Unit-4
5

DYNAMIC PROGRAMMING

We know that divide-and-conquer technique is used to solve the problems that can be divided into independent subproblems. On the other hand, dynamic programming is one such strategy that can be used to solve the problems having dependent subproblems. That is, in case of some problems, their subproblems are shared and they can not be solved independently. In case of some other problems, even though we can solve subproblems independently, many of the calculations may repeat, thus increasing the time.

Consider a problem of finding n^{th} fibonacci number. The formula is given by-

$$F(n) = F(n-1) + F(n-2)$$

with initial conditions, $F(0)=0$ & $F(1)=1$.

Here, if we try to solve $F(n-1)$, that will contain a term $F(n-2)+F(n-3)$. So, $F(n)$ and its subproblem $F(n-1)$ are sharing another subproblem $F(n-2)$. Thus, calculating these repeated terms is simply a waste of time.

So, instead of solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which we can obtain a solution for the original problem.

For example, to compute n^{th} fibonacci number, we can use the initial conditions $F(0) = 0$ & $F(1) = 1$ first and generate consecutive fibonacci numbers.

Computing Binomial Coefficient

It is a best example of dynamic programming strategy. We know that, binomial coefficient nC_k is the number of combinations of k elements from n elements. Binomial coefficients can be obtained from the formula-

$$(a+b)^n = nC_0\, a^n + nC_1\, a^{n-1} b + \ldots$$
$$nC_i\, a^{n-i} b^i + \ldots + nC_n\, a^0 b^n$$

But, here, we will consider a recursive formula

$$nC_k = n-1C_{k-1} + n-1C_k \quad , n > k > 0$$

& $nC_0 = nC_n = 1$

It is obvious that the above relation is a recurrence relation and ^nC_k is computed in terms of the smaller overlapping problems of same type. So, dynamic programming strategy suggests to prepare a table with n+1 rows and k+1 columns as below and the entries of the table are made based on above equation.

	0	1	2	3 ..	k-i	.....	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
.	.						
k	1						1
.	.						
.	.						
n-1	1				$^{n-1}C_{k-i}$		$^{n-1}C_k$
n	1						^nC_k

Here, to get a particular cell value, we will add the entry at the same column and previous row and the entry at previous column and previous row.

NOTE: This triangular shape table is also

known as Pascal's triangle.

The algorithm is as below -

```
ALGORITHM Binomial(n, k)
// To compute binomial coefficient nCk by
// Dynamic programming strategy.
// Input: Non negative integers n and k
//        such that n >= k.
// Output: The value of nCk.
for i ← 0 to n do
    for j ← 0 to min(i, k) do
        if j = 0 or j = k
            C[i, j] ← 1
        else
            C[i,j] ← C[i-1, j-1] + C[i-1, j]
return C[n, k]
```

Analysis:

1. The complexity of the algorithm depends on n and k.
2. The basic operation is addition.

We can observe from the table that, for first $k+1$ rows, the entries will constitute a triangle. Afterwards, it will be of rectangle shape. So, the number of additions differ for first $k+1$ rows and for rest of $n-k$ rows. Moreover, for each values of i and j in the algorithm, we have exactly one addition. Thus, the time complexity is given by

$$C = \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n} \sum_{j=1}^{k} 1$$

$$= \sum_{i=1}^{k} (i-1-1+1) + \sum_{i=k+1}^{n} (k-1+1)$$

$$= \sum_{i=1}^{k} (i-1) + \sum_{i=k+1}^{n} k$$

$$= \sum_{i=1}^{k} i - \sum_{i=1}^{k} 1 + k \cdot \sum_{i=k+1}^{n} 1$$

$$= \frac{k(k+1)}{2} - k + k(n-k-1+1)$$

$$= \frac{(k-1)k}{2} + k(n-k)$$

$$= \frac{k}{2}\{k-1+2n-2k\} = \frac{k(2n-k-1)}{2}$$

$$\simeq nk$$

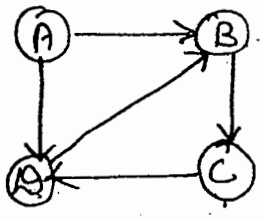
$\therefore C(n,k) \in \Theta(nk)$

Warshall's Algorithm

This algorithm is used for computing the transitive closure of a directed graph.

Def²: The transitive closure of a directed graph with n vertices can be defined as the n x n boolean matrix $T = \{t_{ij}\}$ in which the element in the i^{th} row and j^{th} column is 1 if there exists a non-trivial directed path (i.e. a directed path of positive length) from i^{th} vertex to the j^{th} vertex, otherwise t_{ij} is 0.

For ex- consider ~~di~~digraph & its transitive closure



$T =$

	A	B	C	D
A	0	1	1	1
B	0	1	1	1
C	0	1	1	1
D	0	1	1	1

We can obtain the transitive closure by traversing the given digraph by either DFS or BFS. If we start any of these traversal at the i^{th} vertex, the vertices those can be reached from i can be found. So, by traversing the graph for all the vertices, we will get the

transitive closure. But, the problem here is if the graph has n vertices, it must be traversed n times.

To overcome this problem, Warshall's algorithm is used, which will constructs the transitive closure of a given digraph with n vertices through a series of $n \times n$ boolean matrices $R^{(0)}, R^{(1)}, \ldots, R^{(k)} \ldots R^{(n)}$.

Each of these matrices will give some information on digraphs. The method for creating these matrices is given below-

(i) $R^{(0)}$ is nothing but an adjacency matrix of the graph.

(ii) If an element r_{ij} is 1 in $R^{(k-1)}$, then it remains 1 in $R^{(k)}$

(iii) If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if $r_{ik} = r_{kj} = 1$ in $R^{(k-1)}$

For illustration, consider the above graph. The series of matrices are given as below-

i.e.

$$R^{(k-1)} = \begin{array}{c} \\ k \\ i \end{array} \begin{array}{c} \begin{array}{cc} j & k \end{array} \\ \begin{bmatrix} & \vdots & \\ \cdots & 1 & \cdots \\ 0 & \cdots & 1 \end{bmatrix} \end{array} \Rightarrow R^{(k)} = \begin{array}{c} \\ k \\ i \end{array} \begin{array}{c} \begin{array}{cc} j & k \end{array} \\ \begin{bmatrix} & \\ 1 & \\ 1 & 1 \end{bmatrix} \end{array}$$

For illustration, consider the graph given above. The series of matrices are shown below-

$R^{(0)}$ =

	A	B	C	D
A	0	1	0	1
B	0	0	1	0
C	0	0	0	1
D	0	1	0	0

This is nothing but the adjacency matrix of the given graph.

$R^{(1)}$ =

	A	B	C	D
A	0	1	0	1
B	0	0	1	0
C	0	0	0	1
D	0	1	0	0

NO new '1' is introduced here because, in $R^{(0)}$, $k=1$ & 1^{st} column has only zeros.

$R^{(2)}$ =

	A	B	C	D
A	0	1	1	1
B	0	0	1	0
C	0	0	0	1
D	0	1	1	0

$r_{13}^{(2)}$ and $r_{43}^{(2)}$ are new '1's here.

$R^{(3)}$ =

	A	B	C	D
A	0	1	1	1
B	0	0	1	1
C	0	0	0	1
D	0	1	1	1

$r_{24}^{(3)}$ & $r_{44}^{(3)}$ are newly introduced 1's.

$R^{(4)}$ =

	A	B	C	D
A	0	1	1	1
B	0	1	1	1
C	0	1	1	1
D	0	1	1	1

$r_{22}^{(4)}$, $r_{32}^{(4)}$, $r_{33}^{(4)}$ are new 1's. This $R^{(4)}$ is transitive closure of given graph.

ALGORITHM Warshall(A[1..n][1..n])
// To implement Warshall's algorithm for
// computing the transitive closure.
// Input: The adjacency matrix A of a digraph
// containing n vertices.
// Output: The transitive closure of the digraph.
$R^{(0)} \leftarrow A$
for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do
      $R^{(k)}[i,j] \leftarrow R^{(k-1)}[i,j]$ or $R^{(k-1)}[i,k]$ and $R^{(k-1)}[k,j]$
return $R^{(n)}$

Analysis: The basic operation is assignment based on 'or' and 'and' conditions. For each value of i, j and k, the operation is performed once. So,

$$C(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} 1$$

$$= n^3$$

$$\therefore C(n) \in \Theta(n^3)$$

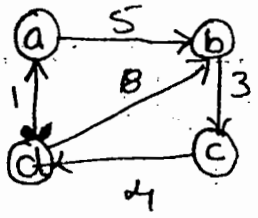
NOTE: The idea behind Warshall's algorithm has been generalized to get the shortest-path between the vertices of a graph. This can be seen in Floyd's algorithm.

Floyd's Algorithm

Suppose that a weighted connected graph, that may be directed or undirected, is given. Finding the distances i.e the length of shortest paths from each vertex to every other vertex of the graph is known as <u>all-pairs shortest-paths problem</u>. The distances from various vertices to other vertices are put in an $n \times n$ matrix format called the distance matrix D. For the weight (cost) matrix,

$$w_{ij} = \begin{cases} 0, & \text{if } i = j \\ \infty, & \text{if there is no edge between } i \text{ \& } j \\ x, & \text{a positive value, if the distance between } i \text{ \& } j \text{ is } x. \end{cases}$$

Consider a weighted digraph -



$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 5 & \infty & \infty \\ \infty & 0 & 3 & \infty \\ \infty & \infty & 0 & 4 \\ 1 & 8 & \infty & 0 \end{bmatrix} \end{array}$$

The distance matrix,

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 5 & 8 & 12 \\ 8 & 0 & 3 & 7 \\ 5 & 10 & 0 & 4 \\ 1 & 6 & 9 & 0 \end{bmatrix} \end{array}$$

If the graph do not have a cycle of negative length, then the distance matrix can be generated by Floyd's algorithm. It suggests to generate a series of matrices $D^{(0)}, D^{(1)}, \ldots, D^{(k)}, \ldots D^{(n)}$ for a graph of n vertices based on the following rules-

$d_{ij}^{(0)} = w_{ij}$ and

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

Following is the illustration for generating distance matrix by Floyd's algorithm for the above graph.

$$D^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 5 & \infty & \infty \\ \infty & 0 & 3 & \infty \\ \infty & \infty & 0 & 1 \\ 1 & 8 & \infty & 0 \end{bmatrix} \end{array}$$

This is nothing but weight (cost) matrix.

$$D^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 5 & \infty & \infty \\ \infty & 0 & 3 & \infty \\ \infty & \infty & 0 & 1 \\ 1 & 6 & \infty & 0 \end{bmatrix} \end{array}$$

~~Here $d_{12} = \min\{8, 6$~~

Here,

$d_{12} = \min\{8, 1+5\}$

$= 6$

$D^{(2)} =$

	a	b	c	d
a	0	5	8	∞
b	∞	0	3	∞
c	∞	∞	0	4
d	1	6	9	0

$d_{13} = \min\{\infty, 8+3\}$
$= 8$

$d_{43} = \min\{\infty, 6+3\}$
$= 9$

$D^{(3)} =$

	a	b	c	d
a	0	5	8	12
b	∞	0	3	7
c	∞	∞	0	4
d	1	6	9	0

$d_{24} = \min\{\infty, 3+4\}$
$= 7$

$d_{14} = \min\{\infty, 8+4\}$
$= 12$

$D^{(4)} =$

	a	b	c	d
a	0	5	8	12
b	8	0	3	7
c	5	10	0	4
d	1	6	9	0

$d_{21} = \min\{\infty, 1+7\}$
$= 8$

$d_{31} = \min\{\infty, 1+4\}$
$= 5$

$d_{32} = \min\{\infty, 6+4\}$
$= 10$

Here, $D^{(4)}$ is the require distance matrix.

```
ALGORITHM Floyd(W[1..n][1..n])
// Floyd's algo. for all-pair shortest-path problem
// Input: Weight matrix W of graph
// Output: Distance matrix D.
D ← W
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i,j] ← min{D[i,j], D[i,k] + D[k,j]}
return D
```

The efficiency of the algorithm is -

$$C(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} 1$$

$$= n^3$$

$\therefore C(n) \in \Theta(n^3)$.

Knapsack Problem :-

Consider a knapsack problem of finding the most valuable subset of n items of weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ that fit into a knapsack of capacity W. The dynamic programming strategy for solving this problem requires to derive a recurrence relation that expresses a solution to an instance of the knapsack problem interms of solutions to its smaller subinstances.

Consider an instance of a problem with first i items having weights $w_1, \ldots, w_i$ and values $v_1, \ldots, v_i$, and the knapsack of capacity j. Here, $1 \le i \le n$, $1 \le j \le W$. Let $V[i, j]$ be the most optimal solution for this instance. Now, the subsets of first i items that fit into knapsack of capacity j can be divided into two

categories viz. those that do not contain i^{th} item and those they do contain i^{th} item. Then, we have the following –

* Among the subsets that do not have i^{th} item, the value of optimal solution is $V[i-1, j]$
* If the subsets include i^{th} item, then obviously $j - w_i \geq 0$. The optimal solution will be $v_i + V[i-1, j-w_i]$.

Thus, we have –

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1, j-w_i]\}, & \text{if } j-w_i \geq 0 \\ V[i-1,j], & \text{if } j-w_i < 0 \end{cases}$$

with the initial conditions –

$V[0, j] = 0 \quad \forall j \geq 0$

$V[i, 0] = 0 \quad \forall i \geq 0$

Our requirement is to find $V[n, W]$ based on the above relation.

Ex: Consider the following problem with three items and the knapsack of capacity, $W = 4$. The weights and values as as –

Item	Weight	Value
A	3	25
B	1	20
C	2	40

Solution: Here, $W = 4$

$w_1 = 3, \quad w_2 = 1, \quad w_3 = 2$

$v_1 = 25, \quad v_2 = 20, \quad v_3 = 40$

As we know, $V[0, j] = 0 \quad \forall j \geq 0$

& $V[i, 0] = 0 \quad \forall i \geq 0$

Since there are only three items, the possibility for item inclusion may be one of 0, 1, 2, 3. Also, the capacity is 4. So, the possibility for one instance may be 0, 1, 2, 3 or 4. Thus, we have following table. The table entries are calculated as shown below.

i \ j	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	25	25
2	0	20	20	25	45
3	0	20	40	60	60

When $i = 1$:—

$V[1, 1] = V[i-1, j] = V[0, 1] = 0$ $\quad \because j - w_i = 1 - 3 < 0$

$V[1, 2] = V[i-1, j] = V[0, 2] = 0$ $\quad \because j - w_i = -1 < 0$

$V[1, 3] = \max\{V[i-1, j], v_1 + V[i-1, j-w_i]\}$

$= \max\{V[0, 3], 25 + V[0, 0]\}$

$= \max\{0, 25 + 0\} = 25$

$V[1, 4] = \max\{V[0, 4], 25 + V[0, 1]\}$

$= 25$

When $i = 2$:—

$V[2, 1] = \max\{V[1, 1], 20 + V[1, 0]\} = 20$

$V[2, 2] = \max\{V[1, 2], 20 + V[1, 1]\} = 20$

$V[2, 3] = \max\{V[1, 3], 20 + V[1, 2]\} = 25$

$V[2, 4] = \max\{V[1, 4], 20 + V[1, 3]\} = 45$

When $i = 3$:—

$V[3, 1] = V[2, 1] = 20$ $\quad \because j - w_i < 0$

$V[3, 2] = \max\{V[2, 2], 40 + V[2, 0]\} = 40$

$V[3, 3] = \max\{V[2, 3], 40 + V[2, 1]\} = 60$

$V[3, 4] = \max\{V[2, 4], 40 + V[2, 2]\} = 60$

After getting complete table, we have to look at the last entry. Because, we are interested in $V[n, W]$, where n is total number of items and W is total capacity.

Here, we have,

$V[n, W] = V[3, 4] = 60.$

So, the total profit we are going to get is 60. Now we have to check, what are the items added to get this profit. Note that,

$V[2, 4] \neq V[3, 4]$. $(\because V[2,4] = 45$ & $V[3,4] = 60)$

This indicates, by inserting item 3 into the knapsack, we are gaining something. Thus, the item 3 must be one of our choices.

Now, from the problem table, it is seen that item 3 has weight 2. So, if we insert it, the remaining capacity is, $W - w_3 = 4 - 2$

$= 2$ only.

So, to check about next item, we have to look at the solution table only up to 2nd column. And, the row will be 2nd obviously, as we have considered item 3 already.

Now, $V[2, 2] \neq V[1, 2]$ $(\because V[1,2] = 0, V[2,2] = 20)$

This implies, inserting item 2, we are going to gain profit. So, item 2 must be a part of solution.

Now, $W - w_2 - w_3 = 4 - 2 - 2$

$= 1$

But, $w_1 = 3$. So, we can't insert item 1.

Thus solution is $\{2, 3\}$ or $\{B, C\}$ with the profit 60.

Memory Functions

The basic idea behind the dynamic programming is to reduce the number of calculations involved in top-down approach for solving recurrence relations having overlapped subproblems. But for some problems, the dynamic programming strategy solves those subproblems that are not at all required for original problem. This is the disadvantage of bottom-up approach. So, to come out of these, we try to combine the good aspects of both top-down and bottom-up approaches. Such a method is based on memory functions.

In this method, we will solve the given problem in top-down approach only, but along with, we maintain the table structure as we do in usual bottom-up dynamic programming approach. Initially, all the entries of the table are initialized with a 'null' symbol to indicate that they have not yet been calculated. Then, whenever a new value has to be calculated, first the table is checked. If it is found, it is used.

Otherwise, if there is a 'null' value, it is calculated and stored at that position.

Thus, in this method, we calculate only those values which are required for the given problem.

~~Note that~~

Let us implement this method for knapsack problem. Here also, recursive relation remains same as before. i.e.

$$V[0,j] = 0 = V[i,0], \quad \forall i, j \geq 0$$

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1, j-w_i]\}, & j - w_i \geq 0 \\ V[i-1,j], & \text{if } j - w_i < 0 \end{cases}$$

The algorithm for knapsack problem using memory function technique is given below-

```
ALGORITHM MFKnapsack(i, j)
// To implement memory function method for
// the knapsack problem.
// Input: i, indicating the number of items
//        j, indicating capacity, i, j ≥ 0
// Output: Value of optimal feasible subset of
//         first i items.
// NOTE: Use of global variables Wt[1..n],
//       Val[1..n] and V[0..n, 0..W] is made.
//       V[0..n, 0..W] is initialized with -1's
//       except row 0 & column 0.
```

```
if V[i, j] < 0
    if j < Wt[i]
        val ← MFknapsack(i-1, j)
    else
        val ← max(MFknapsack(i-1, j),
                  Val[i] + MFknapsack(i-1,
                                      j-Wt[i]))
    V[i, j] ← val
return V[i, j]
```

NOTE: To workout an example is left out-as an exercise for students!