

Input & Output

main.py		Run	Output
<pre>1 from bisect import bisect_right 2 def jobScheduling(startTime, endTime, profit): 3 jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1]) 4 n = len(jobs) 5 dp = [0] * n 6 end_times = [job[1] for job in jobs] 7 for i in range(n): 8 incl_profit = jobs[i][2] 9 idx = bisect_right(end_times, jobs[i][0]) - 1 10 if idx != -1: 11 incl_profit += dp[idx] 12 dp[i] = max(incl_profit, dp[i-1] if i > 0 else 0) 13 return dp[-1] 14 startTime = [1, 2, 3, 3] 15 endTime = [3, 4, 5, 6] 16 profit = [50, 10, 40, 70] 17 print("Maximum Profit:", jobScheduling(startTime, endTime, profit))</pre>	Maximum Profit: 120 == Code Execution Successful ==		

4. Shortest Path

Aim: To find the shortest path from a given source vertex to all other vertices in a weighted graph represented by an adjacency matrix using Dijkstra's Algorithm.

1. Start the program.
2. Input the adjacency matrix graph and the source vertex.
3. Initialize distance array with infinity for all vertices, except the source vertex which is 0.
4. Initialize a visited array to keep track of visited vertices.
5. Print the distance array.
6. Stop the program.

Input & Output

The screenshot shows a Jupyter Notebook interface with the following components:

- Code Cell:** The code cell contains the Python implementation of Dijkstra's algorithm. It defines a function `dijkstra` that takes a graph and a source vertex as input. The code initializes a distance array with infinity for all vertices except the source, which is set to 0. It then iterates through the vertices to find the shortest path to each. The algorithm uses a min-heap to select the vertex with the minimum distance at each step.
- Output Cell:** The output cell displays the results of the execution. It shows a table with two columns: "Vertex" and "Distance from Source". The data is as follows:

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

- Execution Status:** Below the output table, the message "Code Execution Successful" is displayed.

5. Shortest Path (vertex to target)

Aim: To find the shortest path from a given source vertex to a target vertex in a weighted graph represented as an edge list using Dijkstra's Algorithm.

Algorithm:

1. Start the program.
2. Input the edge list, number of vertices, source vertex src, and target vertex tgt.
3. Convert the edge list into an adjacency list for efficient lookup.
4. Initialize distance array with infinity for all vertices, except the source vertex which is 0.
5. Use a priority queue (min-heap) to select the vertex with the minimum distance at each step while the priority queue is not empty.

Input & Output

Programiz Python Online Compiler

The screenshot shows a Python code editor interface with the following details:

- Title Bar:** Programiz Python Online Compiler
- Code Editor (main.py):**

```
1 import heapq
2 def dijkstra_shortest_path(edges, n, src, tgt):
3     graph = {i: [] for i in range(n)}
4     for u, v, w in edges:
5         graph[u].append((v, w))
6         graph[v].append((u, w))
7     dist = [float('inf')] * n
8     dist[src] = 0
9     pq = [(0, src)]
10    while pq:
11        current_dist, u = heapq.heappop(pq)
12        if u == tgt:
13            print(f"Shortest distance from {src} to {tgt} is:",
14                  current_dist)
15            return
16        if current_dist > dist[u]:
17            continue
18        for v, weight in graph[u]:
```
- Run Button:** A blue button labeled "Run".
- Output Panel:**

Shortest distance from 0 to 3 is: 4
== Code Execution Successful ==