Lab -

DIP : Design and Analysis of Algorithm

AIM : Implement the following Greedy Algorithms

(1) Fractional Knapsack Problem

(2) Making a Change Problem

Name : Ambalia Harshit

Department : Computer Engineering (M.Tech sem II)

Roll no : MT001

Date : Jan 2024

Program 01 : Fractional Knapsack Problem

- Description :

    The fractional knapsack problem is a variant of the classic knapsack problem in which the items to be included in the knapsack can be divided into fractions, rather than being required to be taken in whole.

    The goal of the problem is to determine the combination of items that will maximize the total value of the knapsack, subject to the constraint that the total weight of the items must not exceed the capacity of the knapsack.

    In the fractional knapsack problem, the items have both a value and a weight, and the goal is to maximize the value of the items while staying under a certain weight capacity.

    The difference with the classic knapsack problem is that the item can be divided into fractions which allows the algorithm to take a fraction of an item if it can't take the whole item.

    The algorithm will sort the item based on the value per weight ratio, and start taking the item with the highest value per weight ratio, until the knapsack is full. It is called a "greedy" problem because the strategy for solving it is to always choose the item with the highest value-to-weight ratio, regardless of the remaining capacity of the knapsack or the value of the remaining items.

    This approach is considered "greedy" because it prioritizes maximizing the value of the knapsack in the short-term, without considering the overall impact on the final solution.

- Time Complexity :

    This algorithm takes O(n logn).

- Algorithm :

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)
for i = 1 to n
do x[i] = 0
weight = 0
for i = 1 to n
if weight + w[i] ≤ W then
x[i] = 1
weight = weight + w[i]
else
x[i] = (W - weight) / w[i]
weight = W
break
return x

- Code :

```
// Fractional Knapsack Problem
/* In the fractional knapsack problem, the items have both a value and a
weight,
and the goal is to maximize the value of the items while staying under a
certain
weight capacity. The difference with the classic knapsack problem is that
the item
can be divided into fractions which allows the algorithm to take a
fraction of an item
if it can't take the whole item. The algorithm will sort the item based on
the value per
weight ratio, and start taking the item with the highest value per weight
ratio, until
the knapsack is full.
*/

#include <stdio.h>
// #include "/home/hr/Documents/Semester_10/Lab_DAA/HRA.h"

void sort_double_with_multiple_arrays(int length, float
profit_weight_ratio[], float weight_array[], float profit_array[]) {
   for ( int i=1;i<length;i++ ) {
       float current_value = profit_weight_ratio[i];
       int current_index = weight_array[i];
```

```
        float current_third_value = profit_array[i];
        int j = i-1;

        while ( j>=0 && profit_weight_ratio[j]<current_value) {
            profit_weight_ratio[j+1] = profit_weight_ratio[j];
            weight_array[j+1] = weight_array[j];
            profit_array[j+1] = profit_array[j];
            j--;
        }

        profit_weight_ratio[j+1] = current_value;
        weight_array[j+1] = current_index;
        profit_array[j+1] = current_third_value;
    }
}

float fractional_knapsack(int length, float weight[], float profit[],
float knapsack) {
    // Finding the profit/weight ratio
    float profit_by_weight[length];
    for ( int i=0;i<length;i++ )
        profit_by_weight[i] = profit[i] / weight[i];

    // Sorting the array
    sort_double_with_multiple_arrays(length, profit_by_weight, weight,
profit);
    // for (int i=0;i<length;i++) {
    //     printf("%f, %f, %f\n", profit_by_weight[i], weight[i],
profit[i]);
    // }

    // implementing algo
    float total = 0;

    for ( int i=0;i<length;i++ ) {
        if(weight[i]<=knapsack) {
            total += profit[i];
            knapsack -= weight[i];
        }
        else {
```

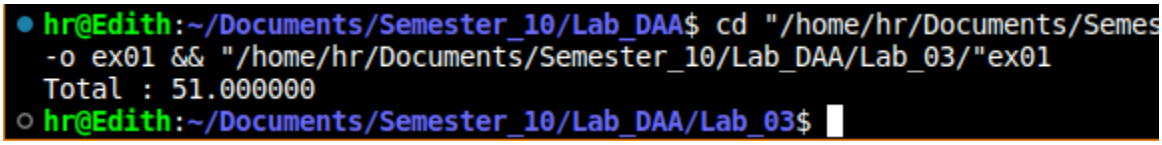
```
            total += ((knapsack/weight[i])*profit[i]);
            knapsack -= (knapsack/weight[i]);
        }
    }
    return total;
}

int main() {
    int n = 7;
    float kw = 15;
    float weight[] = {1, 3, 5, 4, 1, 3, 2};
    float profit[] = {5, 10, 15, 7, 8, 9, 4};

    float max_weight = fractional_knapsack(n, weight, profit, kw);
    printf("Total : %f\n", max_weight);

    return 0;
}
```

- Output Screen-shots :



Program 02 : Making a Change Problem

- Description :

The "making change" problem, also known as the coin change problem, is a problem in computer science and mathematics that involves finding the minimum number of coins or bills of different denominations needed to make a specific monetary amount.

The goal is to use the smallest number of coins possible to make the desired amount.

The problem can be solved using a greedy algorithm, which repeatedly takes the largest denomination of coin or bill that is less than or equal to the remaining amount to be made, until the desired amount is reached. For example, if the desired amount is $8 and the available denominations are [1, 5, 10], the algorithm would take one $5 coin and three $1 coins to make the desired $8.

The greedy approach for solving this problem is based on the observation that, for any given amount, it is always best to use the largest denomination coin available that is less than or equal to the remaining amount. This is because using a larger denomination coin will always result in a smaller number of coins needed to make the desired amount.

However, this approach is not always optimal and may not give the minimum number of coins for certain cases. For example, if the available denominations are [9,6,5] and the target amount is 11, the greedy algorithm will return [9,2] which is not the optimal solution.

- Algorithm :

Algorithm :
1. Sort the array of coins in decreasing order.
2. Initialize the result as empty.
3. Find the largest denomination that is smaller than the current amount.
4. Add found denomination to the result. Subtract value of found denomination from amount.
5. If the amount becomes 0, then print the result.
6. Else repeat steps 3 and 4 for the new value of V.

- Code :

```
// Making a Change Problem

#include <stdio.h>
// #include "/home/hr/Documents/Semester_10/Lab_DAA/HRA.h"

void insertionSortDescending(int arr[], int size) {
   /*
       Input: integer Array, Integer length of the array
       Output: void, prints array at each stage and level
       Explanation: This function will sort the given array in descending
order
   */
   for (int i = 1; i < size; i++) {
       int current = arr[i];
       int j = i - 1;

       while (j >= 0 && arr[j] < current) {
           arr[j + 1] = arr[j];
           j--;
       }
       arr[j + 1] = current;
   }
}

void mainking_change(int length, int coins[], int value) {
   int count_of_coins = 0;
   int temp_value = 0;
   int cost = value;
   for (int i=0;i<length;i++ ) {
       if(coins[i]<=value) {
           int temp = value/coins[i];
           value -= temp*coins[i];
           printf("We need %d Coin of rupees %d = %d\n", temp, coins[i],
(temp*coins[i]));
           temp_value += (temp*coins[i]);
           count_of_coins++;
       }
   }
```

```
   if(temp_value<cost) {
      printf("Can generate change for %d only with given coins, %d is
remaining.\n", temp_value, (cost-temp_value));
   }
}


int main() {
   int length = 5;
   int coins[] = {2, 5, 10, 25, 50};
   int value = 141;
   printf("Value : %d\n", value);
   printf("Coins Are : ");
   for( int i=0;i<length;i++ ) {
      printf("%d ", coins[i]);
   }
   printf("\n");
   insertionSortDescending(coins, length);
   mainking_change(length, coins, value);

   return 0;
}
```

- Output Screen-shots :

```
hr@Edith:~/Documents/Semester_10/Lab_DAA/Lab_03$ cd "/home/hr/Docum
ex02.c -o ex02 && "/home/hr/Documents/Semester_10/Lab_DAA/Lab_03/"e
Value : 141
Coins Are : 2 5 10 25 50
We need 2 Coin of rupees 50 = 100
We need 1 Coin of rupees 25 = 25
We need 1 Coin of rupees 10 = 10
We need 1 Coin of rupees 5 = 5
Can generate change for 140 only with given coins, 1 is remaining.
hr@Edith:~/Documents/Semester_10/Lab_DAA/Lab_03$
```

```
hr@Edith:~/Documents/Semester_10/Lab_DAA/Lab_03$ cd "/home/hr/Docum
ex02.c -o ex02 && "/home/hr/Documents/Semester_10/Lab_DAA/Lab_03/"e
Value : 142
Coins Are : 2 5 10 25 50
We need 2 Coin of rupees 50 = 100
We need 1 Coin of rupees 25 = 25
We need 1 Coin of rupees 10 = 10
We need 1 Coin of rupees 5 = 5
We need 1 Coin of rupees 2 = 2
hr@Edith:~/Documents/Semester_10/Lab_DAA/Lab_03$
```