UNIT – IV

DYNAMIC PROGRAMMING

Introduction of Dynamic Programming

- The most powerful design technique for solving optimization problems.
- ***Dynamic Programming*** is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping subinstances.
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems
- "Programming" here means "planning"

Introduction of Dynamic Programming

- Main idea:
  - Set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - Solve smaller instances once
  - Record solutions in a table
  - Extract solution to the initial instance from that table

Introduction of Dynamic Programming

- Divide and Conquer algorithm partition the problem into disjoint subproblems 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

- Dynamic Programming solves each subproblems 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:-
- **Optimal Substructure**: If an optimal solution contains optimal sub solutions then a problem exhibits optimal substructure.
- **Overlapping subproblems**: When a recursive algorithm would visit the same subproblems repeatedly, then a problem has overlapping subproblems.

Principle of Optimality

- ***Principle of optimality :*** *"In an optimal sequence of decisions or choices, each sub sequence must also be optimal".*
- The principle of optimality is the heart of dynamic programming. It states that to find the optimal solution of the original problem, a solution of each sub problem also must be optimal.
- It is not possible to derive optimal solution using dynamic programming if the problem does not possess the principle of optimality.

- 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:-
- **Substructure**: Decompose the given problem into smaller subproblems. Express the solution of the original problem in terms of the solution for smaller problems.
- **Table Structure**: After solving the sub-problems, store the results to the subproblems 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.
- **Bottom-up Computation**: Using table, combine the solution of smaller subproblems to solve larger subproblems and eventually arrives at a solution to complete problem.

- Note: Bottom-up means:-
- Start with smallest subproblems.
- Combining their solutions obtain the solution to sub-problems of increasing size.
- Until solving at the solution of the original problem.

Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.

Optimal Substructure

- Show that a solution to a problem consists of making a choice, which leaves one or more subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution.
- Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste.
- Need to ensure that a wide enough range of choices and subproblems are considered.

Optimal Substructure

- Optimal substructure varies across problem domains:
  - 1. *How many subproblems* are used in an optimal solution.
  - 2. *How many choices* in determining which subproblem(s) to use.
- Informally, running time depends on (# of subproblems overall) × (# of choices).
- How many subproblems and choices do the examples considered contain?
- Dynamic programming uses optimal substructure bottom up.
  - *First* find optimal solutions to subproblems.
  - *Then* choose which to use in optimal solution to the problem.

Overlapping Subproblems

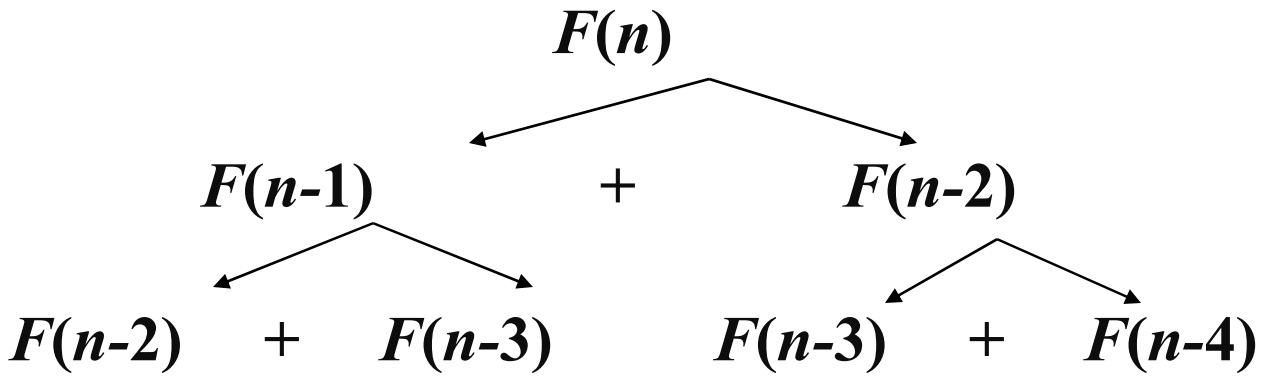
- The space of subproblems must be “small”.
- The total number of distinct subproblems is a polynomial in the input size.
  - A recursive algorithm is exponential because it solves the same problems repeatedly.
  - If divide-and-conquer is applicable, then each problem solved will be brand new.

Example: Fibonacci numbers

- **Recall definition of Fibonacci numbers:**

$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$
$$F(0) = 0$$
$$F(1) = 1$$

- **Computing the n^{th} Fibonacci number recursively (top-down):**



Example: Fibonacci numbers (cont.)

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$F(0) = 0$
$F(1) = 1$
$F(2) = 1+0 = 1$
…
$F(n\text{-}2) =$
$F(n\text{-}1) =$
$F(n) = F(n\text{-}1) + F(n\text{-}2)$

0	1	1	. . .	$F(n\text{-}2)$	$F(n\text{-}1)$	$F(n)$

Efficiency:
- time — n
- space — n

What if we solve it recursively?

Example: Fibonacci numbers (cont.)

- The Fibonacci numbers are the elements of the sequence

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . . ,

Algorithm fib(n)

if n = 0 or n = 1 return 1

return fib(n – 1) + fib(n – 2)

- The original problem F(n) is defined by F(n-1) and F(n-2)

Example: Fibonacci numbers (cont.)

- Notice that if we call, say, fib(5), we produce a call tree that calls the function on the same value many different times:
- fib(5)
- fib(4) + fib(3)
- (fib(3) + fib(2)) + (fib(2) + fib(1))
- ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
- (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

- If we try to use recurrence directly to compute the n^{th} Fibonacci number *F(n)* , we would have to recompute the same values of this function many times

Example: Fibonacci numbers (cont.)

- Certain algorithms compute the n^{th} Fibonacci number without computing all the preceding elements of this sequence.
- It is typical of an algorithm based on the classic bottom-up dynamic programming approach,
- A top-down variation of it exploits so-called memory functions
- The crucial step in designing such an algorithm remains the same => Deriving a recurrence relating a solution to the problem's instance with solutions of its smaller (and overlapping) subinstances.

Dynamic programming

- Dynamic programming usually takes one of two approaches:
- Bottom-up approach: All subproblems that might be needed are solved in advance and then used to build up solutions to larger problems. This approach is slightly better in stack space and number of function calls, but it is sometimes not intuitive to figure out all the subproblems needed for solving the given problem.

- Top-down approach: The problem is broken into subproblems, and these subproblems are solved and the solutions remembered, in case they need to be solved again. This is recursion and Memory Function combined together.

Bottom Up

- In the bottom-up approach we calculate the smaller values of Fibo first, then build larger values from them. This method also uses linear (O(n)) time since it contains a loop that repeats n – 1 times.

```
Algorithm Fibo(n)

    a = 0, b = 1
    repeat n − 1 times
        c = a + b
        a = b
        b  = c
    return b
```

- In both these examples, we only calculate fib(2) one time, and then use it to calculate both fib(4) and fib(3), instead of computing it every time either of them is evaluated.

Divide & Conquer Method Vs Dynamic Programming.

Divide & Conquer Method	Dynamic Programming
1.It deals (involves) three steps at each level of recursion: **Divide** the problem into a number of subproblems. **Conquer** the subproblems by solving them recursively. **Combine** the solution to the subproblems into the solution for original subproblems.	**1.**It involves the sequence of four steps: •Characterize the structure of optimal solutions. •Recursively defines the values of optimal solutions. •Compute the value of optimal solutions in a Bottom-up minimum. •Construct an Optimal Solution from computed information.
2. It is Recursive.	**2.** It is non Recursive.
3. It does more work on subproblems and hence has more time consumption.	**3.** It solves subproblems only once and then stores in the table.
4. It is a top-down approach.	**4.** It is a Bottom-up approach.
5. In this subproblems are independent of each other.	**5.** In this subproblems are interdependent.
6. For example: Merge Sort & Binary Search etc.	**6. For example:** Matrix Multiplication.

Floyd-Warshall algorithm

- Dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph G=(V,E) which is efficient method if it is a dense graph.
- Can be applied for graph having negative weights on edges .
- Let the vertices of G be V = {1, 2.....n} and consider a subset {1 2.....k} of vertices for some k. For any pair of vertices i, j ∈ V, considered all paths from i to j whose intermediate vertices are all drawn from {1, 2.....k}, and let p be a minimum weight path from amongst them.

Floyd-Warshall algorithm

- If k is an intermediate vertex of path p, then we break p down into $i \rightarrow k \rightarrow j$.
- Let $d_{ij}^{(k)}$ be the weight of the shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2.....k}.
- A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

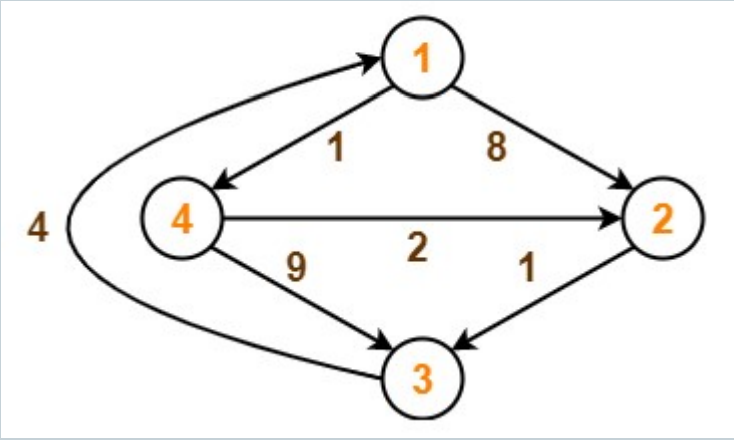
Floyd-Warshall algorithm

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

- Running time is $O(n^3)$.

Floyd-Warshall algorithm

- Consider the below graph. Apply Floyd Warshall's algorithm to find shortest path between all pair of nodes.



Floyd-Warshall algorithm

- When k = 0

$$D_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

- When k =1

$$D_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

Floyd-Warshall algorithm

- When k = 3

$D_3 =$

	1	2	3	4
1	0	8	9	1
2	5	0	1	6
3	4	12	0	5
4	7	2	3	0

- When k =4

$D_4 =$

	1	2	3	4
1	0	3	4	1
2	5	0	1	6
3	4	7	0	5
4	7	2	3	0