Lecture 10-GREEDY ALGORTHIM

An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.
Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

Counting Coins
This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of € 1, 2, 5 and 10 and we are asked to count € 18 then the greedy procedure will be −
1 − Select one € 10 coin, the remaining count is 8
2 − Then select one € 5 coin, the remaining count is 3
3 − Then select one € 2 coin, the remaining count is 1
4 − And finally, the selection of one € 1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result. For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1) Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

Examples Most networking algorithms use the greedy approach. Here is a list of few of them −
- Travelling Salesman Problem
- Prim's Minimal Spanning Tree Algorithm
- Kruskal's Minimal Spanning Tree Algorithm
- Dijkstra's Minimal Spanning Tree Algorithm
- Graph - Map Coloring
- Graph - Vertex Cover

 Knapsack Problem
 Job Scheduling Problem
There are lots of similar problems that uses the greedy approach to find an optimum solution.

Fractional Knapsack Problem

Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don't take it.

Input:

Items as (value, weight) pairs

arr[] = {{60, 10}, {100, 20}, {120, 30}}

Knapsack Capacity, W = 50;

Output:

Maximum possible value = 220

by taking ite ms of weight 20 and 30 kg

In **Fractional Knapsack**, we can break items for maximizing the total value of knapsack. This problem in which we can break an item is also called the fractional knapsack problem.

Input :

Same as above

Output :

Maximum possible value = 240

By taking full items of 10 kg, 20 kg and

2/3rd of last item of 30 kg

A **brute-force solution** would be to try all possible subset with all different fraction but that will be too much time taking.
An **efficient solution** is to use Greedy approach. The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.

A simple code with our own comparison function can be written as follows, please see sort function more closely, the third argument to sort function is our comparison function which sorts the item according to value/weight ratio in non-decreasing order.
After sorting we need to loop over these items and add them in our knapsack satisfying above-mentioned criteria.

```
struct Item
{
   int value, weight;

   // Constructor
   Item(int value, int weight) : value(value), weight(weight)
   {}
};

// Comparison function to sort Item according to val/weight ratio
bool cmp(struct Item a, struct Item b)
{
   double r1 = (double)a.value / a.weight;
   double r2 = (double)b.value / b.weight;
   return r1 > r2;
}

/ Main greedy function to solve problem
double fractionalKnapsack(int W, struct Item arr[], int n)
{
   //    sorting Item on basis of ratio
   sort(arr, arr + n, cmp);

   //    Uncomment to see new order of Items with their ratio
   /*
   for (int i = 0; i < n; i++)
   {
      cout << arr[i].value << "  " << arr[i].weight << " : "
           << ((double)arr[i].value / arr[i].weight) << endl;
   }
   */

   int curWeight = 0;  // Current weight in knapsack
```

```
    double finalvalue = 0.0; // Result (value in Knapsack)

    // Looping through all Items
    for (int i = 0; i < n; i++)
    {
        // If adding Item won't overflow, add it completely
        if (curWeight + arr[i].weight <= W)
        {
            curWeight += arr[i].weight;
            finalvalue += arr[i].value;
        }
// If we can't add current Item, add fractional part of it
        else
        {
            int remain = W - curWeight;
            finalvalue += arr[i].value * ((double) remain / arr[i].weight);
            break;
        }
    }

    // Returning final value
    return finalvalue;
}

// driver program to test above function
int main()
{
    int W = 50;   //    Weight of knapsack
    Item arr[] = {{60, 10}, {100, 20}, {120, 30}};

    int n = sizeof(arr) / sizeof(arr[0]);
 cout << "Maximum value we can obtain = "
         << fractionalKnapsack(W, arr, n);
    return 0;
}
```

Output :

```
Maximum value in Knapsack = 240
```

As main time taking step is sorting, the whole problem can be solved in O(n log n) only.