UNIT IV ALGORITHM DESIGN TECHNIQUES

Dynamic Programming: Matrix-Chain Multiplication – Elements of Dynamic Programming – Longest Common Subsequence- Greedy Algorithms: – Elements of the Greedy Strategy- An Activity-Selection Problem - Huffman Coding.

DYNAMIC PROGRAMMING

Dynamic Programming is the most powerful design technique for solving optimization problems. Divide & Conquer algorithm partition the problem into disjoint sub problems, solve the subproblems recursively and then combine their solution to solve the original problems.

Dynamic Programming is used when the subproblems are not independent, e.g. when they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.

Dynamic Programming solves each subproblem just once and stores the result in a table so that it can be repeatedly retrieved if needed again.

Dynamic Programming is a Bottom-up approach- we solve all possible small problems and then combine to obtain solutions for bigger problems.

Dynamic Programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of achieving sub-problem solutions and appearing to the "principle of optimality".

Characteristics of Dynamic Programming:

Dynamic Programming works when a problem has the following features:-

o **Optimal Substructure:** If an optimal solution contains optimal sub solutions then a problem exhibits optimal substructure.

o **Overlapping subproblems:** When a recursive algorithm would visit the same subproblems repeatedly, then a problem has overlapping subproblems.

If a problem has optimal substructure, then we can recursively define an optimal solution. If a problem has overlapping subproblems, then we can improve on a recursive implementation by computing each subproblem only once.

If a problem doesn't have optimal substructure, there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have overlapping subproblems, we don't have anything to gain by using dynamic programming.

If the space of subproblems is enough (i.e. polynomial in the size of the input), dynamic programming can be much more efficient than recursion.

Elements of Dynamic Programming

There are basically three elements that characterize a dynamic programming algorithm:-

1. **Substructure**: Decompose the given problem into smaller subproblems. Express the solution of the original problem in terms of the solution for smaller problems.

2. **Table Structure**: After solving the sub-problems, store the results of the sub problems in a table. This is done because subproblem solutions are reused many times, and we do not want to repeatedly solve the same problem over and over again.

3. **Bottom-up Computation**: Using tables, combine the solution of smaller subproblems to solve larger subproblems and eventually arrive at a solution to complete the problem.

Bottom-up means:-

1. Start with the smallest subproblems.
2. Combining their solutions obtains the solution to sub-problems of increasing size.
3. Until solving the original problem.

Components of Dynamic programming

1. Stages: The problem can be divided into several subproblems, which are called stages. A stage is a small portion of a given problem. For example, in the shortest path problem, they were defined by the structure of the graph.

2. States: Each stage has several states associated with it. The state for the shortest path problem was the node reached.

3. Decision: At each stage, there can be multiple choices out of which one of the best decisions should be taken. The decision taken at every stage should be optimal; this is called a stage decision.

4. Optimal policy: It is a rule which determines the decision at each stage; a policy is called an optimal policy if it is globally optimal. This is known as the Bellman principle of optimality.

5. Given the current state, the optimal choices for each of the remaining states does not depend on the previous states or decisions. In the shortest path problem, it was not necessary to know how we got a node, only that we did.

6. There exists a recursive relationship that identifies the optimal decisions for stage j, given that stage j+1, has already been solved.

7. The final stage must be solved by itself.

Development of Dynamic Programming Algorithm

It can be broken into four steps:

1. Characterize the structure of an optimal solution.

2. Recursively defined the value of the optimal solution. Like Divide and Conquer, divide the problem into two or more optimal parts recursively. This helps to determine what the solution will look like.

3. Compute the value of the optimal solution from the bottom up (starting with the smallest subproblems)

4. Construct the optimal solution for the entire problem from the computed values of smaller subproblems.

Applications of dynamic programming

1. 0/1 knapsack problem
2. Mathematical optimization problem
3. All pair Shortest path problem
4. Reliability design problem
5. Longest common subsequence (LCS)
6. Flight control and robotics control
7. Time-sharing: It schedules the job to maximize CPU usage

Development of Dynamic Programming Algorithm

1. Characterize the structure of an optimal solution.
2. Define the value of an optimal solution recursively.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct the optimal solution from the computed information.

Dynamic Programming Approach

Let Ai,j be the result of multiplying matrices i through j. It can be seen that the dimension of Ai,j is pi-1 x pj matrix.

Dynamic Programming solution involves breaking up the problems into subproblems whose solution can be combined to solve the global problem.

At the greatest level of parenthesization, we multiply two matrices A1.....n=A1....k x Ak+1....n)

Thus we are left with two questions:

o How to split the sequence of matrices?

o How to parenthesize the subsequence A1.....k andAk+1.....n?

One possible answer to the first question for finding the best value of 'k' is to check all possible choices of 'k' and consider the best among them. But that it can be observed that checking all possibilities will lead to an exponential number of total possibilities. It

can also be noticed that there exists only O(n2) different sequence of matrices, in this way do not reach the exponential growth.

Step1: Structure of an optimal parenthesization: Our first step in the dynamic paradigm is to find the optimal substructure and then use it to construct an optimal solution to the problem from an optimal solution to subproblems.

Let Ai....j where i≤ j denotes the matrix that results from evaluating the product Ai Ai+1....Aj.

If i < j then any parenthesization of the product Ai Ai+1 .....Aj must split that the product between Ak and Ak+1 for some integer k in the range i ≤ k ≤ j. That is for some value of k, we first compute the matrices Ai.....k & Ak+1....j and then multiply them together to produce the final product Ai....j. The cost of computing Ai....k plus the cost of computing Ak+1....j plus the cost of multiplying them together is the cost of parenthesization.

Step 2: A Recursive Solution: Let m [i, j] be the minimum number of scalar multiplication needed to compute the matrixAi....j.

If i=j the chain consist of just one matrix Ai....i=Ai so no scalar multiplication are necessary to compute the product. Thus m [i, j] = 0 for i= 1, 2, 3....n.

If i<j we assume that to optimally parenthesize the product we split it between Ak and Ak+1 where i≤ k ≤j. Then m [i,j] equals the minimum cost for computing the subproducts Ai....k and Ak+1....j+ cost of multiplying them together. We know Ai has dimension pi-1 x pi, so computing the product Ai....k and Ak+1....jtakes pi-1 pk pj scalar multiplication, we obtain

m [i,j] = m [i, k] + m [k + 1, j] + pi-1 pk pj