ANALYSIS AND DESIGN OF ALGORITHMS

Module 4

Dynamic Programming

Course code: 16BCA5D11

Semester: 5th semester

Unit 4: Dynamic Programming

The method

Dynamic Programming was invented by **Richard Bellman**, 1950. It is a very general technique for solving optimization problems. Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of sub-problems. Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.

Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems and optimal substructure**.

Overlapping Sub-Problems

Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.

For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

Optimal Sub-Structure

A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.

For example, the Shortest Path problem has the following optimal substructure property −

If a node **x** lies in the shortest path from a source node **u** to destination node **v**, then the shortest path from **u** to **v** is the combination of the shortest path from **u** to **x**, and the shortest path from **x** to **v**.

The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.

Steps of Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four steps −

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

Computing of Binomial Coefficient and Fibonacci Series

Computing of Binomial Coefficient

Computing binomial coefficients is non optimization problem but can be solved using dynamic programming.

Following are common definition of Binomial Coefficients.

1. A binomial coefficient C(n, k) can be defined as the coefficient of X^k in the expansion of (1 + X)^n.
2. A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects; more formally, the number of k-element subsets (or k-combinations) of an n-element set.

Binomial coefficients are represented by $C(n, k)$ or $\binom{n}{k}$ and can be used to represent the coefficients of a binomail:

$$(a + b)^n = C(n, 0)a^n + ... + C(n, k)a^{n-k}b^k + ... + C(n, n)b^n$$

The recursive relation is defined by the prior power

$C(n, k) = C(n\text{-}1, k\text{-}1) + C(n\text{-}1, k)$ for $n > k > 0$
IC $C(n, 0) = C(n, n) = 1$

Dynamic algorithm constructs a *nxk* table, with the first column and diagonal filled out using the IC.

Construct the table:

		0	1	*k* 2	...	*k*-1	*k*
	0	1					
	1	1	1				
	2	1	2	1			
n	.						
	.						
	.						
	k	1					1
	.						
	.						
	.						
	n-1	1				$C(n\text{-}1, k\text{-}1)$	
	n	1					$C(n, k)$

The table is then filled out iteratively, row by row using the recursive relation.

Algorithm *Binomial*(*n, k*)
    for $i \leftarrow 0$ **to** *n* **do** // fill out the table row wise
        for $i = 0$ **to** min(*i, k*) **do**
            if *j*==0 or *j*==*i* **then** $C[i, j] \leftarrow 1$ // IC
            else $C[i, j] \leftarrow C[i\text{-}1, j\text{-}1] + C[i\text{-}1, j]$ // recursive relation
    return *C*[*n, k*]

The cost of the algorithm is filing out the table. Addition is the basic operation. Because $k \leq n$, the sum needs to be split into two parts because only the half the table needs to be filled out for $i < k$ and remaining part of the table is filled out across the entire row.

$A(n, k)$ = sum for upper triangle + sum for the lower rectangle
$= \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=1}^{n} \sum_{j=1}^{k} 1$
$= \sum_{i=1}^{k} (i-1) + \sum_{i=1}^{n} k$
$= (k-1)k/2 + k(n-k)\ \varepsilon\ \Theta(nk)$

Computing Fibonacci Series

We define Fibonacci numbers as follows:

$$F_1 = F_2 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

There are following two different ways to store the values so that these values can be reused:

a) Memorization (Top Down)
b) Tabulation (Bottom Up)

a) Memorization (Top Down): The memorized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need the solution to a sub-problem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

b) Tabulation (Bottom Up): The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3) and so on. So literally, we are building the solutions of sub-problems bottom-up.

Naive Recursive Algorithm

We first look into the naive recursive algorithm wherein we use recursion to calculate the previous two fibonacci numbers recursively and then use the result to calculate our answer. The algorithm is shown below:

```
int fib(n)
{ if(n==1 || n==2)
return 1;
return fib(n-1)+fib(n-2); }
```

This naive method uses exponential time as the time function looks like the following recurrence.

$$T(n) = T(n-1) + T(n-2) + O(1)$$

$$T(n) \geq 2 \times T(n-2)$$

$$T(n) = \Theta(2n/2)$$

Clearly we can see that this algorithm is highly inefficient and can not be used to solve the problems of Fibonacci Numbers efficiently. We now look into the idea of memorization and how it helps us in solving our problem of Fibonacci Numbers.

Memorization

We see that there are a lot of overlapping sub-problems. Ideally we should be solving an instance of a sub-problem only once. For all subsequent occurrences of the sub-problems, we can use the pre-computed result to solve further queries.

We make a memo table where each entry is initially empty. Whenever we get a query, we first check it's value in the memo table. If we already have the value present in the table then we pick up the value in the table and continue with our work. Else, we compute the value of the query, update it in the table and then continue with our work.

3.3 Naive Algorithm with Memorization

We present the new algorithm below which is a modification of the naive algorithm using the concept of Memorization.

```
int memo[] = empty
int fib(n)
```

```
{ if(memo[n] != empty)
return memo[n]; if(n==1 || n==2)
return 1;
memo[n] = fib(n-1)+fib(n-2);
return memo[n];
}
```

Doing this reduces our running time drastically. We observe that we only calculate a subproblem only once and for each subsequent calls, we only spend O(1) time. In a way, we only calculate a recursion tree once. Thus, the time required to solve a query is equal to the number of unique 2 sub problems we encounter. Since we only face O(n) distinct sub problems, our running time comes out to be O(n), which is a significant improvement from the earlier naive algorithm.

Another way to speed out the algorithm is to process the subproblems in such a way that there is no repetition of a subproblem. This is the bottom up approach.

Bottom-up approach

```
memo[] = empty
for k in range(n)
if(k<=2) f=1;
else f=memo[k-1]+memo[k-2];
memo[k]=f;
return f;
```

This method produces the same complexity as the previous memorization method, i.e. O(n).

All pairs shortest path-Floyd's algorithm

Given a directed, connected weighted graph G(V,E), for each edge ⟨u,v⟩∈E, a weight w(u,v) is associated with the edge. The **all pairs of shortest paths problem** (APSP) is to find a shortest path from u to v for every pair of vertices u and v in V.

Algorithms for the APSP problem

- Matrix Multiplication / Repeated Squaring
- The Floyd-Warshall Algorithm
- Transitive Closure of a Graph

Floyd–Warshall algorithm is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of shortest paths between all pairs of vertices.

The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.

The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.

The algorithm proceeds by allowing an additional intermediate vertex at each step. For each introduction of a new intermediate vertex x, the shortest path between any pair of vertices u and v, x,u,v∈V, is the minimum of the previous best estimate of δ(u,v), or the combination of the paths from u→x and x→v.

$$\delta(u,v) \leftarrow \min(\delta(u,v), \delta(u,x)+\delta(x,v))$$

Let the directed graph be represented by a weighted matrix W.

FLOYD - WARSHALL (W)

1. n ← rows [W].
2. D^0 ← W
3. for k ← 1 to n
4. do for i ← 1 to n
5. do for j ← 1 to n
6. do $d_{ij}^{(k)} \leftarrow \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
7. return $D^{(n)}$

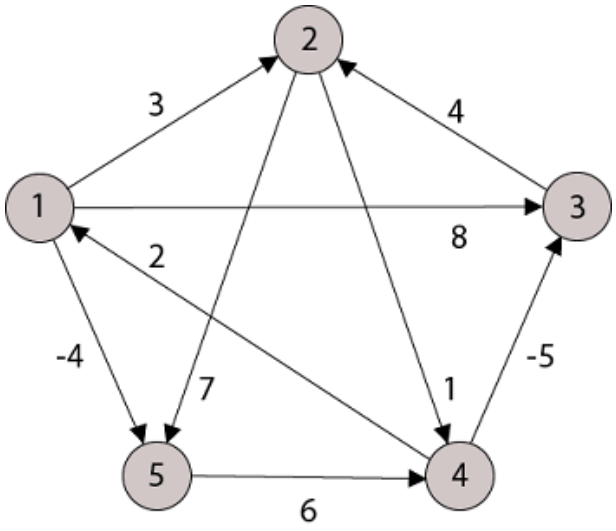
Let $d_{ij}^{(k)}$ be the weight of the shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2.....k}.

A recursive definition is given by

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

The strategy adopted by the Floyd-Warshall algorithm is **Dynamic Programming**. The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-6. Each execution of line 6 takes O (1) time. The algorithm thus runs in time $\theta(n^3)$.

Example: Apply Floyd-Warshall algorithm for constructing the shortest path. Show that matrices $D^{(k)}$ and $\pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph.



Solution:

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} \text{ if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} \text{ if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Step (i) When k = 0

$$D^{(0)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & \infty \\ 2 & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(0)} = \begin{matrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 3 & NIL \\ 4 & NIL & NIL & NIL & NIL \\ NIL & NIL & NIL & 5 & NIL \end{matrix}$$

Step (ii) When k =1

$$d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$d_{14}^{(1)} = \min (d_{14}^{(0)}, d_{11}^{(0)} + d_{14}^{(0)})$

$d_{14}^{(1)} = \min (\infty , 0 + \infty) = \infty$

$d_{35}^{(1)} = \min (d_{35}^{(0)}, d_{31}^{(0)} + d_{15}^{(0)})$

$d_{35}^{(1)} = \min (\infty, \infty + (-4)) = \infty$

$d_{15}^{(1)} = \min (d_{15}^{(0)}, d_{11}^{(0)} + d_{15}^{(0)})$

$d_{15}^{(1)} = \min (-4, 0 + -4) = -4$

$d_{42}^{(1)} = \min (d_{42}^{(0)}, d_{41}^{(0)} + d_{12}^{(0)})$

$d_{42}^{(1)} = \min (\infty, 2 + 3) = 5$

$d_{21}^{(1)} = \min (d_{21}^{(0)}, d_{21}^{(0)} + d_{11}^{(0)})$

$d_{21}^{(1)} = \min (\infty, \infty + 0) = \infty$

$d_{43}^{(1)} = \min (d_{43}^{(0)}, d_{41}^{(0)} + d_{13}^{(0)})$

$d_{43}^{(1)} = \min (\infty, 2 + 8) = 10$

$d_{23}^{(1)} = \min (d_{23}^{(0)}, d_{21}^{(0)} + d_{13}^{(0)})$

$d_{23}^{(1)} = \min ((\infty, \infty + 8) = \infty$

$d_{45}^{(1)} = \min (d_{45}^{(0)}, d_{41}^{(0)} + d_{15}^{(0)})$

$d_{45}^{(1)} = \min (\infty, 2 + (-4)) = -2$

$d_{31}^{(1)} = \min (d_{31}^{(0)}, d_{31}^{(0)} + d_{11}^{(0)})$

$d_{31}^{(1)} = \min (\infty, \infty + 0) = \infty$

$d_{51}^{(1)} = \min (d_{51}^{(0)}, d_{51}^{(0)} + d_{11}^{(0)})$

$d_{51}^{(1)} = \min (\infty, \infty + 0) = \infty$

$$D_{ij}^{(1)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & \infty \\ 2 & 5 & 10 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(1)} = \begin{matrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 3 & NIL \\ 4 & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{matrix}$$

Step (iii) When k = 2

$d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

$d_{14}^{(2)} = \min (d_{14}^{(1)}, d_{12}^{(1)} + d_{24}^{(1)})$

$d_{14}^{(2)} = \min (\infty, 3 + 1) = 4$

$d_{21}^{(2)} = \min (d_{21}^{(1)}, d_{22}^{(1)} + d_{21}^{(1)})$

$d_{21}^{(2)} = \min (\infty , 0 + \infty) = \infty$

$d_{34}^{(2)} = \min (d_{34}^{(1)}, d_{32}^{(1)} + d_{24}^{(1)})$

$d_{34}^{(2)} = \min (-5, 4 + 1) = -5$

$d_{35}^{(2)} = \min (d_{35}^{(1)}, d_{32}^{(1)} + d_{25}^{(1)})$

$d_{35}^{(2)} = \min (\infty, 4 + 7) = 11$

$d_{43}^{(2)} = \min (d_{43}^{(1)}, d_{42}^{(1)} + d_{23}^{(1)})$

$d_{43}^{(2)} = \min (10, 5 + \infty) = 10$

$D_{ij}^{(2)} =$

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	-5	11
2	5	10	0	-2
∞	∞	∞	6	0

$\pi^{(2)} =$

NIL	1	1	2	1
NIL	NIL	NIL	2	2
NIL	3	NIL	3	2
4	1	1	NIL	1
NIL	NIL	NIL	5	NIL

Step (iv) When k = 3

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$d_{14}^{(3)} = \min\left(d_{14}^{(2)}, d_{13}^{(2)} + d_{34}^{(2)}\right)$$

$$d_{14}^{(3)} = \min(4, 8 + (-5)) = 3$$

$$D_{ij}^{(3)} = \begin{matrix} 0 & 3 & 8 & 3 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & 11 \\ 2 & 5 & 10 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(3)} = \begin{matrix} NIL & 1 & 1 & 3 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 3 & 2 \\ 4 & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{matrix}$$

Step (v) When k = 4

$$d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$$d_{21}^{(4)} = \min (d_{21}^{(3)}, d_{24}^{(3)} + d_{41}^{(3)})$$

$$d_{21}^{(4)} = \min (\infty, 1 + 2) = 3$$

$$d_{23}^{(4)} = \min (d_{23}^{(3)}, d_{24}^{(3)} + d_{43}^{(3)})$$

$$d_{23}^{(4)} = \min (\infty, 1 + 10) = 11$$

$$d_{25}^{(4)} = \min (d_{25}^{(3)}, d_{24}^{(3)} + d_{45}^{(3)})$$

$$d_{25}^{(4)} = \min (7, 1 + (-2)) = -1$$

$$d_{51}^{(4)} = \min (d_{51}^{(3)}, d_{54}^{(3)} + d_{41}^{(3)})$$

$$d_{51}^{(4)} = \min (\infty, 6 + 2) = 8$$

$$d_{31}^{(4)} = \min (d_{31}^{(3)}, d_{34}^{(3)} + d_{41}^{(3)})$$

$$d_{31}^{(4)} = \min (\infty, -5 + 2) = -3$$

$$d_{52}^{(4)} = \min (d_{52}^{(3)}, d_{54}^{(3)} + d_{42}^{(3)})$$

$$d_{52}^{(4)} = \min (\infty, 6 + 5) = 11$$

$$d_{32}^{(4)} = \min (d_{32}^{(3)}, d_{34}^{(3)} + d_{42}^{(3)})$$

$$d_{32}^{(4)} = \min (4, -5 + 5) = 0$$

$$d_{53}^{(4)} = \min (d_{53}^{(3)}, d_{54}^{(3)} + d_{43}^{(3)})$$

$$d_{53}^{(4)} = \min (\infty, 6 + 10) = 16$$

$D_{ij}^{(4)} =$

0	3	8	3	-4
3	0	11	1	-1
-3	0	0	-5	-7
2	5	10	0	-2
8	11	16	6	0

$\pi^{(4)} =$

NIL	1	1	3	1
4	NIL	4	2	2
4	4	NIL	3	4
4	1	1	NIL	1
4	4	4	5	NIL

Step (vi) When k = 5

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$d_{25}^{(5)} = \min\left(d_{25}^{(4)}, d_{25}^{(4)} + d_{55}^{(3)}\right)$$

$$d_{25}^{(5)} = \min(-1, -1 + 0) = -1$$

$$d_{23}^{(5)} = \min\left(d_{23}^{(4)}, d_{25}^{(4)} + d_{53}^{(3)}\right)$$

$$d_{23}^{(5)} = \min(11, -1 + 16) = 11$$

$$d_{35}^{(5)} = \min\left(d_{35}^{(4)}, d_{35}^{(4)} + d_{55}^{(3)}\right)$$

$$d_{35}^{(5)} = \min(-7, -7 + 0) = -7$$

$$D_{ij}^{(5)} = \begin{matrix} 0 & 3 & 8 & 3 & -4 \\ 3 & 0 & 11 & 1 & -1 \\ -3 & 0 & 0 & -5 & -7 \\ 2 & 5 & 10 & 0 & -2 \\ 8 & 11 & 16 & 6 & 0 \end{matrix} \qquad \pi^{(5)} = \begin{matrix} NIL & 1 & 1 & 5 & 1 \\ 4 & NIL & 4 & 2 & 4 \\ 4 & 4 & NIL & 3 & 4 \\ 4 & 1 & 1 & NIL & 1 \\ 4 & 4 & 4 & 5 & NIL \end{matrix}$$