Experiment Title: Implementation of Programs on Greedy method-II.

Aim/Objective: To understand the concept and implementation of Basic programs on Greedy method.

Description: The students will understand and able to implement programs on Greedy method.

Pre-Requisites:

Knowledge: Greedy Method and its related problems in C/Java/Python

Tools: Code Blocks/Eclipse IDE

Pre-Lab:

You are given a knapsack with a fixed capacity and a set of items, each with a weight and a value. Find the maximum value you can obtain by taking fractions of items in a greedy manner (maximize the value-to-weight ratio).

Input: A list of items with their values and weights, and the capacity of the knapsack.

Output: The maximum value that can be obtained.

Example:

Input:

Items = [(60, 10), (100, 20), (120, 30)], Capacity = 50

Output: 240.0

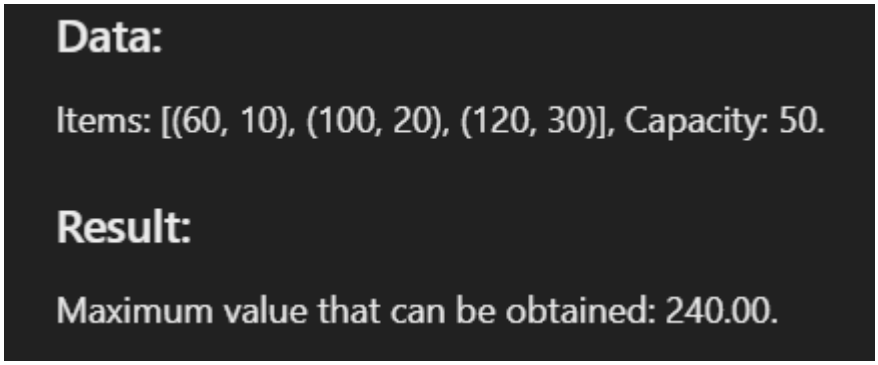
- **Procedure/Program:**

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

typedef struct {
   int value;
   int weight;
   float ratio;
} Item;
```

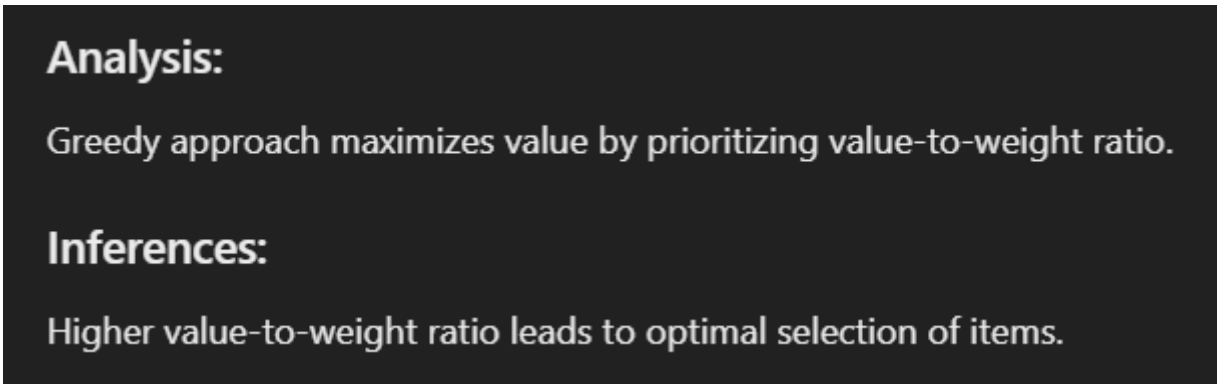
```
int compare(const void* a, const void* b) {

   float r1 = ((Item*)a)->ratio;
   float r2 = ((Item*)b)->ratio;
   return (r2 > r1) - (r1 > r2);
}

float fractionalKnapsack(Item items[], int n, int capacity) {
   for (int i = 0; i < n; i++) {
      items[i].ratio = (float) items[i].value / items[i].weight;
   }
   qsort(items, n, sizeof(Item), compare);

   float totalValue = 0;
   int remainingCapacity = capacity;

   for (int i = 0; i < n; i++) {
      if (remainingCapacity == 0) {
         break;
      }

      if (items[i].weight <= remainingCapacity) {
         totalValue += items[i].value;
         remainingCapacity -= items[i].weight;
      } else {
         totalValue += items[i].value * ((float)remainingCapacity / items[i].weight);
```

```
 remainingCapacity = 0;
        }
    }
    return totalValue;
}

int main() {
    Item items[] = {{60, 10}, {100, 20}, {120, 30}};
    int capacity = 50;
    int n = sizeof(items) / sizeof(items[0]);

    printf("Maximum value that can be obtained: %.2f\n", fractionalKnapsack(items, n, capacity));

    return 0;
}
```

- **Data and Results:**



- **Inference Analysis:**



In-Lab:

Given a set of characters with their frequencies, use the Huffman coding algorithm to generate the optimal binary prefix codes.

Input: A set of characters and their corresponding frequencies.

Output: The Huffman codes for each character.

Example:

Input:

char[] characters = {'F', 'G', 'H', 'I', 'J'};

int[] frequencies = {2, 7, 24, 14, 10};

Huffman Codes:

F: 000

G: 001

J: 01

I: 10

H: 11

Output: Huffman codes for the characters.

'F' → 000

'G' → 001

'J' → 01

'I' → 10

'H' → 11

- **Procedure/Program:**

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

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

struct MinHeap {
   int size;
   int capacity;
   struct Node **array;
};

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

struct MinHeap* createMinHeap(int capacity) {
   struct MinHeap* minHeap = (struct MinHeap*) malloc(sizeof(struct MinHeap));
   minHeap->capacity = capacity;
   minHeap->size = 0;
   minHeap->array = (struct Node**) malloc(minHeap->capacity * sizeof(struct Node*));
   return minHeap;
}

void swapNodes(struct Node** a, struct Node** b) {
   struct Node* temp = *a;
   *a = *b;
   *b = temp;
```

```
}
void minHeapify(struct MinHeap* minHeap, int idx) {
   int smallest = idx;
   int left = 2 * idx + 1;
   int right = 2 * idx + 2;

   if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]-
>freq)
      smallest = left;

   if (right < minHeap->size && minHeap->array[right]->freq < minHeap-
>array[smallest]->freq)
      smallest = right;

   if (smallest != idx) {
      swapNodes(&minHeap->array[smallest], &minHeap->array[idx]);
      minHeapify(minHeap, smallest);
   }
}

struct Node* extractMin(struct MinHeap* minHeap) {
   struct Node* temp = minHeap->array[0];
   minHeap->array[0] = minHeap->array[minHeap->size - 1];
   --minHeap->size;
   minHeapify(minHeap, 0);
   return temp;
}

void insertMinHeap(struct MinHeap* minHeap, struct Node* node) {
   ++minHeap->size;
   int i = minHeap->size - 1;
   while (i && node->freq < minHeap->array[(i - 1) / 2]->freq) {
      minHeap->array[i] = minHeap->array[(i - 1) / 2];
      i = (i - 1) / 2;
   }
   minHeap->array[i] = node;
}
```

```
struct Node* buildHuffmanTree(char characters[], int frequencies[], int size) {
   struct Node *left, *right, *top;
   struct MinHeap* minHeap = createMinHeap(size);

   for (int i = 0; i < size; ++i)
      minHeap->array[i] = createNode(characters[i], frequencies[i]);

   minHeap->size = size;
   for (int i = (minHeap->size - 2) / 2; i >= 0; --i)
      minHeapify(minHeap, i);

   while (minHeap->size != 1) {
      left = extractMin(minHeap);
      right = extractMin(minHeap);

      top = createNode('$', left->freq + right->freq);
      top->left = left;
      top->right = right;

      insertMinHeap(minHeap, top);
   }

   return extractMin(minHeap);
}

void printHuffmanCodes(struct Node* root, char* code, int top) {
   if (root->left) {
      code[top] = '0';
      printHuffmanCodes(root->left, code, top + 1);
   }

   if (root->right) {
      code[top] = '1';
      printHuffmanCodes(root->right, code, top + 1);
   }

   if (!root->left && !root->right) {
      code[top] = '\0';
```

```
        printf("'%c' → %s\n", root->character, code);
    }
}
int main() {
    char characters[] = {'F', 'G', 'H', 'I', 'J'};
    int frequencies[] = {2, 7, 24, 14, 10};
    int size = sizeof(characters) / sizeof(characters[0]);

    struct Node* root = buildHuffmanTree(characters, frequencies, size);

    char code[100];
    printHuffmanCodes(root, code, 0);

    return 0;
}
```

- **Data and Results:**

Data:

Input characters: {'F', 'G', 'H', 'I', 'J'}, frequencies: {2, 7, 24, 14, 10}.

Result:

Huffman codes: 'F' → 000, 'G' → 001, 'J' → 01, 'I' → 10, 'H' → 11.

- **Analysis and Inferences:**

Analysis:

Huffman algorithm optimally assigns shorter codes to higher frequencies.

Inferences:

Efficient compression achieved by assigning codes based on frequencies.

Post-Lab:

There are n people whose IDs go from 0 to n - 1 and each person belongs exactly to one group. Given the array group Sizes of length n telling the group size each person belongs to, return the groups there are and the people's IDs each group includes. You can return any solution in any order and the same applies for IDs. Also, it is guaranteed that there exists at least one solution.

Example 1:

Input: group_Sizes = [3, 3, 3, 3, 3, 1, 3]

Output: [[5], [0, 1, 2], [3, 4, 6]]

Explanation: Other possible solutions are [[2, 1, 6], [5], [0, 4, 3]] and [[5], [0, 6, 2], [4, 3, 1]].

Example 2:

Input: group_Sizes = [2, 1, 3, 3, 3, 2]

Output: [[1], [0, 5], [2, 3, 4]]

- **Procedure/Program:**

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

void groupThePeople(int* group_Sizes, int n) {
   int **result = (int**)malloc(n * sizeof(int*));
   int *groupCount = (int*)calloc(n, sizeof(int));

   for (int i = 0; i < n; i++) {
      int size = group_Sizes[i];
      if (groupCount[size] == 0) {
         result[size] = (int*)malloc(size * sizeof(int));
      }
      result[size][groupCount[size]] = i;
      groupCount[size]++;
   }

   for (int i = 0; i < n; i++) {
      if (groupCount[i] > 0) {
         printf("[");
         for (int j = 0; j < groupCount[i]; j++) {
            printf("%d ", result[i][j]);
```

```
            }
            printf("]\n");
        }
    }

    free(result);
    free(groupCount);
}

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

    int* group_Sizes = (int*)malloc(n * sizeof(int));

    for (int i = 0; i < n; i++) {
        scanf("%d", &group_Sizes[i]);
    }

    groupThePeople(group_Sizes, n);

    free(group_Sizes);

    return 0;
}
```

- **Data and Results:**

Data: Input contains group sizes for people, ranging from 0 to n-1.

Result: Groups are printed based on their respective group sizes.

- **Analysis and Inferences:**

Analysis: Time complexity is O(n), efficient for grouping people with sizes.

Inferences: Solution works well with varying group sizes for multiple people.

- **Sample VIVA-VOCE Questions:**

1. How does a Greedy algorithm make decisions at each step?

Greedy Algorithm Decisions: It makes locally optimal choices at each step, hoping to find a global optimum.

2. Does the greedy approach consider future data and choices?

Greedy Approach and Future Choices: No, it makes decisions based only on current data, not future possibilities.

3. Applications of Greedy Method?

Applications of Greedy Method:

- Kruskal's and Prim's algorithms (minimum spanning tree)
- Huffman coding (data compression)
- Activity selection problem
- Fractional knapsack problem

4. Difference between Divide & Conquer algorithm & Greedy algorithm?

Difference Between Divide & Conquer and Greedy Algorithms:

- **Divide & Conquer**: Breaks a problem into subproblems, solves them recursively, and combines solutions.
- **Greedy**: Makes a sequence of choices based on immediate benefits, not considering future steps.

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.