*Dynamic Programming Principle*

*Background:*

Although we had focused on solving two problems by using Dynamic Programming, but we still can not figure out when to use Dynamic Programming.

This Chapter would discuss two basic Elements when using Dynamic Programming to solve the Best Optimization Problem: *Best Sub - Structure* and *Overlapping Sub - Question*. *Also we will continue to discuss the Memo Method, and go deeper into How to use the overlapping feature best with the help of Memo Method.*

*Best Sub - Structure:*

The first step of solving the Best Optimization Problem by using Dynamic Programming it to describe the Structure by Best Solution. Two Problems all have the Best Sub-Structure. *(Matrix Chain Multiplication and Dynamic Programming Introduction)*

*One Normal Principle when try to find the Best Sub-Structures:*

1. To Solve the Best Sub-Structure is to make one choice, to choose the first Cutting Location for Steel, and choose the Division Location for the Matrix Chain Multiplication. The result of this kind of Selection would bring one or more Problems which need to be solved.
2. For the Given Problem, in the possible first step, you already know which choice can get the Best Solution. But now do not need to care how to get the Best Solution, just assume that we already get this kind of solution.
3. Given the specific Selection which can get Best - Solution, then you will make sure how many solutions must be generated, and how to describe the Sub - Problem Space.
4. Using ‘Cut - and - Paste’ technology to prove that the Solution of the Part of Original Solution, each solution of the problem is itself’s solution.

Th Good Experience to describe the Sub - Problem is to try to keep the Sub - Problem Space as simpler as it could be, only needs to enlarge it when necessary.

*For different problems in different ranges, the different of Best Sub - Structure has two different sides:*

1. *How many Sub - Problems have been involved in the Original Problem.*
2. *How many choices need to be considered when we make sure one Sub - Problem.*

*First Question:*

* Steel Cutting with the length of n, make sure how many Sub - Problems in the Original Problem.
* Figure out how many Best Choices in the n - j Steel Cutting Question.

*Second Question:*

* Matrix Chain Multiplication Ai\*Ai+1...Aj-1\*Aj, there have two Sub - Problems, we need to figure out the Best Solution of Ai\*Ai+1...Ak and Ak+1\*Ak+2...Aj, and both Sub - Problems need to solve the Best Solution.
* Once figure out the Best Solution of Sub - Problem, then we can get the value of k among j - i choices.

*Cost Analysis:*

For these two situations, we need to multiple the total number of Sub - Question with the total choice number of Sub - Question to calculate total run - time Cost of Dynamic Programming.

* For Steel Cutting Problem, we have O(n) Sub - Problem, and each Sub - Problem can have n choices, therefore the total Run-time Cost equals to O(n \* n).
* For Matrix Chain Multiplication Problem, we have O(n \* n) Sub - Problems, and for each Sub - Problem, we just need to consider the n - 1 choices, therefore the Runtime Cost equals to O(n \* n \* n).

*Sub - Question Img Analyze:*

*Analyze the Sub - Question in the same way. Each point corresponds to one Sub - Question, but need to consider the corresponding edge that connects with two Vertex.*

* Remember that, there have n vertexes in the Sub - Problem of the Steel Cutting Problem, also there have n edge for each Vertex. So the total Run-Time Cost equals to O(n \* n).
* For Matrix Chain Multiplication Problem, we have O(n \* n) vertexes and there have n - 1 edge for each Vertex. So the total Run-Time Cost equals to O(n \* n \* n).

*Steps:*

Normally, when we solve the Dynamic Programming Problem, we need to solve the problem from *Bottom to Top*, *which is to say we need to get the Best Solution of Sub-Problem and in further step to get the Best Solution of original Problem.*

*Difference Between Dynamic Programming and Greedy Algorithm:*

Dynamic Programming Algorithm has much common with Greedy Algorithm, especially that the solution to solve Greedy Algorithm Problem can also be used to solve the Dynamic Programming Problem.

We are not choose the Best Solution from the Sub - Question and then choose among it, but make the Greedy Choice at first - seen from the partial, it is the Best Solution - then we need to solve the Sub - Problem, but we do not spend time to solve all possible Sub - Questions.

*Tricky Points:*

We need to figure out whether the problem has the Best Sub - Structure when trying to use Dynamic Programming Algorithm. Normally, *when solving two problems and they would not share the same resource, then we can call two problems independent.*

*Overlapping Sub - Solution:*

The second property by using the Dynamic Programming Method to solve the Best Optimization Question is to ensure that the Sub - Question Space should be ‘*small*’ enough, so that Recursive Algorithm can solve Sub - Question repeatedly, but do not generate the new Sub-Question.

* If we solve Sub - Question by using Recursion Algorithm repeatedly, then we call the Optimization Problem has the Overlapping Sub - problems.
* While the method that is appropriate to Divide and Conquer Algorithm is to generate the fully new Sub - Problem in each recursive steps.

*Dynamic Programming Algorithm normally uses the Overlapping Property: solve sub-problem one time, save the solution into the table, and when need the sub - table again, then query the table directly, so that the cost that query this table each time is constant.*

Below is the table m[1...n][1...n]:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 0 | Round 1 | Round 2 | Round 3 | Round 4 | Round 5 |
| 2 |  | 0 | Round 1 | Round 2 | Round 3 | Round 4 |
| 3 |  |  | 0 | Round 1 | Round 2 | Round 3 |
| 4 |  |  |  | 0 | Round 1 | Round 2 |
| 5 |  |  |  |  | 0 | Round 1 |
| 6 |  |  |  |  |  | 0 |

We need to calculate element m[3][4] for 4 times, from m[2][4], m[3][5], m[3][6] to m[1][4]. So, if we calculate m[3][4] repeatedly, then it would cost a lot and the time would rise a lot.

*Pseudo - Code:*

*Recursive\_Matrix\_Chain(p, i, j)*

*If ( i == j )*

*Return 0;*

*m[i][j] = +8;*

*For ( k = i to j - 1 )*

*{*

*q = Recursive\_Matrix\_Chain(p, i, k) +*

*Recursive\_Matrix\_Chain(p, k + 1, j) +*

*p(i - 1) \* pk \* pj;*

*If ( q < m[i][j] )*

*m[i][j] = q;*

*}*

*Return m[i][j];*

*Conclusion:*

Compare the Recursion Algorithm without memo with the Bottom-to-Top Algorithm, the latter is far more efficient, since it utilize the Overlapping Sub - Question. The Recursion Algorithm solve the sub - question repeatedly as long as it meets the question. The memo can improve the efficient largely.

*Re - Construction of Best Solution:*

Consider from reality, normally we need to save the specific selection for each sub-question into one table, then we do not need to reconstruct these information according to cost information.

*Memo:*

However, we can keep solving the problem from Top - to - Bottom Algorithm which has the same efficiency with the from Bottom - to - Top Algorithm. The thinking pattern here is to add Memo. We can maintain the table and keep the solution into the table, but also keep control flow.

Recursion Algorithm with Memo help maintain each element to save the solution. The initial value of the table has been setup as the special value which means its initialization value has not been updated. When meet the Sub - Question first time, we need to calculate the value and keep it in the table. After that, if we meet the same Sub - Question, then we just query the table and return its value.

*Pseudo - Code:*

*Memoized\_Matrix\_Chain(p):*

*int n = p.length() - 1;*

*Let m[1...n][1...n] to be a new array.*

*For ( i = 1; i < = n; i ++ )*

*{*

*For ( j = i; j < = n; j ++)*

*{*

*m[i][j] = -8;*

*}*

*}*

*Return Lookup\_Chain(m, p, 1, n);*

*Lookup\_Chain(m, p, i, j):*

*If ( m[i][j] < -8 )*

*Return m[i][j];*

*If ( i == j )*

*m[i][i] = 0;*

*Else*

*{*

*For ( k = i to j - 1 )*

*{*

*q = Recursive\_Matrix\_Chain(p, i, k) +*

*Recursive\_Matrix\_Chain(p, k + 1, j) +*

*p(i - 1) \* pk \* pj;*

*If ( q < m[i][j] )*

*m[i][j] = q;*

*}*

*}*

*Return m[i][j];*