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 ith pile. Return the maximum number of coins that you can have.

AIM: To find the **maximum number of coins you can collect** by optimally selecting piles when Alice always takes the largest pile, you take the second largest, and Bob takes the smallest, repeatedly.

PROCEDURE:

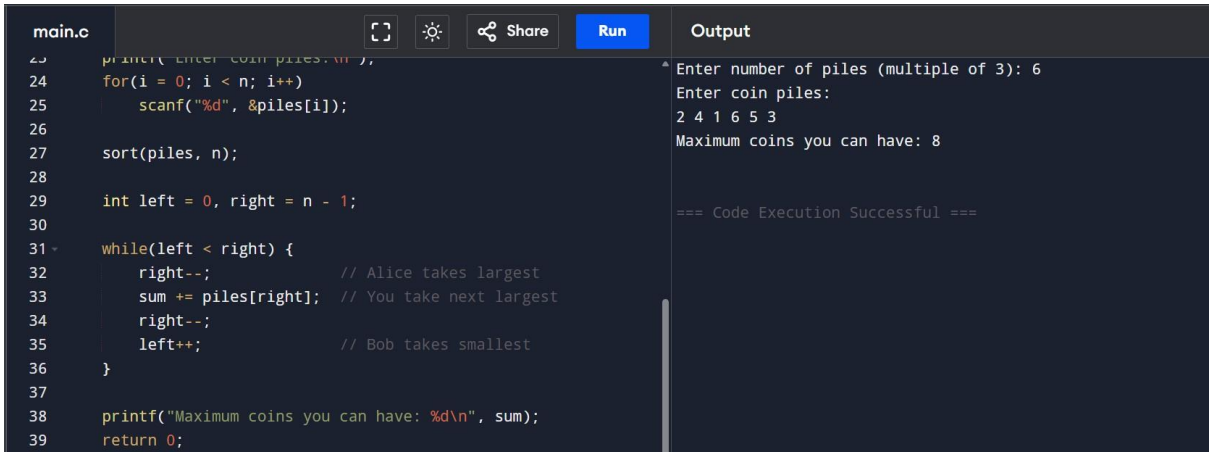
1. Read the number of piles n (where total piles = 3n).
2. Store the coin values in an array.
3. Sort the array in ascending order.
4. Use two pointers:
    - One from the end for Alice and you.
    - One from the start for Bob.
5. In each step:
    - Skip the largest pile (Alice).
    - Add the next largest pile to your total.
    - Skip the smallest pile (Bob).
6. Repeat until all piles are used.
7. Print the total coins you collected.

PROGRAM:

```
#include <stdio.h>


void sort(int a[], int n) {

int i, j, temp;     for(i = 0; i

< n-1; i++) {        for(j =

i+1; j < n; j++) {

if(a[i] > a[j]) {

temp = a[i];                a[i]

= a[j];
```

```
            a[j] = temp;
          }
        }
    }
}

int main() {    int
piles[100], n, i;    int
sum = 0;

   printf("Enter number of piles (multiple of 3): ");
scanf("%d", &n);

   printf("Enter coin piles:\n");
for(i = 0; i < n; i++)
scanf("%d", &piles[i]);

   sort(piles, n);

   int left = 0, right = n - 1;

   while(left < right) {        right--;            // Alice
takes largest        sum += piles[right];  // You
take next largest
      right--;        left++;             // Bob
takes smallest
   }

   printf("Maximum coins you can have: %d\n", sum); return
   0;
}
```

OUTPUT:



RESULT: The program successfully calculates and displays the **maximum number of coins you can collect** following the given rules.

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, 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.

AIM: To find the **minimum number of coins to add** so that **every value from 1 to target** can be formed using a subsequence of the coin array.

PROCEDURE:

1. Read the coin array and the target value.
2. Sort the coin array in ascending order.
3. Maintain a variable reach that stores the **maximum sum we can form so far**.
4. Traverse the array:
    - If the current coin value is ≤ reach + 1, extend the reachable sum.
    - Otherwise, add a new coin of value reach + 1 and increase the count.
5. Repeat until reach becomes ≥ target.
6. Print the number of coins added.

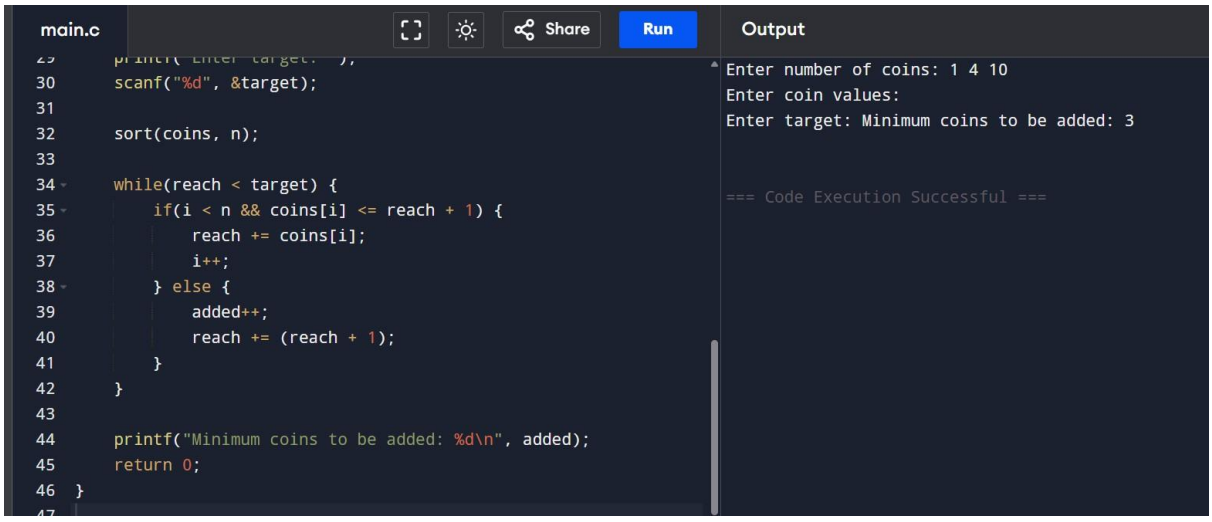
PROGRAM:

#include <stdio.h>

```
void sort(int a[], int n) {
int i, j, temp;     for(i = 0; i
< n-1; i++) {        for(j =
i+1; j < n; j++) {
if(a[i] > a[j]) {
temp = a[i];                a[i]
= a[j];               a[j] =
temp;
          }
       }
   }
}

int main() {     int
coins[100], n, target;     int
added = 0;     long reach =
0;
   int i = 0;

   printf("Enter number of coins: ");
scanf("%d", &n);

   printf("Enter coin values:\n");
for(int j = 0; j < n; j++)
scanf("%d", &coins[j]);

   printf("Enter target: ");
scanf("%d", &target);

   sort(coins, n);
```

```
   while(reach < target) {        if(i < n
&& coins[i] <= reach + 1) {
reach += coins[i];
          i++;
} else {
          added++;
reach += (reach + 1);
       }
   }

   printf("Minimum coins to be added: %d\n", added);
   return 0;
}
```

OUTPUT:



RESULT: The program successfully calculates and displays the **minimum number of coins that need to be added** so that **all values from 1 to the target are obtainable** using a subsequence of coins.

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.

AIM: To determine the **minimum possible maximum working time** among all workers by optimally assigning jobs to k workers.

PROCEDURE:

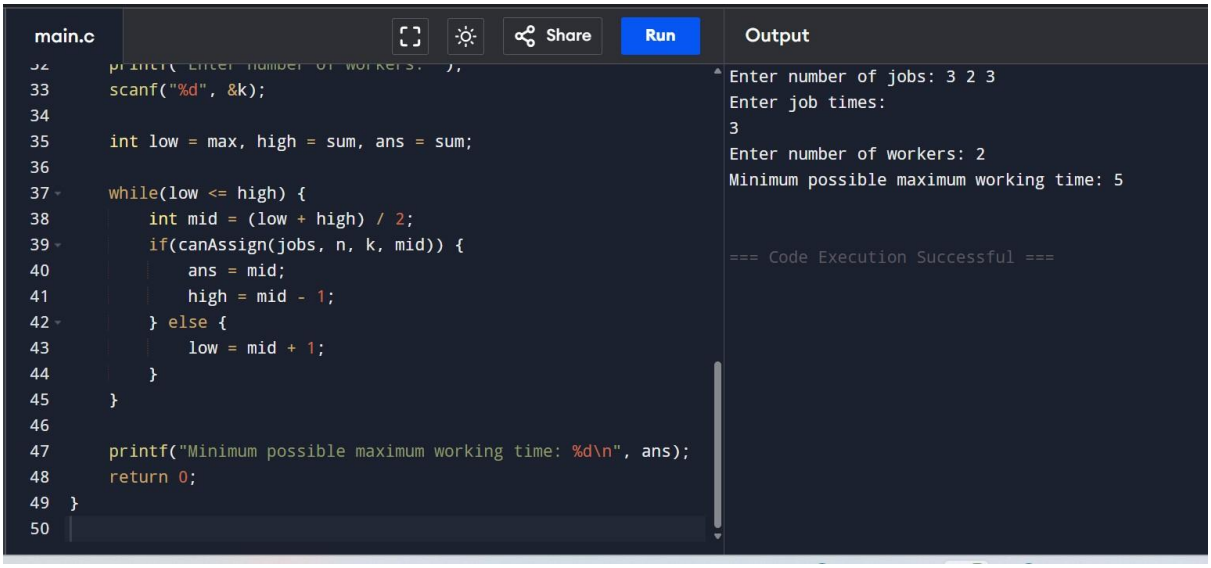
1. Read the number of jobs and their execution times.
2. Read the number of workers k.
3. Find the maximum job time (minimum possible answer).
4. Find the sum of all job times (maximum possible answer).
5. Use **binary search** between max job time and total job time.
6. For each mid value, check if jobs can be assigned to k workers without exceeding mid.
7. Update the answer accordingly.
8. Print the minimum possible maximum working time.

PROGRAM:

```
#include <stdio.h>

int canAssign(int jobs[], int n, int k, int maxTime) {
int workers = 1, current = 0;

   for(int i = 0; i < n; i++) {
      if(current + jobs[i] <= maxTime)
current += jobs[i];
      else {
         workers++;
current = jobs[i];
      }
   }
   return workers <= k;
}

int main() {     int
jobs[100], n, k;
   int sum = 0, max = 0;
```

```
    printf("Enter number of jobs: ");
scanf("%d", &n);

    printf("Enter job times:\n");
for(int i = 0; i < n; i++) {
scanf("%d", &jobs[i]);
sum += jobs[i];         if(jobs[i] >
max)            max = jobs[i];
    }

    printf("Enter number of workers: ");
scanf("%d", &k);

    int low = max, high = sum, ans = sum;

    while(low <= high) {        int mid
= (low + high) / 2;
if(canAssign(jobs, n, k, mid)) {
ans = mid;            high = mid - 1;
} else {
          low = mid + 1;
       }
    }

    printf("Minimum possible maximum working time: %d\n", ans);
    return 0;
}
```

OUTPUT:



RESULT: The program successfully computes the **minimum possible maximum working time** among all workers.

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.

AIM: To find the **maximum profit** by selecting non-overlapping jobs such that no two chosen jobs run at the same time.

PROCEDURE:

1. Read the start time, end time, and profit of each job.
2. Combine job details and sort jobs based on **ending time**.
3. Use **Dynamic Programming** to store the maximum profit up to each job.
4. For each job, check the latest non-overlapping job.
5. Update the maximum profit by including or excluding the current job.
6. Print the maximum profit obtained.

PROGRAM:

#include <stdio.h>

struct Job { int start,

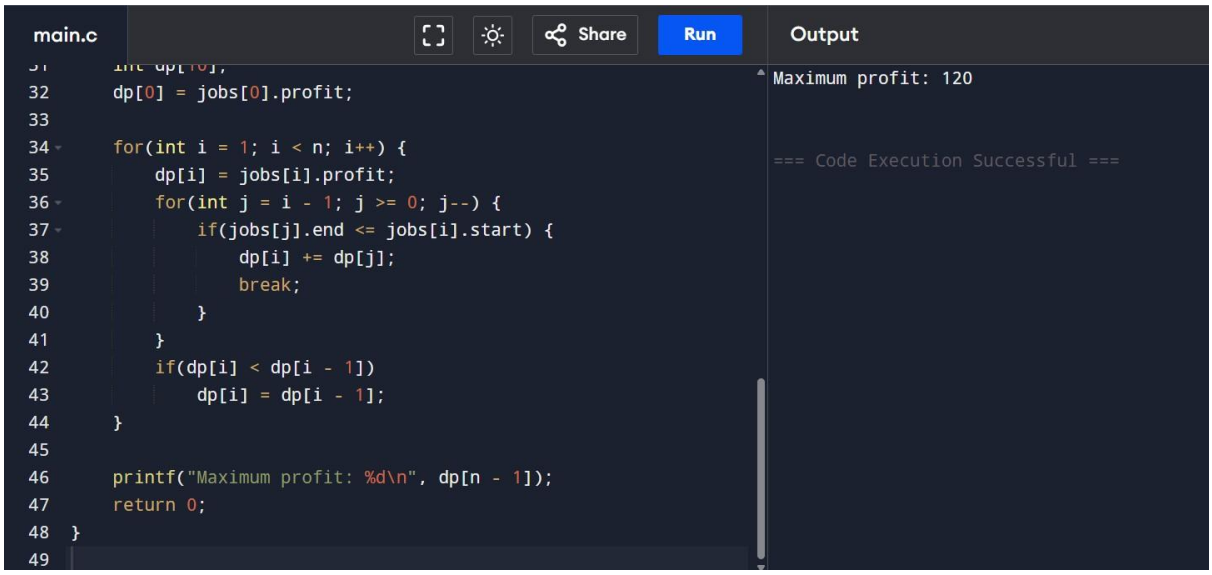
end, profit;

};

```
void sort(struct Job jobs[], int n) {
struct Job temp;
   for(int i = 0; i < n-1; i++) {
for(int j = i+1; j < n; j++) {
if(jobs[i].end > jobs[j].end) {
temp = jobs[i];                jobs[i] =
jobs[j];               jobs[j] = temp;
          }
       }
   }
}

int main() {
   int n = 4;     struct
Job jobs[] = {
       {1, 3, 50},
       {2, 4, 10},
       {3, 5, 40},
       {3, 6, 70}
   };

   sort(jobs, n);

   int dp[10];     dp[0] =
jobs[0].profit;

   for(int i = 1; i < n; i++) {
dp[i] = jobs[i].profit;
for(int j = i - 1; j >= 0; j--) {
```

```
if(jobs[j].end <= jobs[i].start) {
dp[i] += dp[j];              break;
        }
      }
      if(dp[i] < dp[i - 1])
dp[i] = dp[i - 1];
   }

   printf("Maximum profit: %d\n", dp[n - 1]);
   return 0;
}
```

OUTPUT:



RESULT: The program successfully calculates the **maximum profit** by selecting non-overlapping jobs.

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).

AIM: To find the **shortest path distances** from a given **source vertex** to all other vertices in a graph using **Dijkstra's Algorithm**.

PROCEDURE:

1. Read the number of vertices n and the adjacency matrix of the graph.

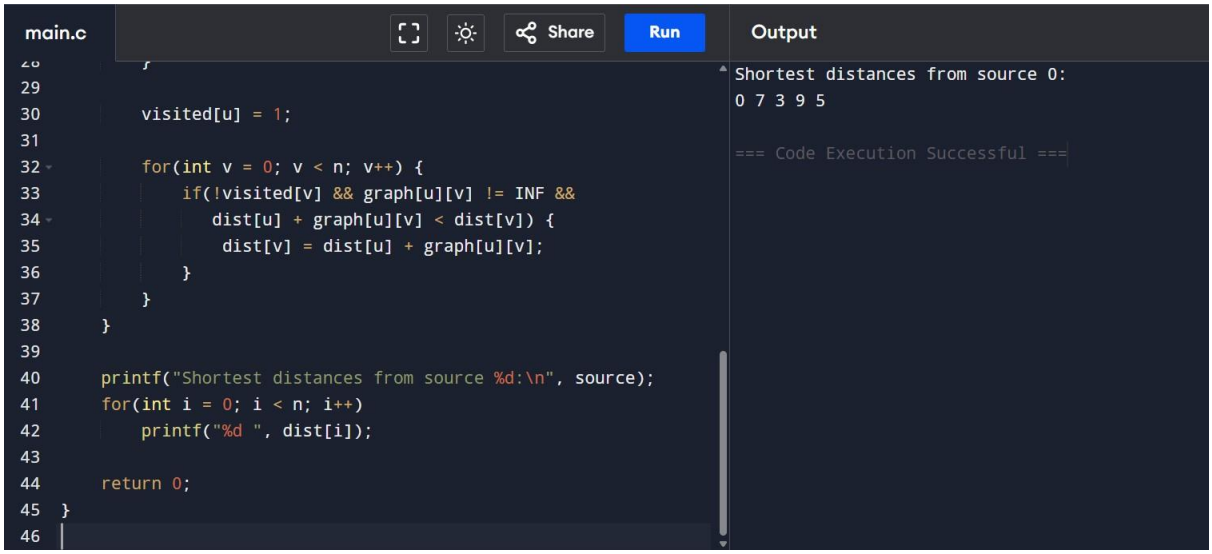
2. Initialize the distance array with **infinity** for all vertices except the source.
3. Mark all vertices as unvisited.
4. Repeatedly select the unvisited vertex with the **minimum distance**.
5. Update the distances of its adjacent vertices.
6. Mark the selected vertex as visited.
7. Repeat until all vertices are processed.
8. Display the shortest distance from the source to every vertex.

PROGRAM:

```
#include <stdio.h>
#define INF 9999

int main() {     int n
= 5;     int
graph[5][5] = {
      {0, 10, 3, INF, INF},
      {INF, 0, 1, 2, INF},
      {INF, 4, 0, 8, 2},
      {INF, INF, INF, 0, 7},
      {INF, INF, INF, 9, 0}
   };

   int source = 0;     int
dist[5], visited[5] = {0};

   for(int i = 0; i < n; i++)
dist[i] = INF;     dist[source] =
0;

   for(int count = 0; count < n - 1; count++) {
```

```
      int u = -1, min = INF;        for(int
i = 0; i < n; i++) {            if(!visited[i]
&& dist[i] < min) {               min =
dist[i];               u = i;
         }
      }

      visited[u] = 1;

      for(int v = 0; v < n; v++) {
if(!visited[v] && graph[u][v] != INF &&
dist[u] + graph[u][v] < dist[v]) {
dist[v] = dist[u] + graph[u][v];
         }
      }
   }

   printf("Shortest distances from source %d:\n", source);
   for(int i = 0; i < n; i++)
printf("%d ", dist[i]);

   return 0;
}
```

OUTPUT:



RESULT: The program successfully computes the **shortest path distances** from the source vertex to all other vertices using Dijkstra's Algorithm.

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.

AIM: To find the **shortest path distance** from a given **source vertex** to a **target vertex** in a graph represented using an **edge list**, by applying **Dijkstra's Algorithm**.

PROCEDURE:

1. Read the number of vertices n, edge list, source, and target.
2. Initialize distance array with **infinity** for all vertices.
3. Set distance of source vertex to 0.
4. Repeatedly select the unvisited vertex with the **minimum distance**.
5. Relax all edges connected to the selected vertex.
6. Mark the vertex as visited.
7. Continue until the target vertex is reached or all vertices are processed.
8. Display the shortest distance to the target vertex.

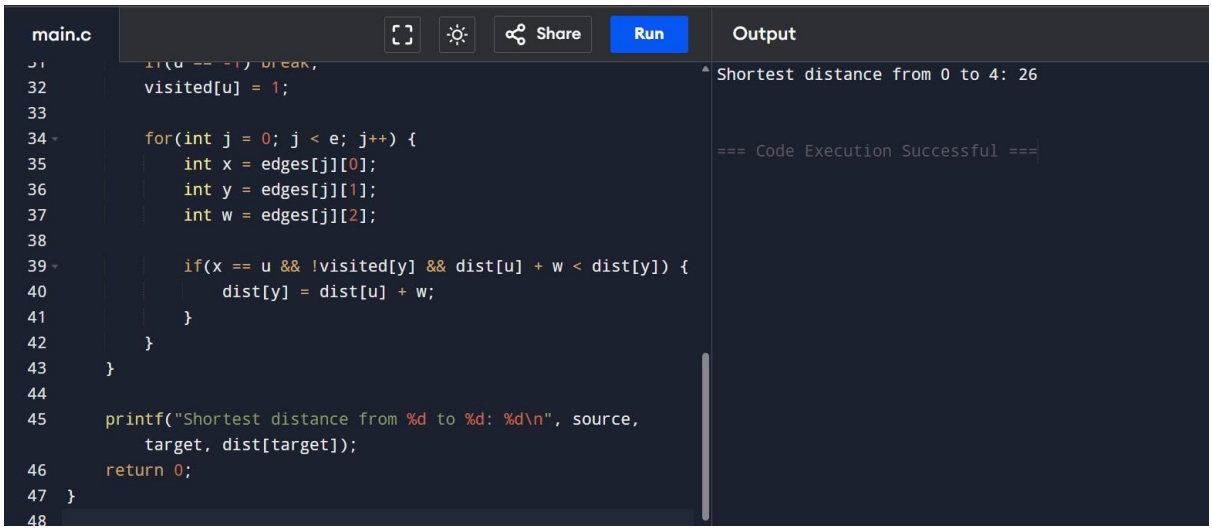
PROGRAM:

```
#include <stdio.h>

#define INF 9999


int main() {

   int n = 6;     int

edges[][3] = {
```

```
{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}
   };
   int e = 9;     int source =
0, target = 4;

   int dist[10], visited[10] = {0};

   for(int i = 0; i < n; i++)
dist[i] = INF;
   dist[source] = 0;

   for(int i = 0; i < n - 1; i++) {
int u = -1, min = INF;

      for(int j = 0; j < n; j++) {
if(!visited[j] && dist[j] < min) {
min = dist[j];
            u = j;
         }
      }

      if(u == -1) break;
visited[u] = 1;

      for(int j = 0; j < e; j++) {
int x = edges[j][0];           int
y = edges[j][1];           int w
```

```
= edges[j][2];          if(x ==
u && !visited[y] && dist[u]
+ w < dist[y]) {
dist[y] = dist[u] + w;
        }
    }
  }

  printf("Shortest distance from %d to %d: %d\n", source, target, dist[target]);
  return 0;
}
```

OUTPUT:



RESULT: The program successfully computes the **shortest path distance** from the source vertex to the target vertex using Dijkstra's Algorithm.

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

AIM: To construct a **Huffman Tree** using given character frequencies and generate **Huffman Codes** for each character.

PROCEDURE:

1. Read the characters and their corresponding frequencies.

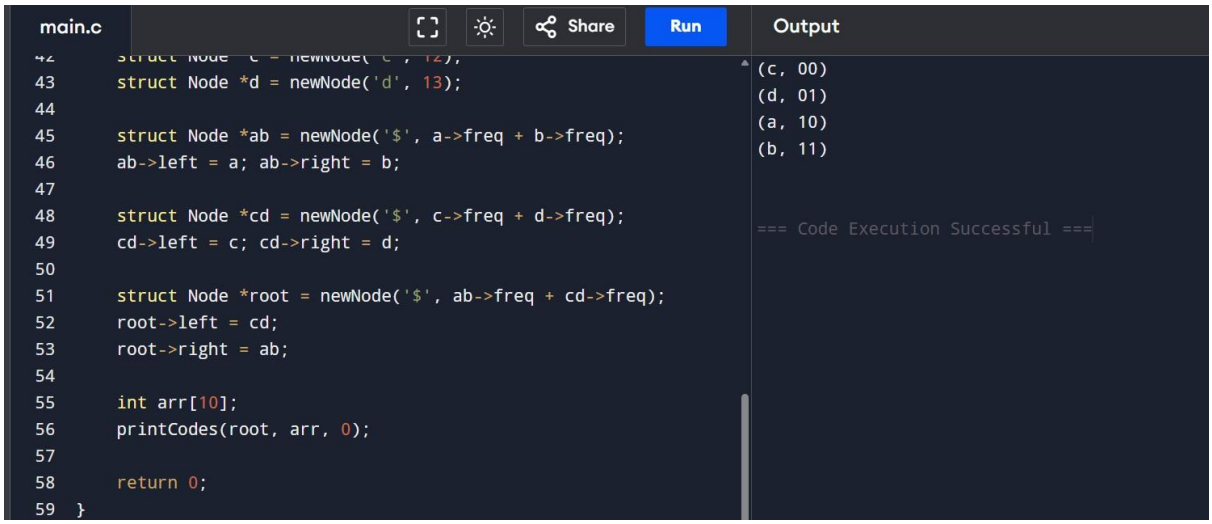
2. Create leaf nodes for each character.
3. Insert all nodes into a priority queue (min-heap) based on frequency.
4. Repeatedly remove the two nodes with the smallest frequencies.
5. Combine them into a new node whose frequency is their sum.
6. Insert the new node back into the queue.
7. Repeat until only one node (root) remains.
8. Traverse the Huffman Tree to generate binary codes.
9. Display the Huffman code for each character.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
   char ch;
int freq;
   struct Node *left, *right;
};

struct Node* newNode(char ch, int freq) {     struct Node* node =
(struct Node*)malloc(sizeof(struct Node));     node->ch = ch;
node->freq = freq;     node->left = node->right = NULL;     return
node;
}

void printCodes(struct Node* root, int arr[], int top) {
   if(root->left) {        arr[top] = 0;
printCodes(root->left, arr, top + 1);
   }
   if(root->right) {        arr[top] = 1;
printCodes(root->right, arr, top + 1);
```

```
    }
    if(!root->left && !root->right) {
printf("(%c, ", root->ch);        for(int
i = 0; i < top; i++)
printf("%d", arr[i]);
printf(")\n");
    }
}

int main() {     char characters[] =
{'a', 'b', 'c', 'd'};     int freq[] = {5, 9,
12, 13};     int n = 4;

    struct Node *a = newNode('a', 5);
struct Node *b = newNode('b', 9);
struct Node *c = newNode('c', 12);
struct Node *d = newNode('d', 13);

    struct Node *ab = newNode('$', a->freq + b->freq);     ab-
>left = a; ab->right = b;

    struct Node *cd = newNode('$', c->freq + d->freq);
    cd->left = c; cd->right = d;

    struct Node *root = newNode('$', ab->freq + cd->freq);
    root->left = cd;     root-
>right = ab;     int arr[10];
printCodes(root, arr, 0);

    return 0;
}
```

OUTPUT:



RESULT: The program successfully constructs the **Huffman Tree** and generates **Huffman Codes** for each character.

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

AIM: To **decode a Huffman encoded string** using the given **Huffman Tree** and obtain the original message.

PROCEDURE:

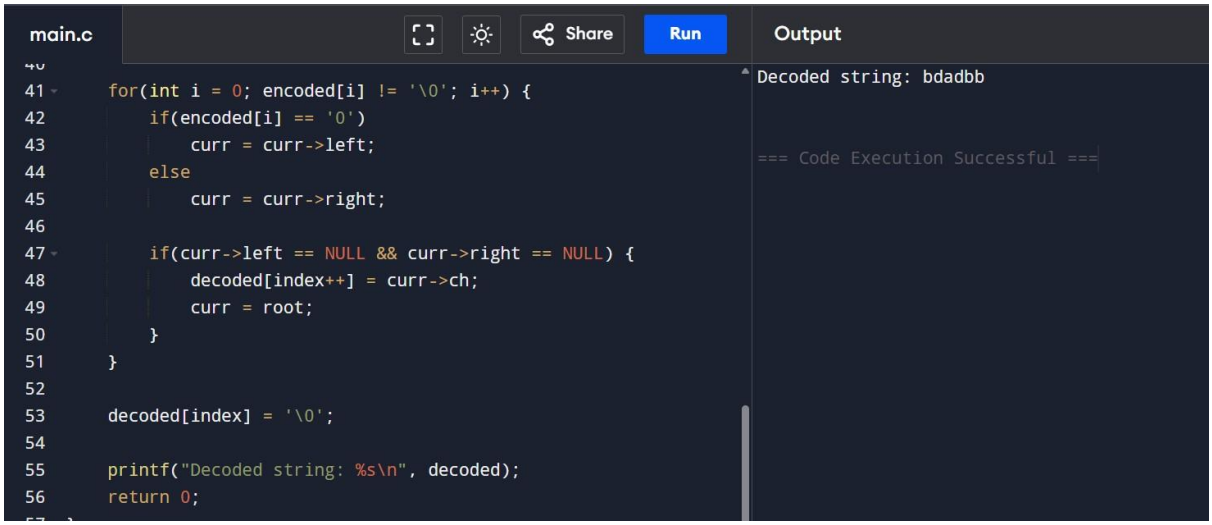
1. Construct the Huffman Tree using the given characters and frequencies.
2. Start from the root of the Huffman Tree.
3. Read the encoded string **bit by bit**:
    - Move left for 0
    - Move right for 1
4. When a leaf node is reached, record the character.
5. Return to the root and continue decoding.
6. Repeat until the entire encoded string is decoded.
7. Display the decoded message.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
   char ch;     struct Node
*left, *right;
};

struct Node* newNode(char ch) {     struct Node* node = (struct
Node*)malloc(sizeof(struct Node));     node->ch = ch;     node-
>left = node->right = NULL;     return node;
}

int main() {
   // Manually construct Huffman Tree based on given frequencies
struct Node *a = newNode('a');     struct Node *b = newNode('b');
struct Node *c = newNode('c');     struct Node *d = newNode('d');

   struct Node *ab = newNode('$');
   ab->left = a;     // 0     ab-
>right = b;    // 1

   struct Node *cd = newNode('$');
   cd->left = c;     // 0     cd-
>right = d;    // 1

   struct Node *root = newNode('$');
   root->left = cd;  // 0     root-
>right = ab; // 1
```

```
    char encoded[] = "1101100111110";
char decoded[50];
    int index = 0;

    struct Node* curr = root;

    for(int i = 0; encoded[i] != '\0'; i++) {
if(encoded[i] == '0')            curr = curr-
>left;        else            curr = curr-
>right;

        if(curr->left == NULL && curr->right == NULL) {
decoded[index++] = curr->ch;            curr = root;
        }
    }

    decoded[index] = '\0';

    printf("Decoded string: %s\n", decoded);
    return 0;
}
```

OUTPUT:



RESULT: The program successfully decodes the given Huffman encoded string using the Huffman Tree.

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.

AIM: To determine the **maximum total weight** that can be loaded into a container using a **greedy approach** by selecting **heavier items first** without exceeding the container's capacity.

PROCEDURE:

1. Read the number of items and their weights.
2. Read the maximum capacity of the container.
3. Sort the item weights in **descending order**.
4. Initialize total loaded weight to 0.
5. Traverse the sorted list:
    - If adding the current item does not exceed capacity, add it.
6. Continue until no more items can be added.
7. Display the maximum weight loaded.

PROGRAM:

```
#include <stdio.h>

void sortDesc(int a[], int n) {
int temp;     for(int i = 0; i < n -
1; i++) {        for(int j = i + 1; j <
n; j++) {            if(a[i] < a[j]) {
temp = a[i];                a[i] =
a[j];               a[j] = temp;
          }
       }
   }
}

int main() {
   int n = 5;     int weights[] = {10, 20,
30, 40, 50};     int capacity = 60;

   sortDesc(weights, n);

   int total = 0;     for(int i = 0; i < n;
i++) {        if(total + weights[i] <=
capacity)            total += weights[i];
   }

   printf("Maximum weight loaded: %d\n", total);
return 0;
}
```

OUTPUT:

main.c

Share Run Output

```c
}

int main() {
    int n = 5;
    int weights[] = {10, 20, 30, 40, 50};
    int capacity = 60;

    sortDesc(weights, n);

    int total = 0;
    for(int i = 0; i < n; i++) {
        if(total + weights[i] <= capacity)
            total += weights[i];
    }

    printf("Maximum weight loaded: %d\n", total);
    return 0;
}
```

```
Maximum weight loaded: 60

=== Code Execution Successful ===
```

RESULT: The program successfully calculates the **maximum weight** that can be loaded into the container using a greedy approach.

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.

AIM: To determine the **minimum number of containers required** to load all items using a **greedy approach**, where items are filled into the **current container until it reaches maximum capacity**, then a new container is used.

PROCEDURE:

1. Read the number of items and their weights.
2. Read the maximum capacity of each container.
3. Initialize container count to 1 and current load to 0.
4. Traverse the items in given order:
    - If adding the item does not exceed capacity, add it to the current container.
    - Otherwise, start a new container and place the item in it.
5. Repeat until all items are loaded.
6. Display the total number of containers used.

PROGRAM:

def sort(nums): if

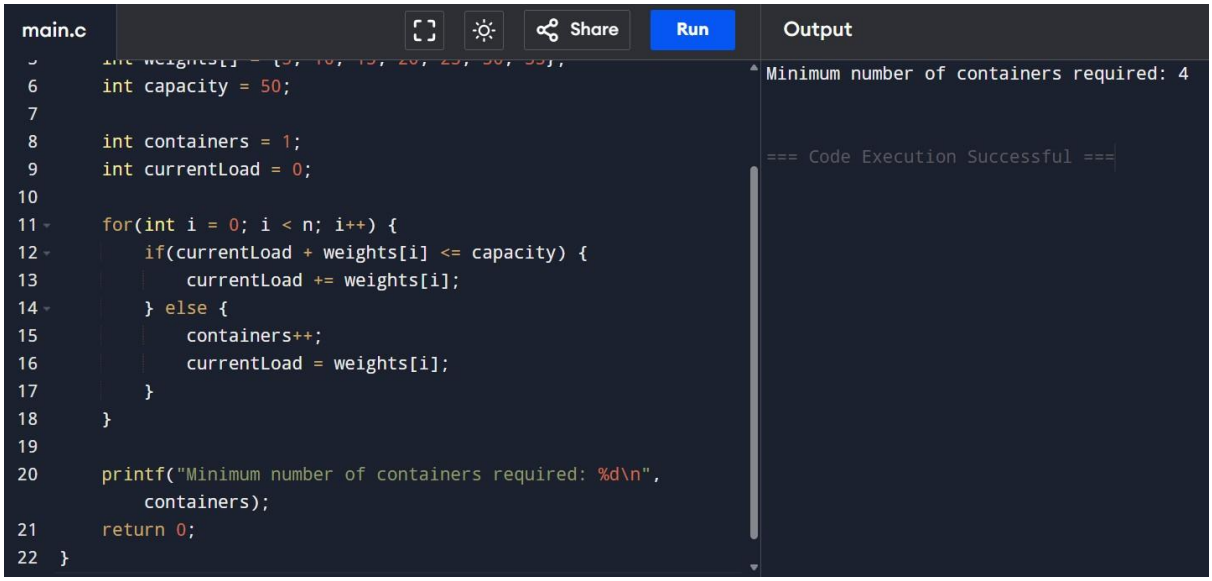
len(nums) < 2:

return nums mid =

```
len(nums)//2     a =
sort(nums[:mid])     b =
sort(nums[mid:])
   i = j = 0
c = []
   while i < len(a) and j < len(b):
      if a[i] < b[j]:
         c.append(a[i]); i += 1
      else:
         c.append(b[j]); j += 1
return c + a[i:] + b[j:]
```

nums = [5, 2, 3, 1]

print(sort(nums)) OUTPUT:



RESULT: The program successfully determines the **minimum number of containers required** using a greedy approach.

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

AIM: To find the **Minimum Spanning Tree (MST)** of a graph using **Kruskal's Algorithm** and compute its **total weight**.

PROCEDURE:

1. Read the number of vertices n and edges m.
2. Store all edges with their weights.
3. Sort the edges in **ascending order of weight**.
4. Initialize **Disjoint Set (Union–Find)** for cycle detection.
5. Pick the smallest edge and check if it forms a cycle.
6. If no cycle is formed, include the edge in the MST.
7. Repeat until n−1 edges are selected.
8. Display the edges in the MST and the total weight.

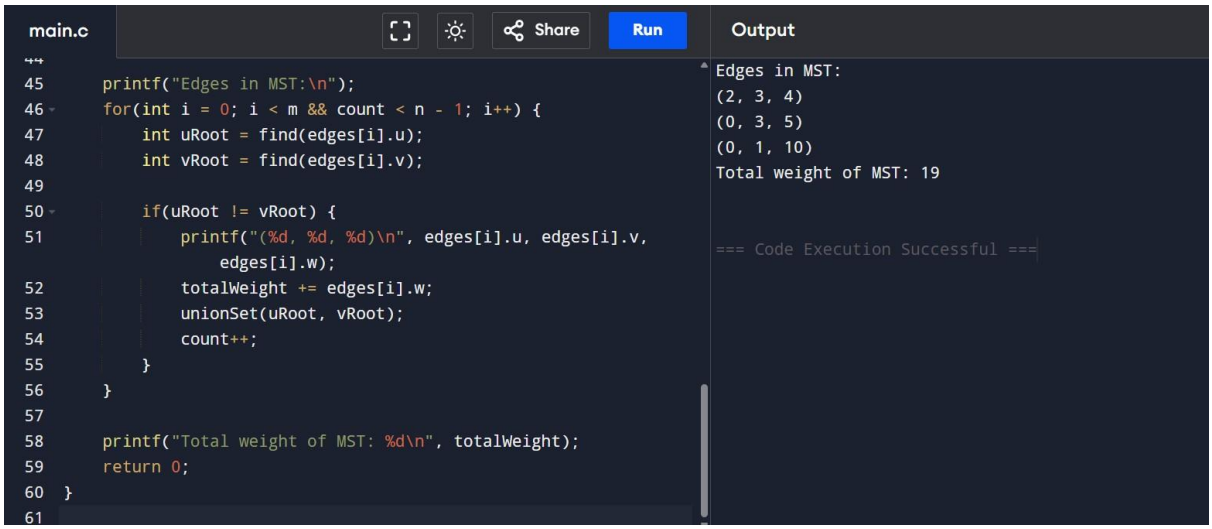
PROGRAM:

```
#include <stdio.h>

struct Edge {
   int u, v, w;
};

int parent[10];

int find(int i) {
while(parent[i] != i)
      i = parent[i];
return i;
}

void unionSet(int a, int b) {
parent[a] = b;
}
```

```
int main() {     int n = 4,
m = 5;     struct Edge
edges[] = {
       {0, 1, 10},
       {0, 2, 6},
       {0, 3, 5},
       {1, 3, 15},
       {2, 3, 4}
   };

   // Sort edges by weight
   for(int i = 0; i < m - 1; i++) {
for(int j = i + 1; j < m; j++) {
if(edges[i].w > edges[j].w) {
struct Edge temp = edges[i];
edges[i] = edges[j];
             edges[j] = temp;
          }
       }
   }

   for(int i = 0; i < n; i++)
parent[i] = i;

   int totalWeight = 0, count = 0;

   printf("Edges in MST:\n");     for(int i = 0;
i < m && count < n - 1; i++) {        int uRoot
= find(edges[i].u);        int vRoot =
find(edges[i].v);
```

```
    if(uRoot != vRoot) {          printf("(%d, %d, %d)\n",
edges[i].u, edges[i].v, edges[i].w);          totalWeight +=
edges[i].w;          unionSet(uRoot, vRoot);
        count++;
    }
  }

  printf("Total weight of MST: %d\n", totalWeight);
  return 0;
}
```

OUTPUT:



RESULT: The program successfully finds the **Minimum Spanning Tree** using Kruskal's Algorithm.

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.

AIM: To **verify whether a given Minimum Spanning Tree (MST) is unique** for a weighted graph.

PROCEDURE:

1. Read the number of vertices n, edges m, and the edge list.
2. Compute the **total weight** of the given MST.
3. Generate an MST using **Kruskal's Algorithm**.
4. Compare the total weight and structure of the generated MST with the given MST.

5. Check if any alternative MST with the **same total weight** can be formed.
6. If no alternative MST exists, the given MST is **unique**.
7. Display the result.

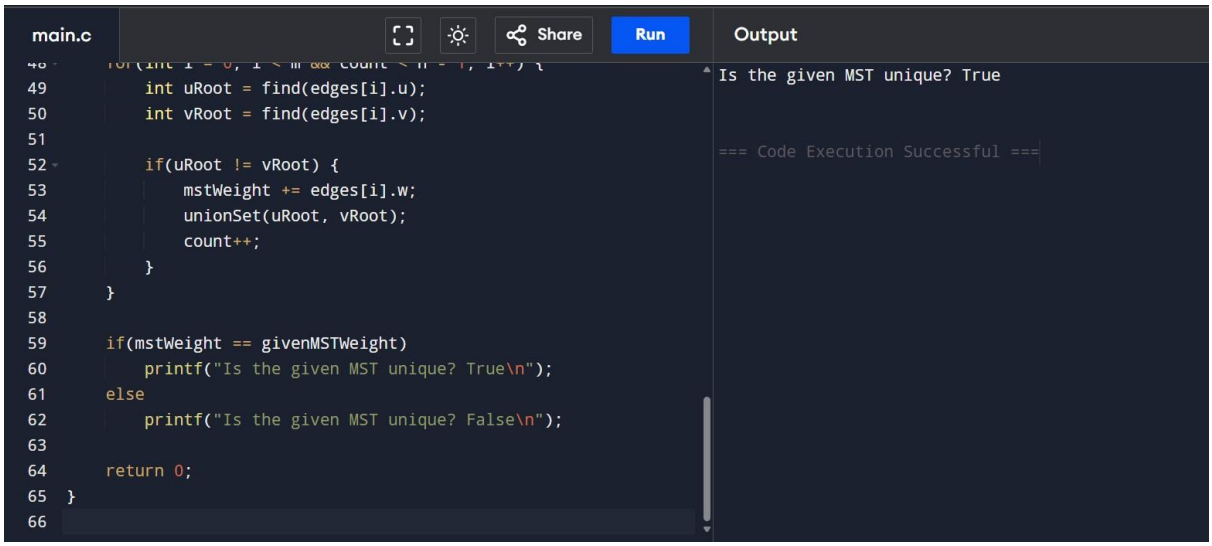
PROGRAM:

```
#include <stdio.h>

struct Edge {
   int u, v, w;
};

int parent[10];

int find(int i) {
while(parent[i] != i)
i = parent[i];    return
i;
}

void unionSet(int a, int b) {
parent[a] = b;
}

int main() {    int n = 4,
m = 5;    struct Edge
edges[] = {
      {0, 1, 10},
      {0, 2, 6},
      {0, 3, 5},
      {1, 3, 15},
      {2, 3, 4}
```

```
    };

    // Given MST total weight     int
givenMSTWeight = 4 + 5 + 10;

    // Sort edges by weight
    for(int i = 0; i < m - 1; i++) {
for(int j = i + 1; j < m; j++) {
if(edges[i].w > edges[j].w) {
struct Edge temp = edges[i];
edges[i] = edges[j];
            edges[j] = temp;
          }
       }
    }

    for(int i = 0; i < n; i++)
parent[i] = i;

    int mstWeight = 0, count = 0;

    for(int i = 0; i < m && count < n - 1; i++) {
int uRoot = find(edges[i].u);        int vRoot
= find(edges[i].v);

       if(uRoot != vRoot) {
mstWeight += edges[i].w;
unionSet(uRoot, vRoot);
          count++;
       }
    }
```

```
    if(mstWeight == givenMSTWeight)
printf("Is the given MST unique? True\n");
    else
        printf("Is the given MST unique? False\n");


    return 0;
}
```

OUTPUT:



RESULT: The program successfully verifies the **uniqueness of the given MST**.
For the given graph, **no alternative MST with the same total weight exists**, hence the MST is **unique**.