DIGITAL ASSIGNMENT 2 [GREEDY AND DYNAMIC PROGRAMMING]

CSE2012–DESIGN AND ANALYSIS OF ALGORITHMS (L25-26)[MRS GAYATHRI P]

FEBURARY 10, 2023

ANIRUDH VADERA

20BCE2940

QUESTION 1:

Write menu-driven program to implement 0/1 and fractional knapsack problem using greedy approach.

PSEUDOCODE:

Fractional Knapsack:

In this algorithm, the first step is to calculate the ratio of profit to weight for each item, and then sort the items based on this ratio in descending order. The next step is to iterate through each item and either take the entire item if its weight is less than or equal to the remaining capacity of the knapsack, or take a fraction of the item equal to the remaining capacity divided by the weight of the item. The algorithm continues this process until the knapsack is full or there are no more items left to take. The final result is the total profit obtained from taking the items.

- Input: The input of the program is the number of items (n), the maximum weight the knapsack can hold (W), and arrays of profits and weights for each item.
- Sorting: The algorithm sorts the items based on the ratio of profit to weight. The higher the ratio, the more valuable the item is considered to be. This step ensures that the most valuable items are selected first.
- Ratio calculation: The algorithm calculates the ratio of profit to weight for each item.
- Sorting according to the ratio: The items are sorted in descending order of their ratios.
- Used array: The algorithm declares a "used" array of size n to store the fraction of each item that will be added to the knapsack.
- Filling the knapsack: The algorithm fills the knapsack with items, starting with the most valuable (highest ratio) item. If the entire item can be added to the knapsack, it is added and the remaining weight is updated. If the item is too large to fit in the remaining space, the fraction of the item that fits is added and the loop is terminated.
- Output: The algorithm outputs the sorted arrays of profits and weights, the usage of each item (as a fraction), and the total profit obtained from filling the knapsack.
- Clean up: The algorithm frees the memory allocated for the ratio, weight, and profit arrays.
- **Time Complexity: The time complexity of the fractional knapsack algorithm is O(n log n) because of the sorting step.**

ANIRUDH VADERA
DIGITAL ASSIGNMENT 2 [GREEDY AND DYNAMIC PROGRAMMING]

0-1 Knapsack:

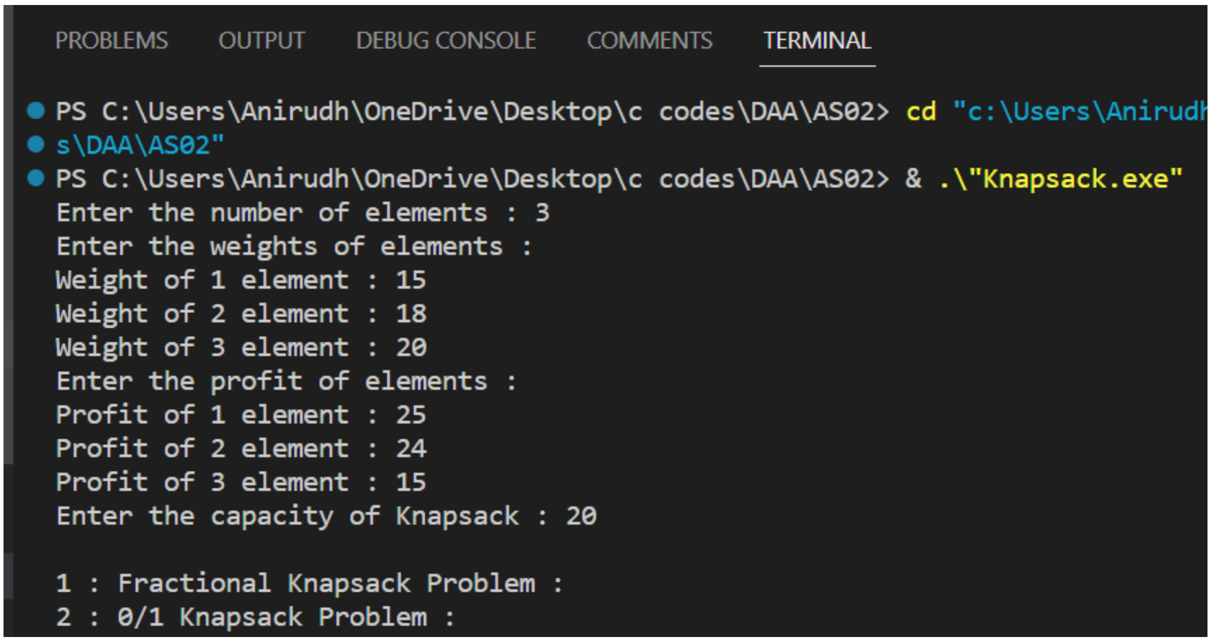
- In the 0/1 knapsack problem, we have a set of items, each with a weight and a value, and a knapsack that can hold a certain maximum weight. The objective is to fill the knapsack with items in such a way that the total value is maximum, but the total weight of the items must not exceed the knapsack's maximum weight. Each item can either be taken in its entirety or not taken at all, hence the name "0/1".
- This code uses a greedy approach to solve the 0/1 knapsack problem. The algorithm works as follows:
- The items are sorted based on their value-to-weight ratio, in descending order.
- A loop is used to iterate over each item in the sorted list of items.
- If the current item's weight is less than or equal to the remaining capacity of the knapsack, the entire item is added to the knapsack.
- If the current item's weight is greater than the remaining capacity of the knapsack, the item is not added to the knapsack.
- The total value of the items in the knapsack and the items' usage is then printed out.
- The greedy approach used in this code takes the items with the highest value-to-weight ratios first, ensuring that the knapsack is filled with items that provide the most value for the least weight.

SOURCE CODE:

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

void fractionalKnapsack(int *weight, int *profit, int W, int n)
{
    // Sorting according to ratio
    int *ratio = (int *)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++)
    {
        ratio[i] = profit[i] / weight[i];
    }

    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (ratio[i] < ratio[j])
            {
                int temp = ratio[j];
                ratio[j] = ratio[i];
                ratio[i] = temp;
                temp = weight[j];
                weight[j] = weight[i];
                weight[i] = temp;

                temp = profit[j];
                profit[j] = profit[i];
                profit[i] = temp;
            }
        }
    }
    float *used = (float *)malloc(n * sizeof(float));
    float totalProfit = 0;
    for (int i = 0; i < n; i++)
    {
        used[i] = 0.0;
    }
    for (int i = 0; i < n; i++)
    {
        if (weight[i] <= W)
        {
            W -= weight[i];
            used[i] = 1.0;
            totalProfit += used[i] * profit[i];
            printf("Added object (%d, %d) completely in the bag. Space left:
%d.\n", profit[i], weight[i], W);
        }
        else
        {
            used[i] = (float)W / weight[i];
            totalProfit += used[i] * profit[i];
            W = 0;
            printf("Added %f%% (%d, %d) of object in the bag.\n", used[i],
profit[i], weight[i]);
            break;
        }
    }
    printf("The Profit array is: \n");
    printf("[ ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", profit[i]);
    }
    printf("]\n");
    printf("The Weight array is: \n");
    printf("[ ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", weight[i]);
    }
    printf("]\n");
    printf("The usage of objects is as follows: \n");
    printf("[ ");
    for (int i = 0; i < n; i++)
    {
        printf("%f ", used[i]);
    }
    printf("]\n");
    printf("Filled the bag with objects worth %f\n", totalProfit);
    free(ratio);
    free(weight);
    free(profit);
}

void Knapsack01(int *weight, int *profit, int W, int n)
{
    // Sorting according to ratio
    int *ratio = (int *)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++)
    {
        ratio[i] = profit[i] / weight[i];
    }

    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (ratio[i] < ratio[j])
            {
                int temp = ratio[j];
                ratio[j] = ratio[i];
                ratio[i] = temp;

                temp = weight[j];
                weight[j] = weight[i];
                weight[i] = temp;

                temp = profit[j];
                profit[j] = profit[i];
                profit[i] = temp;
            }
        }
    }
    float *used = (float *)malloc(n * sizeof(float));
    float totalProfit = 0;
    for (int i = 0; i < n; i++)
    {
```

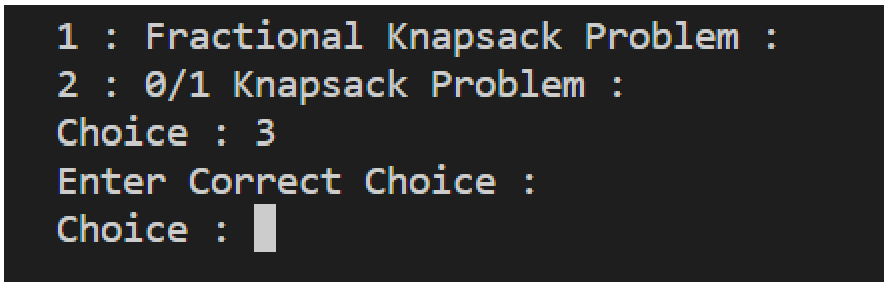
```c
        used[i] = 0.0;
    }
    for (int i = 0; i < n; i++)
    {
        if (weight[i] <= W)
        {
            W -= weight[i];
            used[i] = 1.0;
            totalProfit += used[i] * profit[i];
            printf("Added object (%d, %d) completely in the bag. Space left: %d.\n", profit[i], weight[i], W);
        }
    }
    printf("The Profit array is: \n");
    printf("[ ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", profit[i]);
    }
    printf("]\n");
    printf("The Weight array is: \n");
    printf("[ ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", weight[i]);
    }
    printf("]\n");
    printf("The usage of objects is as follows: \n");
    printf("[ ");
    for (int i = 0; i < n; i++)
    {
        printf("%f ", used[i]);
    }
    printf("]\n");
    printf("Filled the bag with objects worth %f\n", totalProfit);
}

int main()
{
    // Menu Driven Program
    int flag = 1;

    while (flag == 1)
    {
        int n;
        printf("Enter the number of elements : ");
        scanf("%d", &n);
        int *profit = (int *)malloc(n * sizeof(int));
        int *weight = (int *)malloc(n * sizeof(int));
        printf("Enter the weights of elements : \n");
        for (int i = 0; i < n; i++)
        {
            printf("Weight of %d element : ", i + 1);
            scanf("%d", &weight[i]);
        }
        printf("Enter the profit of elements : \n");
        for (int i = 0; i < n; i++)
        {
            printf("Profit of %d element : ", i + 1);
            scanf("%d", &profit[i]);
        }
        // The maximum weight u can take
        int W;
        printf("Enter the capacity of Knapsack : ");
        scanf("%d", &W);
        // Checking the choice
        printf("\n1 : Fractional Knapsack Problem : \n");
        printf("2 : 0/1 Knapsack Problem : \n");
        int choice;
        printf("Choice : ");
        scanf("%d", &choice);
        if (choice != 1 && choice != 2)
        {
            printf("Enter Correct Choice : \n");
            printf("Choice : ");
            scanf("%d", choice);
        }
        if (choice == 1)
        {
            fractionalKnapsack(weight, profit, W, n);
        }
        else
        {
            Knapsack01(weight, profit, W, n);
        }
        free(weight);
        free(profit);
        printf("Do You want to continue : 1: Yes , 0: No : ");
        scanf("%d", &flag);
    }

    return 0;
}
```

OUTPUT SCREENSHOT:



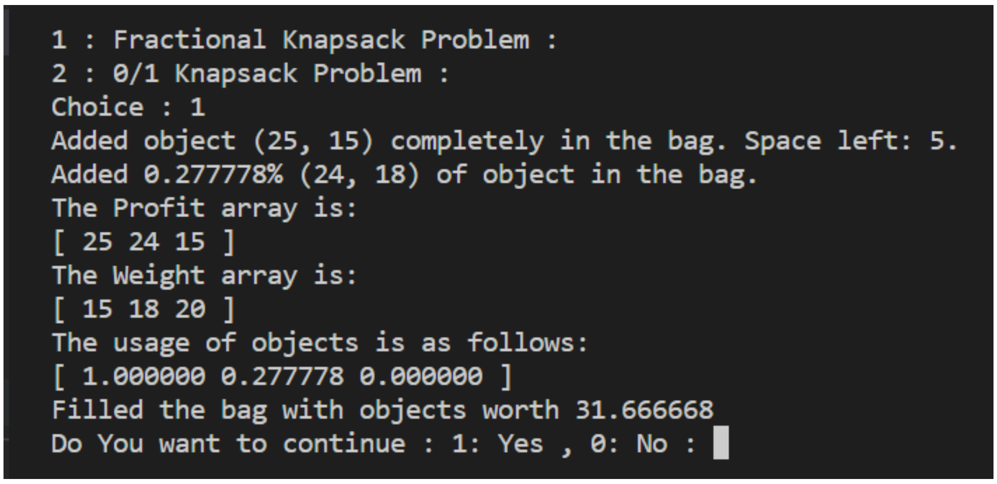
```
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02> cd "c:\Users\Anirudh
s\DAA\AS02"
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02> & .\"Knapsack.exe"
Enter the number of elements : 3
Enter the weights of elements :
Weight of 1 element : 15
Weight of 2 element : 18
Weight of 3 element : 20
Enter the profit of elements :
Profit of 1 element : 25
Profit of 2 element : 24
Profit of 3 element : 15
Enter the capacity of Knapsack : 20

1 : Fractional Knapsack Problem :
2 : 0/1 Knapsack Problem :
```

If wrong choice is given:



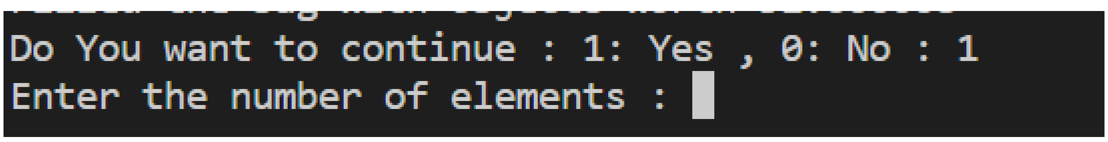
```
1 : Fractional Knapsack Problem :
2 : 0/1 Knapsack Problem :
Choice : 3
Enter Correct Choice :
Choice :
```

Fractional Knapsack Problem:

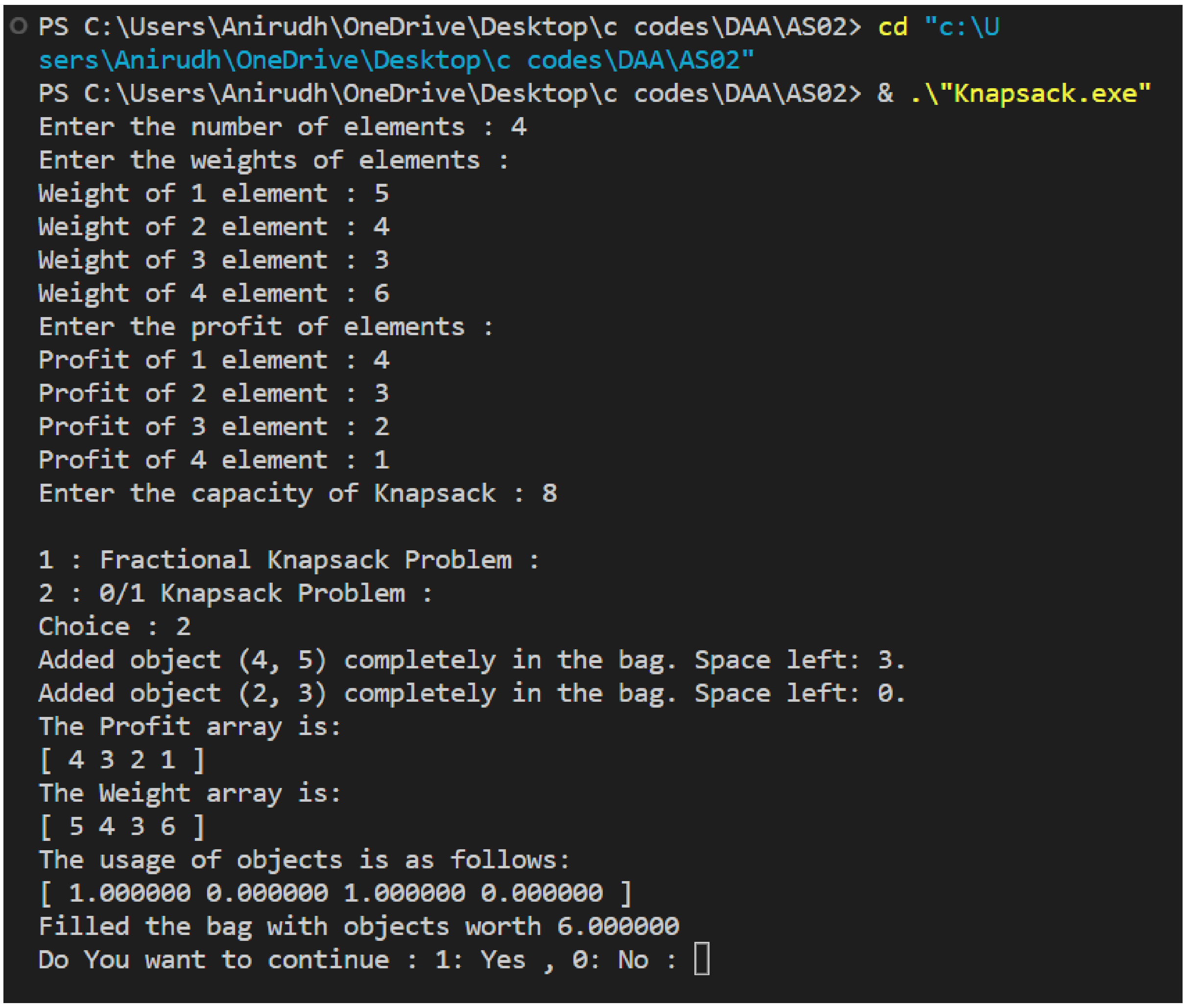


```
1 : Fractional Knapsack Problem :
2 : 0/1 Knapsack Problem :
Choice : 1
Added object (25, 15) completely in the bag. Space left: 5.
Added 0.277778% (24, 18) of object in the bag.
The Profit array is:
[ 25 24 15 ]
The Weight array is:
[ 15 18 20 ]
The usage of objects is as follows:
[ 1.000000 0.277778 0.000000 ]
Filled the bag with objects worth 31.666668
Do You want to continue : 1: Yes , 0: No :
```

Continuing our program for 0-1 knapsack:


```
Do You want to continue : 1: Yes , 0: No : 1
Enter the number of elements : 
```

0-1 Knapsack Problem:


```
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02> cd "c:\U
sers\Anirudh\OneDrive\Desktop\c codes\DAA\AS02"
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02> & .\"Knapsack.exe"
Enter the number of elements : 4
Enter the weights of elements :
Weight of 1 element : 5
Weight of 2 element : 4
Weight of 3 element : 3
Weight of 4 element : 6
Enter the profit of elements :
Profit of 1 element : 4
Profit of 2 element : 3
Profit of 3 element : 2
Profit of 4 element : 1
Enter the capacity of Knapsack : 8

1 : Fractional Knapsack Problem :
2 : 0/1 Knapsack Problem :
Choice : 2
Added object (4, 5) completely in the bag. Space left: 3.
Added object (2, 3) completely in the bag. Space left: 0.
The Profit array is:
[ 4 3 2 1 ]
The Weight array is:
[ 5 4 3 6 ]
The usage of objects is as follows:
[ 1.000000 0.000000 1.000000 0.000000 ]
Filled the bag with objects worth 6.000000
Do You want to continue : 1: Yes , 0: No : 
```

QUESTION 2:

Design and implement Huffman encoding algorithm using greedy approach.

PSEUDOCODE:

```
void HuffmanCodes(char data[],int freq[],int size){
        //In minheap we will store a MinHeapNode which contains two variables data and
frequency.Data represents the character.Also there exist two pointers left and right which is storing the
address of the node which is at the left of and at the right of the given node.
        struct MinHeapNode *left,*right,*top;

        //create a min heap using STL.compare function is ensuring that element should be arranged
according to frequency in the minheap.
        priority_queue<MinHeapNode*,vector<MinHeapNode*>,compare> minheap;


       // For each character create a leaf node and insert each leaf node in the heap.
        for(int  i=0;i<size;i++){
            minheap.push(new MinHeapNode(data[i],freq[i]));
         }

         //Iterate while size of min heap doesn't become 1
         while(minheap.size()!=1){
             //Extract two nodes from the heap.
             left = minheap.top();
             minheap.pop();

             right = minheap.top();
             minheap.pop();

             //Create a new internal node having frequency equal to the sum of
             two extracted nodes.Assign '$' to this node and make the two
extracted
             node as left and right children of this new node.Add this node to
the
             heap.
             tmp = new MinHeapNode('$',left->freq+right->freq);
             tmp->left = left;
             tmp->right = right;

             minheap.push(tmp);
         }
     }
```

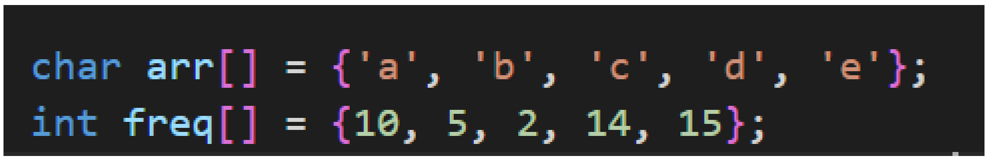
- Count the frequency of each character in the input text.
- Create a leaf node for each character, with the frequency of the character as its weight.
- Build a Huffman tree by merging the two lowest-weight leaf nodes into a new internal node, until there is only one tree node left. The weight of each internal node is the sum of the weights of its children.
- Assign binary codes to each character, with the help of the Huffman tree. Start from the root and traverse the tree, assigning 0 to the left child and 1 to the right child, until a leaf node is reached. The binary code for the character stored in that node is the path taken to reach it, in which the digits 0 and 1 are interpreted as left and right, respectively.

SOURCE CODE:

```
#include <bits/stdc++.h>
using namespace std;

struct MinHeapNode
{
    char data;
    unsigned freq;
    MinHeapNode *left, *right;

    MinHeapNode(char data, unsigned freq)
    {
        left = right = NULL;
        this->data = data;
        this->freq = freq;
    }
};

struct compare
{
    bool operator()(MinHeapNode *l, MinHeapNode *r)
    {
        return (l->freq > r->freq);
    }
};

void printCodes(struct MinHeapNode *root, string str)
{
    if (!root)
        return;
    if (root->data != '$')
        cout << root->data << ": " << str << "\n";
    printCodes(root->left, str + "0");
    printCodes(root->right, str + "1");
}

void HuffmanCodes(char data[], int freq[], int size)
{
    struct MinHeapNode *left, *right, *top;
    priority_queue<MinHeapNode *, vector<MinHeapNode *>, compare> minHeap;
    for (int i = 0; i < size; ++i)
        minHeap.push(new MinHeapNode(data[i], freq[i]));
    while (minHeap.size() != 1)
    {
        left = minHeap.top();
        minHeap.pop();
        right = minHeap.top();
        minHeap.pop();
        top = new MinHeapNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        minHeap.push(top);
    }
    printCodes(minHeap.top(), "");
}

int main()
{
    char arr[] = {'a', 'b', 'c', 'd', 'e'};
    int freq[] = {10, 5, 2, 14, 15};
    int size = sizeof(arr) / sizeof(arr[0]);
    HuffmanCodes(arr, freq, size);
}
```

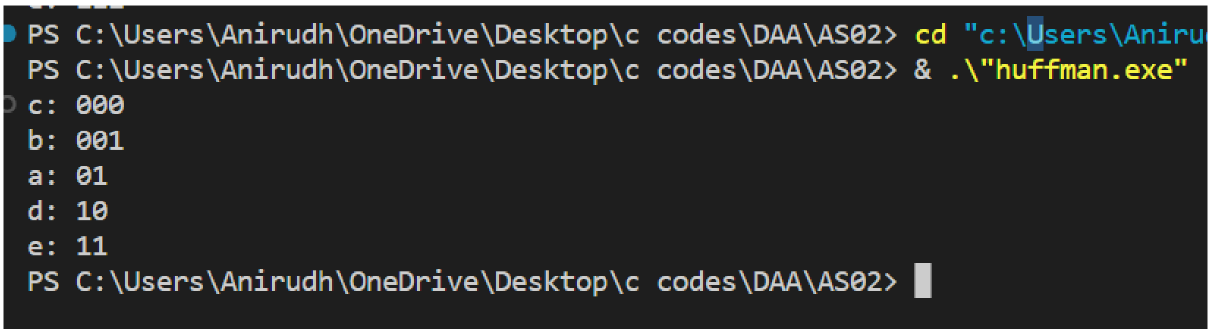
OUTPUT SCREENSHOT:

The frequency is given as :



```
char arr[] = {'a', 'b', 'c', 'd', 'e'};
int freq[] = {10, 5, 2, 14, 15};
```

The Prefix Codes are:



```
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02> cd "c:\Users\Aniru
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02> & .\"huffman.exe"
c: 000
b: 001
a: 01
d: 10
e: 11
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02>
```

QUESTION 3:

Implement 0/1 knapsack problem using dynamic programming technique.

PSEUDOCODE:

- The algorithm is a dynamic programming solution to the 0/1 knapsack problem. The 0/1 knapsack problem is a problem in combinatorial optimization where you are given a set of items with their weights and values, and a knapsack with a maximum weight capacity. The goal is to determine the maximum value that can be obtained by taking some or all of the items without exceeding the knapsack's weight capacity.
- The algorithm uses a two-dimensional dp array to store the maximum profit that can be obtained for each possible weight (0 to W) and for each item (0 to n). The array is initialized with all the values set to 0, then we iterate through the array to fill it up. The two nested for loops iterate through each item and each possible weight, and the values in the dp array are updated based on the current item and weight.

The logic of the algorithm can be explained in the following steps:

- If the current item is 0 or the current weight is 0, then the maximum profit that can be obtained is 0.
- If the weight of the current item is less than or equal to the current weight, then the maximum profit that can be obtained is the maximum of the profit of taking the current item and the profit obtained from the previous item and weight, or the profit obtained from the previous item and weight without taking the current item.
- If the weight of the current item is greater than the current weight, then the maximum profit that can be obtained is the profit obtained from the previous item and weight without taking the current item.
- Repeat the process for all items and all possible weights, and the final value in the dp array will be the maximum profit that can be obtained.
- **The algorithm has a time complexity of O(nW), where n is the number of items and W is the maximum weight capacity. The space complexity of the algorithm is O(nW).**

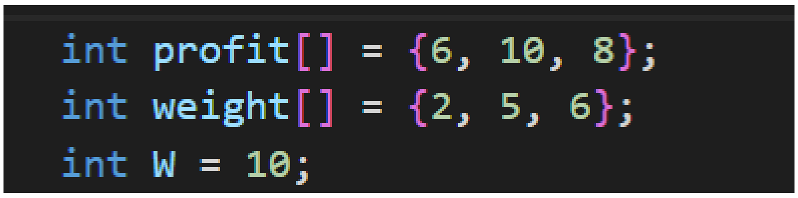
SOURCE CODE:

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

// Performing our logic
int knapSack(int W, int weight[], int profit[], int n)
{
    int i, w;
    // Creating a dp memotization array
    int **dp = (int **)malloc((n + 1) * sizeof(int *));
    for (int i = 0; i < n + 1; i++)
    {
        dp[i] = (int *)malloc((W + 1) * sizeof(int));
    }
    for (i = 0; i <= n; i++)
    {
        for (w = 0; w <= W; w++)
        {
            if (i == 0 || w == 0)
            {
                dp[i][w] = 0;
            }
            else if (weight[i - 1] <= w)
            {
                dp[i][w] = fmax(profit[i - 1] + dp[i - 1][w - weight[i - 1]],
dp[i - 1][w]);
            }
            else
            {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
    printf("The memo array is : \n");
    printf("[ \n");
    for (int i = 0; i < n + 1; i++)
    {
        printf("[ ");
        for (int j = 0; j < W + 1; j++)
        {
            printf("%d ", dp[i][j]);
        }
        printf("]\n");
    }
    printf("]\n");
    return dp[n][W];
}
// Driver code
int main()
{
    int profit[] = {6, 10, 8};
    int weight[] = {2, 5, 6};
    int W = 10;
    int n = sizeof(profit) / sizeof(profit[0]);
    printf("The Maximum Profit that can be carried out is : %d", knapSack(W,
weight, profit, n));
```

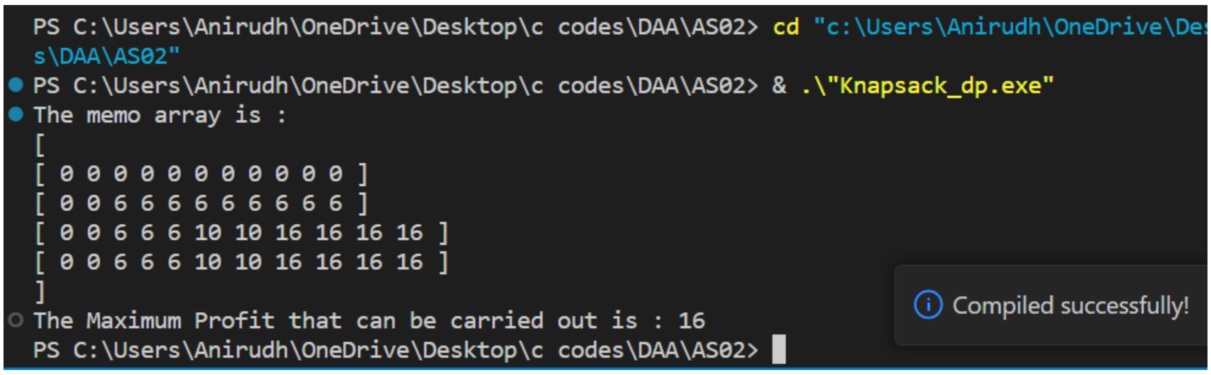
```
    return 0;
}
```

OUTPUT SCREENSHOT:

Profit , Weight array and knapsack weight:



```
    int profit[] = {6, 10, 8};
    int weight[] = {2, 5, 6};
    int W = 10;
```



```
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02> cd "c:\Users\Anirudh\OneDrive\De
s\DAA\AS02"
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02> & .\"Knapsack_dp.exe"
The memo array is :
[
[ 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 6 6 6 6 6 6 6 6 6 ]
[ 0 0 6 6 6 10 10 16 16 16 16 ]
[ 0 0 6 6 6 10 10 16 16 16 16 ]
]
The Maximum Profit that can be carried out is : 16
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02>
```

QUESTION 4:

Implement LCS problem using dynamic programming technique.

PSEUDOCODE:

- Read the lengths of two strings and input the strings.
- Allocate memory for a 2-dimensional memoization array "dp" using malloc.
- Initialize the first row and column of the "dp" array with zeros.
- Loop through both strings, comparing the characters at each position.
- If the characters are equal, update the current cell in the dp array to be 1 + the value of the cell above-left of it.
- If the characters are not equal, update the current cell in the dp array to be the maximum of the values in the cell above it and the cell to its left.
- Repeat this process for all cells in the dp array.
- The value in the bottom-right cell of the dp array represents the length of the longest common subsequence.
- If the length is not zero, traverse the dp array from the bottom-right cell to the top-left cell, recording the common characters in a separate string.
- Print the memoization array and the length of the longest common subsequence.
- If the length of the LCS is zero, print that there is no common subsequence.

Explanation:

The code solves the LCS problem by creating a 2-dimensional memoization array "dp" where each cell represents the length of the longest common subsequence ending at that point in the two input strings. The idea is to build this array from scratch, starting from the first characters of both strings and comparing each pair of characters in the strings.

Based on the comparison, the algorithm updates the values in the "dp" array using the following logic:

If the characters are equal, the value in the current cell is updated to be 1 + the value of the cell diagonally above-left of it.

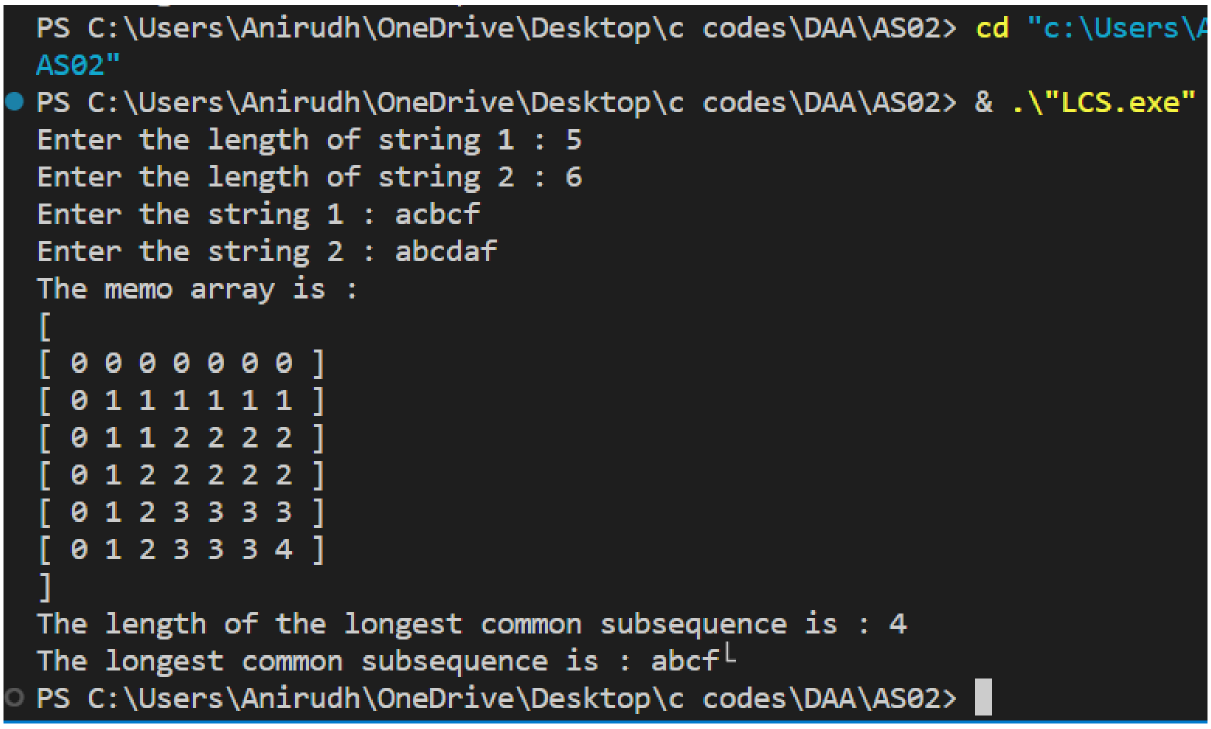
If the characters are not equal, the value in the current cell is updated to be the maximum of the values in the cell directly above it and the cell directly to its left.

This process is repeated for all cells in the dp array, and the value in the bottom-right cell represents the length of the longest common subsequence. If the length is not zero, the algorithm traces back through the dp array, recording the characters in the LCS in a separate string. Finally, the memoization array and the length of the LCS are printed.

SOURCE CODE:

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

int main()
{
    int l1, l2;
    printf("Enter the length of string 1 : ");
    scanf("%d", &l1);
    printf("Enter the length of string 2 : ");
    scanf("%d", &l2);
    char *str1 = (char *)malloc(l1 * sizeof(char));
    char *str2 = (char *)malloc(l2 * sizeof(char));
    printf("Enter the string 1 : ");
    scanf("%s", str1);
    printf("Enter the string 2 : ");
    scanf("%s", str2);
    // Creating a dp memotization array
    int **dp = (int **)malloc((l1 + 1) * sizeof(int *));
    for (int i = 0; i < l1 + 1; i++)
    {
        dp[i] = (int *)malloc((l2 + 1) * sizeof(int));
    }
    // Initializing dp array
    for (int i = 0; i < l1 + 1; i++)
    {
        dp[i][0] = 0;
    }
    for (int i = 0; i < l2 + 1; i++)
    {
        dp[0][i] = 0;
    }

    // Performing our logic
    for (int i = 1; i < l1 + 1; i++)
    {
        for (int j = 1; j < l2 + 1; j++)
        {
            if (str1[i - 1] == str2[j - 1])
            {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            }
            else
            {
                dp[i][j] = fmax(dp[i][j - 1], dp[i - 1][j]);
            }
        }
    }
    printf("The memo array is : \n");
    printf("[ \n");
    for (int i = 0; i < l1 + 1; i++)
    {
        printf("[ ");
        for (int j = 0; j < l2 + 1; j++)
        {
            printf("%d ", dp[i][j]);
        }
        printf("]\n");
    }
    printf("]\n");
    printf("The length of the longest common subsequence is : %d\n",
dp[l1][l2]);
    if (dp[l1][l2] != 0)
    {
        int l = dp[l1][l2];
        char *lcs = (char *)malloc(l * sizeof(char));
        int idx = 0;
        int curr = 0;
        for (int i = 0; i < l2 + 1; i++)
        {
            if (dp[l1][i] != curr)
            {
                lcs[idx++] = str2[i - 1];
                curr++;
            }
        }
        printf("The longest common subsequence is : %s", lcs);
    }
    else
    {
        printf("There is no common subsequence");
    }

    return 0;
}
```

OUTPUT SCREENSHOT:



```
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02> cd "c:\Users\A
AS02"
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02> & .\"LCS.exe"
Enter the length of string 1 : 5
Enter the length of string 2 : 6
Enter the string 1 : acbcf
Enter the string 2 : abcdaf
The memo array is :
[
[ 0 0 0 0 0 0 0 ]
[ 0 1 1 1 1 1 1 ]
[ 0 1 1 2 2 2 2 ]
[ 0 1 2 2 2 2 2 ]
[ 0 1 2 3 3 3 3 ]
[ 0 1 2 3 3 3 4 ]
]
The length of the longest common subsequence is : 4
The longest common subsequence is : abcf└
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02>
```

QUESTION 5:

Design an algorithm and implement travelling salesman problem using dynamic programming approach.

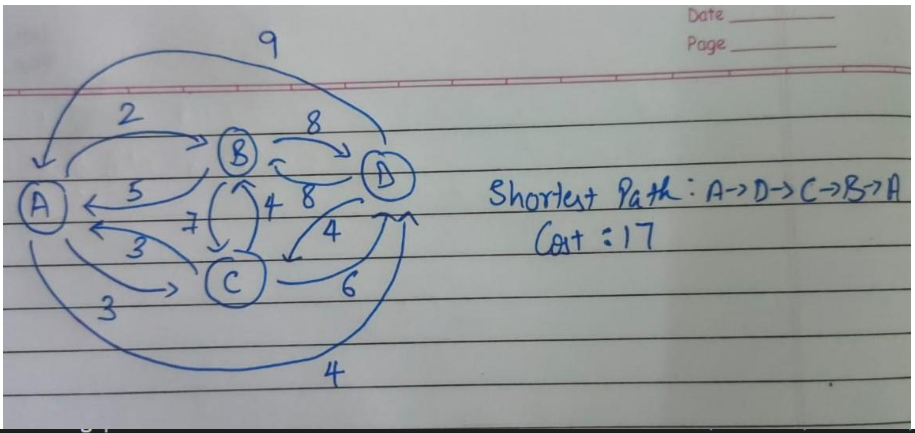
PSEUDOCODE:

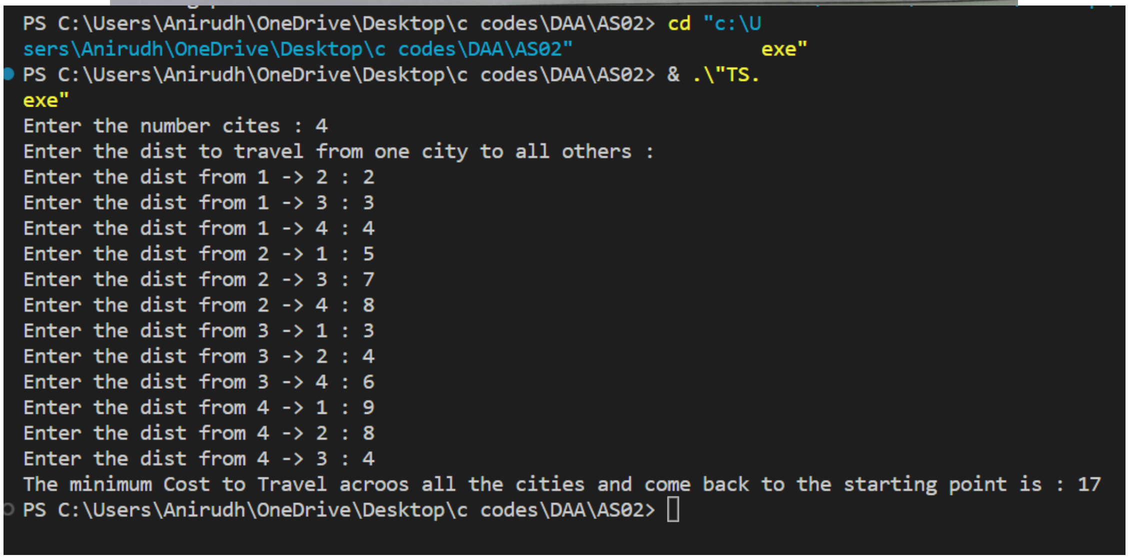
- n representing the number of cities to be visited.
- MAX representing the maximum possible cost to travel.
- memo array to store the intermediate results of subproblems to reduce the number of redundant calculations.
- Define a recursive function tsp:
- Inputs to this function are the current city i, the mask representing the set of cities visited till now, and the dist array representing the cost of traveling from one city to another.
- The function will return the minimum cost to travel from city i to all other cities, visiting all of them exactly once and returning to the starting city.
- If mask is equal to ((1<<i)|3), this means that all cities have been visited and the starting city has been reached, so return the cost to travel from the starting city to the current city.
- If the minimum cost has already been calculated, return the stored result from the memoization array.
- Iterate over all cities j such that j is not the starting city, not equal to the current city and has not been visited yet.
- Compute the minimum cost to travel from the current city to city j and call the tsp function recursively with j as the new current city and mask & (~(1<<i)) representing the new set of visited cities (excluding the current city).
- Return the minimum of all possible paths.
- In the main function, take input n representing the number of cities and dist array representing the cost of traveling from one city to another.
- Initialize the ans variable to MAX.
- Iterate over all cities i and calculate the minimum cost to travel from city i to all other cities, visiting all of them exactly once and returning to the starting city, by calling the tsp function with i as the starting city and ((1<<(n+1)) - 1) representing the set of all cities to be visited.
- Store the minimum of all possible paths in the ans variable.
- Return the value of ans.
- **The time complexity of this approach is O(2^n * n^2). The use of memoization reduces the number of redundant calculations, leading to a considerable reduction in the time required to solve the TSP problem.**

SOURCE CODE:

```
#include <iostream>

using namespace std;
const int n = 4;
const int MAX = 1000000;

// memoization for top down recursion
int memo[n + 1][1 << (n + 1)];

int tsp(int i, int mask, int **dist)
{
    if (mask == ((1 << i) | 3))
        return dist[1][i];
    // memoization
    if (memo[i][mask] != 0)
        return memo[i][mask];

    int res = MAX;

    for (int j = 1; j <= n; j++)
        if ((mask & (1 << j)) && j != i && j != 1)
            res = std::min(res, tsp(j, mask & (~(1 << i)), dist) +
dist[j][i]);
    return memo[i][mask] = res;
}
int main()
{
    int n;
    cout << "Enter the number cites : ";
    cin >> n;

    int **dist = (int **)malloc((n + 1) * sizeof(int *));

    cout << "Enter the dist to travel from one city to all others : " << endl;
    for (int i = 0; i < n + 1; i++)
    {
        dist[i] = (int *)malloc((n + 1) * sizeof(int));
        for (int j = 0; j < n + 1; j++)
        {
            if (i == 0 || j == 0)
            {
                dist[i][j] = 0;
            }
            else
            {
                if (i != j)
                {
                    cout << "Enter the dist from " << i << " -> " << j << " :
";
                    cin >> dist[i][j];
                }
                else
                {
                    dist[i][j] = 0;
                }
            }
        }
    }
    int ans = MAX;
    for (int i = 1; i <= n; i++)
        ans = std::min(ans, tsp(i, (1 << (n + 1)) - 1, dist) + dist[i][1]);
    cout << "The minimum Cost to Travel acroos all the cities and come back to
the starting point is : " << ans << endl;
    return 0;
}
```

OUTPUT SCREENSHOT:





```
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02> cd "c:\U
sers\Anirudh\OneDrive\Desktop\c codes\DAA\AS02"          exe"
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02> & .\"TS.
exe"
Enter the number cites : 4
Enter the dist to travel from one city to all others :
Enter the dist from 1 -> 2 : 2
Enter the dist from 1 -> 3 : 3
Enter the dist from 1 -> 4 : 4
Enter the dist from 2 -> 1 : 5
Enter the dist from 2 -> 3 : 7
Enter the dist from 2 -> 4 : 8
Enter the dist from 3 -> 1 : 3
Enter the dist from 3 -> 2 : 4
Enter the dist from 3 -> 4 : 6
Enter the dist from 4 -> 1 : 9
Enter the dist from 4 -> 2 : 8
Enter the dist from 4 -> 3 : 4
The minimum Cost to Travel acroos all the cities and come back to the starting point is : 17
PS C:\Users\Anirudh\OneDrive\Desktop\c codes\DAA\AS02> 
```