Experiment Title: Implementation of Programs on Coin Change Problem - Greedy Strategy

Aim/Objective: To understand the concept and implementation of Basic programs on Coin Change Problem - Greedy Strategy

Description: The students will understand and able to implement programs on Coin Change Problem -Greedy Strategy.

Pre-Requisites:

Knowledge: Coin Change Problem -Greedy Strategy in C/C++/Python

Tools: Code Blocks/Eclipse IDE

Pre-Lab:

Design and analyze the problem using greedy method.

Problem statement: Given a set of tasks, each with a start time si and finish time fi, the goal is to find the maximum number of non-overlapping intervals (tasks) that can be scheduled on a single machine.

Input: A list of intervals I = {I1, I2, . . . , In}, where each interval Ii is defined by two integers Ii = (si , fi) with si < fi . Two intervals Ii , Ij are compatible, i.e. disjoint, if they do not intersect (fi < sj or si < fj).

Output: A maximum subset of pairwise compatible (disjoint) intervals in I.

Example:

Input:

intervals = [(1, 3), (2, 4), (3, 5), (5, 7), (6, 8)]

Output:

Selected Intervals: [(1, 3), (3, 5), (5, 7)]

Number of Intervals: 3

- **Procedure/Program:**

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

typedef struct {
   int start;
   int finish;
} Interval;
```

```
int compare(const void *a, const void *b) {
   Interval *intervalA = (Interval *)a;
   Interval *intervalB = (Interval *)b;
   return intervalA->finish - intervalB->finish;
}

void intervalScheduling(Interval intervals[], int n) {
   qsort(intervals, n, sizeof(Interval), compare);

   Interval selectedIntervals[n];
   int count = 0;
   int lastFinishTime = -1;

   for (int i = 0; i < n; i++) {
      if (intervals[i].start >= lastFinishTime) {
         selectedIntervals[count++] = intervals[i];
         lastFinishTime = intervals[i].finish;
      }
   }

   printf("Selected Intervals: ");
   for (int i = 0; i < count; i++) {
      printf("(%d, %d) ", selectedIntervals[i].start, selectedIntervals[i].finish);
   }
   printf("\n");
   printf("Number of Intervals: %d\n", count);
}

int main() {
   Interval intervals[] = {{1, 3}, {2, 4}, {3, 5}, {5, 7}, {6, 8}};
   int n = sizeof(intervals) / sizeof(intervals[0]);

   intervalScheduling(intervals, n);

   return 0;
}
```

- **Data and Results:**

Data:

Given a list of intervals: `[(1, 3), (2, 4), (3, 5), (5, 7), (6, 8)]`.

Result:

Selected intervals: `[(1, 3), (3, 5), (5, 7)]`. Number of intervals: 3.

- **Analysis and Inferences:**

Analysis:

Greedy approach selects intervals based on earliest finish time.

Inferences:

Sorting intervals by finish time maximizes the number of non-overlapping intervals.

In-Lab:

Design and analyze the following problem using greedy method.

You are given an integer array coin representing coin of different denominations and an integer amount representing a total amount of money. Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot bemade up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

Example 1:

Input: coins = [1, 2, 5, 10], amount = 29

Output: 4

Explanation: 29 = 10 + 10 + 5 + 2 + 2

Example 2:

Input: coins = [5], amount = 11Output: -1

Example 3:

Input: coins = [2], amount = 0Output: 0

Constraints:

1 <= coins.length <= 12

1 <= coins[i] <= 2^{31} - 1

0 <= amount <= 10^4

- **Procedure/Program:**

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

int compare(const void *a, const void *b) {
   return (*(int*)b - *(int*)a);
}

int coinChange(int* coins, int coinsSize, int amount) {
   qsort(coins, coinsSize, sizeof(int), compare);
   int count = 0;

   for (int i = 0; i < coinsSize; i++) {
```

```
        while (amount >= coins[i]) {
            amount -= coins[i];
            count++;
        }
        if (amount == 0) {
            return count;
        }
    }

    return (amount == 0) ? count : -1;
}

int main() {
    int coins1[] = {1, 2, 5, 10};
    int amount1 = 29;
    int coinsSize1 = sizeof(coins1) / sizeof(coins1[0]);
    printf("Example 1 - Result: %d\n", coinChange(coins1, coinsSize1, amount1));

    int coins2[] = {5};
    int amount2 = 11;
    int coinsSize2 = sizeof(coins2) / sizeof(coins2[0]);
    printf("Example 2 - Result: %d\n", coinChange(coins2, coinsSize2, amount2));

    int coins3[] = {2};
    int amount3 = 0;
    int coinsSize3 = sizeof(coins3) / sizeof(coins3[0]);
    printf("Example 3 - Result: %d\n", coinChange(coins3, coinsSize3, amount3));

    return 0;
}
```

- **Data and Results:**

Data:

- Coins = [1, 2, 5, 10], Amount = 29.
- Coins = [5], Amount = 11.
- Coins = [2], Amount = 0.

Result:

- Example 1: 4 coins required to make amount 29.
- Example 2: Cannot make amount 11 using coins of 5.
- Example 3: Amount is already 0, no coins needed.

- **Analysis and Inferences:**

Analysis:

- Greedy approach works well with sorted coin denominations.
- Sorting coins helps in minimizing the number of coins.
- The greedy method is efficient for most cases with limited denominations.

Inferences:

- Greedy solution fails for some cases like Example 2.
- Optimal solutions depend on coin denominations and the amount.
- Dynamic programming might be necessary for complex cases.

Post-Lab:

Design and analyse the problem using greedy method.

Problem statement: Given a graph which represents a flow network where every edge has a capacity. Also, given two vertices source ‘s’ and sink ‘t’ in the graph, find the maximum possible flow from s to t with the following constraints:

- Flow on an edge doesn’t exceed the given capacity of the edge.
- Incoming flow is equal to outgoing flow for every vertex except s and t.

Input:

First line contains no. of vertices

Second line contains edges with their capacities

Next line contains source vertex and sink vertex

Output:

The maximum outflow from source to sink using greedy strategy.

- **Procedure/ Program:**

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

#define MAX_VERTICES 100

bool bfs(int graph[MAX_VERTICES][MAX_VERTICES], int source, int sink, int parent[],
int V) {
   bool visited[MAX_VERTICES] = {false};
   int queue[MAX_VERTICES];
   int front = 0, rear = 0;

   queue[rear++] = source;
   visited[source] = true;
   parent[source] = -1;

   while (front < rear) {
      int u = queue[front++];
      for (int v = 0; v < V; v++) {
         if (!visited[v] && graph[u][v] > 0) {
            queue[rear++] = v;
            visited[v] = true;
```

```
                parent[v] = u;
                if (v == sink) return true;
            }
        }
    }
    return false;
}

int fordFulkerson(int graph[MAX_VERTICES][MAX_VERTICES], int source, int sink, int
V) {
    int parent[MAX_VERTICES];
    int maxFlow = 0;

    while (bfs(graph, source, sink, parent, V)) {
        int pathFlow = INT_MAX;
        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
            pathFlow = (pathFlow < graph[u][v]) ? pathFlow : graph[u][v];
        }

        maxFlow += pathFlow;

        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
            graph[u][v] -= pathFlow;
            graph[v][u] += pathFlow;
        }
    }

    return maxFlow;
}

int main() {
    int V, E;
    scanf("%d", &V);
    scanf("%d", &E);

    int graph[MAX_VERTICES][MAX_VERTICES] = {0};

    for (int i = 0; i < E; i++) {
```

```
        int u, v, capacity;
        scanf("%d %d %d", &u, &v, &capacity);
        graph[u][v] = capacity;
    }

    int source, sink;
    scanf("%d %d", &source, &sink);

    int maxFlow = fordFulkerson(graph, source, sink, V);

    printf("Maximum Flow: %d\n", maxFlow);

    return 0;
}
```

- **Data and Results:**

Data:

Input includes vertices, edges with capacities, and source-sink vertices.

Result:

Maximum flow is computed from the source to the sink.

- **Analysis and Inferences:**

Analysis:

Ford-Fulkerson algorithm finds maximum flow using augmenting paths and BFS.

Inferences:

Greedy approach efficiently computes the maximum flow in flow networks.

- **Sample VIVA-VOCE Questions:**

1. Define feasible and optimal solution?

- Feasible solution: Satisfies all problem constraints.
- Optimal solution: Best solution in terms of objective function, among feasible solutions.

2. Specify the constrains of coin change problem?

- Given a set of coin denominations and a target amount.
- The sum of selected coins must equal the target amount.
- Each coin denomination can be used multiple times.

3. What is the time complexity for coin change problem?

- Time complexity is **O(n * m)**, where n is the target amount and m is the number of denominations.

Evaluator Remark (if Any):	**Marks Secured ___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.