***Greedy Algorithm:***

***Background:***

Solving the Best Optimization problem needs a series of steps, and in each step we can have multi - choices. For many Optimization problem, using Dynamic Programming Problem seems to be a storm in a teacup, we can use one more effective and more high efficient method.

***Definition - Greedy Algorithm:***

In each step, Greedy Algorithm choose the Best Solution in each step, which is to say, it always help choose Locality Best Solution in each step and hopes this kind of selection can help choose the Best Solution.

*Greedy Algorithm can not ensure to get the Best Solution, but it does help many problems to get the Best Solution.*

*Greedy Algorithm is a kind of Strong Algorithm Design Method which can be used to solve many problems including Minimum-Spanning-Tree Algorithm, Dijkstra Algorithm to solve the single shortest route and Chvatal Greedy Heuristic Method.*

***Activity Selection Problem:***

***Description:***

Multi-activities compete for shared resources, and the aim is to select one Biggest Compatible Activity Collection.

Assume there has only *one Activity Collection S = {a1, a2, a3, ..., an}*, and these activities share the same resource, however this kind of resource can only be used for sometime and only for one activity. *For activity ai, it has start time si and end time fi, here 0 <= si < fi < infinite.*

If it is selected, then the task ai would happen during the time period [ si, fi ). *If two activities ai and aj satisfy [ si, fi ) and [ sj, fj ) and both of which are not overlapped, then they are called compatible, which is to say, if si >= fj or sj >= fi, then ai and aj are compatible.* In the Activity Selection Problem, then we hope to choose the Biggest Compatible Activity Collection. *Assume that the activity has been sorted according to its Increasing Sequence:*

*f1 <= f2 <= f3 <= ... <= fn-1 <= fn*

*So the activity has been sorted means the started time and finished time of the activity has been sorted according to its finished time.*

***Example:***

*Activity Collection:*

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| si | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
| fi | 4 | 5 | 6 | 7 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

For this instance, sub - collection {a2, a9, a11} consists of different activities, but it is not the biggest collection, since the sub - collection {a1, a4, a8, a11} is bigger. Actually the sub - collection {a1, a4, a8, a11} is the bigger compatible one, and another bigger one is {a2, a4, a9, a11}.

***Solution:***

Normally, we try to divide the problem into two Sub - Question by using Dynamic Programming, and then combine two Best Solutions into one Best Solution of two Sub - Question. When make sure to use which problem to the Best Solution, then we need to take several questions into consideration.

*Greedy Algorithm needs us to take one selection into consideration (Greedy Selection), when makes Greedy Choice, one of the sub-question must be empty, therefore leave one non-empty sub-question.*

***Best Sub-Structure in Activity Selection Problem***

Assume that Sij presents that start after the end of activity ai, and end before the start of activity aj. Assume we hope to get one Biggest Compatible Activity Sub - Collection of Sij, and in further step, we assume that Aij is such Sub - Collection, including the activity aij. The Best Solution includes activity ak, then we get two separate Sub - Problem, find out the Biggest Compatible Activity Sub - Collection of Sik, and find out the Biggest Compatible Activity Sub - Collection of Skj. *(The Sik means these activities start after the end of activity ai and end before the start of activity ak. The Skj means these activities start after the end of activity ak and end before the start of activity aj.)*

Make *Aik = Aij intersection Sik, Akj = Aij intersection Sjk*, then Aik would include the activities end before the start of ak, Akj would include the activities start after the end of ak. Therefore, we have *Aij = Aik intersection { ak } intersection Akj*, and *among Sij, the Biggest Compatible Sub - Collection has | Aij | = | Aik | + | Akj | + 1 activities.*

*( By using the cut and paste method to verify that the Best Solution Aij contain the Best Solution of two Sub - Problem Sik and Skj.)*

Therefore, we can describe the Best Solution of Activity Selection Problem, and use Dynamic Programming Algorithm to solve Activity Selection Problem. *If we use c[ i, j ] to describe the size of Best Solution Sij, then we can get the Recursion:*

*c[ i, j ] = c[ i, k ] + c[ k, j ] + 1*

Of course, if we do not sure whether the Best Solution includes activity ak, then we need to consider all activities in Sij and find which activity can get the Best Solution, so

*c[ i, j ] = 0 (Sij equals to Empty)*

*c[ i, j ] = max{ c[ i, k ] + c[ k, j ] + 1 } (Sij does not equal to Empty.)*

After that, we can design the Recursive Algorithm with Memo or by using the Bottom to Top method to fill in the table.

***Greedy Selection***

For the activity selection, what is Greedy Selection ? We need to choose one activity, and after it has been selected, the remaining resources can be as much as other task uses.

*Consider all Selective Activities, some of which can be stopped at first. From intuition, we choose the most early ended activity, after eliminating the current activity, all left resources can be used by as much other activities.*

*Change the words, since the activity has been sorted according to the end time, so Greedy Selection is to choose the Activity a1. Select the activity with the earliest end time is not the only solution of Greedy Algorithm.*

***Description:***

The Activity Selection Problem has the Best Sub-Structure. *Sk = { ai belongs to S: si <= fk } defines all activities that starts before the end of activity ak. After select the Best Solution a1, then the solution of the original problem is to solve S1, find the Best Solution for all activities that starts before the end of activity ak.*

The Best Sub-Structure tells us that if a1 is among the Best Solution, then the original solution consists of activity a1 and Sub-Question S1.

***Prove:***

The process of prove can be eliminated, since we can see that although we can use Dynamic Programming to solve the Activity Selection Problem. But actually, *through choose the earliest activity recursively, and keep other compatible activities, we can get one Best Solution for such problem.*

Repeat this process, until we do not need any remaining activity. Here, we only need to proceed all activities according to the earliest finishing time, each activity can only be estimated once.

***Key:***

In Activity Selection Problem, we normally solve the problem from Top to Bottom, select one activity and put into the Best Solution, and solve for all remaining Sub-Problems.

Greedy Algorithm is normally designed from Top to Bottom: *we make an selection, then solve the remaining Sub-Problem, but we do not solve the problem from Bottom to Top, and make the selection.*

***Recursion Greedy Algorithm***

***Explanation:***

* *Array m represents start time array of all activities.*
* *Array f represents end time array of all activities.*
* *Variable k is used to specify Sub - Problem Sk.*
* *Variable n is used issue the Problem Size.*

Array s and f are sorted array based on f. *For integrity of all array, we add an extra virtual activity a0, it defines the start time equals to 0 and the finish time equals to 0.* After that we call *RESURSIVE-ACTIVITY-SELECTOR(s, f, 0, n)* to solve the original problem.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| s | 0 | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
| f | 0 | 4 | 5 | 6 | 7 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

*Recursive\_Activity\_Selector(s, f, k, n):*

*Supplement:*

* The variable m is used to divide array into two separate array.
* For the first part array, it can be represented by k...m.
* The second part array, it can be represented by m...n.
* In array [k...m], find out one Biggest Compatible Collection. In array [k...m], find out one Biggest Compatible Collection that all activities start before the end of ak and end before the start of am.
* Under the same situation, in array [m...n], find out one Biggest Compatible Collection that all activities start before the end of am and end before the start of an.
* Combine these two parts of Biggest Compatible Collection into final Collection with am.

*Pseudo - Code:*

*m = k + 1;*

*while ( m <= n && s[m] < f[k] )*

*{*

*m ++;*

*}*

*If ( m <= n ) {*

*Return {am} with Recursive\_Activity\_Selector(s, f, m, n);*

*}*

*Analysis:*

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| s | 0 | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
| f | 0 | 4 | 5 | 6 | 7 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

1. *Recursive\_Activity\_Selector(s, f, 0, 11):*

*k = 0; n = 11;*

*m = 1; s[0] = 0, f[0] = 0; s[1] = 1, f[1] = 4;*

*s[1] > f[0] && 1 <= 11;*

*{a1} U Recursive\_Activity\_Selector(s, f, 1, 11);*

1. *Recursive\_Activity\_Selector(s, f, 1, 11):*

*k = 1, n = 11;*

*m = 4; s[1] = 1, f[1] = 4; s[4] = 5, f[4] = 7;*

*s[4] > f[1] && 4 <= 11;*

*{a1} U {a4} U Recursive\_Activity\_Selector(s, f, 4, 11);*

1. *Recursive\_Activity\_Selector(s, f, 4, 11):*

*k = 4, n = 11;*

*m = 8; s[4] = 5, f[4] = 7; s[8] = 8, f[8] = 11;*

*s[8] > f[4] && 8 <= 11;*

*{a1} U {a4} U {a8} U Recursive\_Activity\_Selector(s, f, 8, 11);*

1. *Recursive\_Activity\_Selector(s, f, 8, 11):*

*k = 8, n = 11;*

*m = 11; s[8] = 8, f[8] = 11; s[11] = 12, f[11] = 16;*

*s[11] > f[8] && 8 <= 11;*

*{a1} U {a4} U {a8} U {a11};*

***Iteration Greedy Algorithm***

*Greedy\_Activity\_Selector(s, f):*

*Instruction:*

*Array s and f are Sorted according to array f, therefore, the first element in array f would be with the least finishing time and the first element in array s would also be with the least finishing time.*

*Pseudo - Code:*

*n = f.Length();*

*k = 1;*

*A = { a1 };*

*For ( m = 2; m <= n; m++ )*

*{*

*If ( s[ m ] >= f[ k ] )*

*{*

*A = A U { am };*

*k = m;*

*}*

*}*

*Return A;*