Greedy Technique & Algorithms

Content

Introduction

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solutions are best fit for Greedy. If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming. But Greedy algorithms cannot always be applied.

Properties of Greedy technique are:-

1. Optimal Substructure - A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.
2. Greedy Workout - Works optimally.

Examples

Following are some standard algorithms that are Greedy algorithms.

1. **Kruskal's algorithm**

   In Kruskal's algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.

2. **Prim's algorithm**

   In Prim's algorithm also, we create a MST by picking edges one by one. We maintain two sets: a set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.

3. **Dijkstra's Shortest path algorithm**

   Dijkstra's algorithm is very similar to Prim's algorithm. The shortest-path tree is built up, edge by edge. We maintain two sets: a set of the vertices already included in the tree and the set of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.

4. **Huffman Coding**

   Huffman Coding is a lossless compression technique. It assigns variable-length bit codes to different characters. The Greedy Choice is to assign the least bit length code to the most frequent character.

Applications

1. **Activity Selection Problem**

   Problem

You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Example

Consider the following 3 activities sorted by
by finish time.

```
start[]  =  {10, 12, 20};
finish[] =  {20, 25, 30};
```

A person can perform at most **two** activities. The maximum set of activities that can be executed is {0, 2} [These are indexes in start[] and finish[]]

Approach

The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.

1. Sort the activities according to their finishing time.
2. Select the first activity from the sorted array and print it.
3. Do the following for the remaining activities in the sorted array.
    a. If the start time of this activity is greater than or equal to the finish time of the previously selected activity then select this activity and print it.

Implementation -
https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Greedy%20Technique/activitySelection.cpp

Working

Let the given set of activities be S = {1, 2, 3, …n} and activities are sorted by finish time. The greedy choice is to always pick activity 1. How come activity 1 always provides one of the optimal solutions. We can prove it by showing that if there is another solution B with the first activity other than 1, then there is also a solution A of the same size with activity 1 as the first activity. Let the first activity selected by B be k, then there always exist A = {B – {k}} U {1}.
(Note that the activities in B are independent and k has the smallest finishing time among all. Since k is not 1, finish(k) >= finish(1)).

2. **Minimum Absolute Difference in array**

Brute force

For each variable check the differences with other elements and maintain the minimum difference.
Time Complexity - $O(n^2)$
Space Complexity - O(1)

Optimal Solution

We will use the greedy technique.
First we will sort the array & find the consecutive difference of elements in array.
Iterate through the array & maintain a minimum.
Implementation -
https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Greedy%20Technique/minimumAbsoluteDifferenceInArray.cpp
Time Complexity - O(nlog(n))

Space Complexity - O(1)

3. 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 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.

Brute-force

A brute-force solution would be to try all possible subset with all different fraction but that will be too much time taking.

Efficient solution

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 the 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.

After sorting we need to loop over these items and add them in our knapsack satisfying above-mentioned criteria.

Implementation-
https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Greedy%20Technique/fractionalKnapSack.cpp

Time Complexity - O(nlog(n))

Space Complexity - O(1)

4. Weighted Job Scheduling

Problem

Given N jobs where every job is represented by following three elements of it.

1. Start Time
2. Finish Time
3. Profit or Value Associated (>= 0)

Find the maximum profit subset of jobs such that no two jobs in the subset overlap.

Approach

1) First sort jobs according to finish time.

2) Now apply the following recursive process.

```
// Here arr[] is array of n jobs
findMaximumProfit(arr[], n)
{
  a) if (n == 1) return arr[0];
  b) Return the maximum of following two profits.
     (i) Maximum profit by excluding current job, i.e.,
        findMaximumProfit(arr, n-1)
     (ii) Maximum profit by including the current job
}
```

How to find the profit including your current job?

The idea is to find the latest job before the current job (in sorted array) that doesn't conflict with the current job 'arr[n-1]'. Once we find such a job, we recur for all jobs till that job and add profit from the current job to result.

In the above example, "job 1" is the latest non-conflicting for "job 4" and "job 2" is the latest non-conflicting for "job 3".

Implementation - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Greedy%20Technique/weightedJobScheduling.cpp

The above solution may contain many overlapping subproblems. For example if lastNonConflicting() always returns previous job, then findMaxProfitRec(arr, n-1) is called twice and the time complexity becomes O(n*2n). As another example when lastNonConflicting() returns previous to previous job, there are two recursive calls, for n-2 and n-1. In this example case, recursion becomes same as Fibonacci Numbers.

So this problem has both properties of Dynamic Programming and Optimal substructure.

DP Solution

Implementation - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Greedy%20Technique/weightedJobSchedulingDP.cpp