UNIT 2 DYNAMIC PROGRAMMING TECHNIQUE

Structure **Page No**

2.0 INTRODUCTION

Dynamic programming is an optimization technique. Optimization problems are those which required either minimum result or maximum result. Means finding the optimal solution for any given problem. Optimal means either finding the maximum answer or minimum answer. For example if we want to find the profit, will always find out the maximum profit and if we want to find out the cost will always find out the minimum cost. Although both greedy and dynamic are used to solve the optimization problem but there approach to finding the optimal solution is totally different. In greedy method we try to follow defined procedure to get the optimal result. Like Kruskal's for finding minimum cost spanning tree. Always select edge with minimum weight and that gives us best result or Dijkstra's for shortest path, always select the shortest path vertex and continue relaxing the vertices, so you get a shortest path. For any problem there may be a many solutions which are feasible, we try to find out the all feasible solutions and then pick up the best solution. Dynamic programming approach is different and time consuming compare to greedy method. Dynamic programming granted to find the optimal solution of any problem if their solution exist. Mostly dynamic programming problems are solved by using recursive formulas, though we will not use recursion of programming but formulas are recursive. It works on the concept of recursion i.e. dividing a bigger problem into similar type of sub problems and solving them recursively. The only difference is- DP does save the outputs of the smaller sub problem into a table and avoids the task of repetition in calculating the same sub problem. First it searches for the output of the sub problem in the table, if the output is not available then only it calculates the sub problem. This reduces the time complexity to a great extent. Due to its nature, DP is useful in solving the problems which can be divided into similar sub problems. If each sub problem is different in nature, DP does not help in reducing the time complexity.

So how the dynamic programming approach works.

- Breaks down the complex problem into simple sub problem.
- Find optimal solution to these sub problems.
- Store the results of sub problems.
- Re-use that result of sub problems so that same sub problem comes you don't need to calculate again.
- Finally calculates the results of complex problem.
- Storing the results of sub problems is called memorization.

2.1 OBJECTIVES

After going through the unit you will be able to:

- Define the Principle of Optimality
- Define all the stages of Dynamic Programming
- Solve Chained Matrix Multiplication, Optimal Binary Search Tree, All Pair Shortest Path Problem through Dynamic Programming Approach

2.2 PRINCIPAL OF OPTIMALITY

Dynamic programming follows the principal of optimality. If a problem have an optimal structure, then definitely it has principal of optimality. A problem has optimal sub structure if an optimal solution can be constructed efficiently from optimal solution of its sub-problems. Principal of optimality shows that a problem can be solved by taking a sequence of decision to solve the optimization problem. In dynamic programming in every stage we takes a decision. The Principle of Optimality states that components of a globally optimum solution must themselves be optimal.

A problem is said to satisfy the Principle of Optimality if the sub solutions of an optimal solution of the problem are themselves optimal solutions for their sub problems.
Examples:

- The shortest path problem satisfies the Principle of Optimality.
- This is because if $a, x_1, x_2, \dots, x_n$is a shortest path from node a to node b in a graph, then the portion of x_i to x_j on that path is a shortest path from x_i to x_j.
- The longest path problem, on the other hand, does not satisfy the Principle of Optimality. For example the undirected graph of nodes $a, b, c, d,$ and e and edges$(a, b), (b, c), (c, d), (d, e)$and (e, a). That is, G is a ring. The longest (noncyclic) path from a to d to a, b, c, d. The sub-path from b to c on that path is simply the edge b,c. But that is not the longest path from b to c. Rather b, a, e, d, c is the longest path. Thus, the sub path on a longest path is not necessarily a longest path.

Steps of Dynamic Programming:

Dynamic programming has involve four major steps:

1. Develop a mathematical notation that can express any solution and sub solution for the problem at hand.
2. Prove that the Principle of Optimality holds.

3. Develop a recurrence relation that relates a solution to its sub solutions, using the math notation of step 1. Indicate what the initial values are for that recurrence relation, and which term signifies the final solution.
4. Write an algorithm to compute the recurrence relation.

- Steps 1 and 2 need not be in that order. Do what makes sense in each problem.
- Step 3 is the heart of the design process. In high level algorithmic design situations, one can stop at step 3.
- Without the Principle of Optimality, it won't be possible to derive a sensible recurrence relation in step 3.
- When the Principle of Optimality holds, the 4 steps of DP are guaranteed to yield an optimal solution. No proof of optimality is needed.

2.3 MATRIX MULTIPLICATION

- Matrix multiplication a binary operation of multiplying two or more matrices one by one that are conformable for multiplication. For example two matrices A, B having the dimensions of $p \times q$ and $s \times t$ respectively; would be conformable for $A \times B$multiplication only if q==s and for $B \times A$ multiplication only if t==p.
- Matrix multiplication is associative in the sense that if A, B, and C are three matrices of order$m \times n$, $n \times p$ and $p \times q$ then the matrices (AB)C and A(BC) are defined as (AB)C = A (BC) and the product is an $m \times q$ matrix.
- Matrix multiplication is not commutative. For example two matrices A and B having dimensions $m \times n$ and $n \times p$ then the matrix AB = BA can't be defined. Because BA are not conformable for multiplication even if AB are conformable for matrix multiplication.
- For 3 or more matrices, matrix multiplication is associative, yet the number of scalar multiplications may very significantly depending upon how we pair the matrices and their product matrices to get the final product.

Example: Suppose there are three matrices A is 100×10, B is 10×50 and C is 50×5, then number of multiplication required for (AB)C is $AB = 100 \times 10 \times 5 = 50000$, $(AB)C = 100 \times 50 \times 5 = 25000$. Total multiplication for (AB)C= 75000 mulplication.Similarly number of multiplication required for A(BC) is BC = $10 \times 50 \times 5 = 2500$ and A(BC) = $100 \times 10 \times 5 = 5000$. Total multiplication for A(BC) = 7500.

In short the product of matrices (AB)C takes 75000 multiplication when first the product of A and B is computed and then product AB is multiplied with C. On the other hand, if the product BC is calculated first and then product of A with matrix BC is taken then 7500 multiplications are required. In case when large number of matrices are to be multiplied for which the product is defined, proper parenthesizing through pairing of matrices, may cause dramatic saving in number of scalar operations.
This raises the question of how to parenthesize the pairs of matrices within the expression $A_1A_2 \ldots A_n$, a product of n matrices which is defined; so as to optimize the computation of the product $A_1A_2 \ldots A_n$. The product is known as Chained Matrix Multiplication.

2.4 MATRIX CHAIN MULTIPLICATION

Given a sequence of matrices that are conformable for multiplication in that sequence, the problem of matrix-chain-multiplication is the process of selecting the optimal pair of matrices in every step in such a way that the overall cost of multiplication would be minimal. If there are total N matrices in the sequence then the total number of different ways of selecting matrix-pairs for multiplication will be $^{2n}C_n/(n+1)$. We need to find out the optimal one. In directly we can say that total number of different ways to perform the matrix chain multiplication will be equal to the total number of different binary trees that can be generated using N-1 nodes i.e. 2nCn/(n+1). The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications. Since matrix multiplication follows Associative property, rearranging the parentheses also yields the same result of multiplication. That means any pair in the sequence can be multiplied and that will not affect the end result: But the total number of multiplication operations will change accordingly. A product of matrices is fully satisfied if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$ then we can fully parenthesize the product $A_1A_2A_3A_4$ in five distinct ways:

$(A_1(A_2(A_3A_4)))$,
$(A_1((A_2A_3)A_4)$,
$(A_1(A_2A_3))A_4)$,
$((A_1A_2)(A_3A_4))$,
$(((A_1A_2)A_3)A_4)$.

How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. We shall use the dynamic-programming method to determine how to optimally parenthesize a matrix chain.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

Step 1: The structure of an optimal parenthesization:

- An optimal parenthesization of $A_1 \ldots A_n$ must break the product into two expressions, each of which is parenthesized or is a single array
- Assume the break occurs at position k.
- In the optimal solution, the solution to the product $A_1 \ldots A_k$ must be optimal
  - Otherwise, we could improve $A_1 \ldots A_n$ by improving $A_1 \ldots A_k$.
  - But the solution to $A_1 \ldots A_n$ is known to be optimal
  - This is a contradiction
  - Thus the solution to $A_1 \ldots A_n$ is known to be optimal
- This problem exhibits the Principle of Optimality:
  - The optimal solution to product $A_1 \ldots A_n$ contains the optimal solution to two subproducts
- Thus we can use Dynamic Programming
  - Consider a recursive solution
  - Then improve it's performance with memorization or by rewriting bottom up

Step 2: A recursive solution

For the matrix-chain multiplication problem, we pick as our sub problems the problems of determining the minimum cost of parenthesizing $A_i\, A_{i+1} \ldots A_j$ for $1 \leq i \leq j \leq n$.

- Let $m[i,j]$ be the minimum number of scalar multiplication needed to compute the matrix $A_{i\ldots j}$; for the full problem, the lowest cost way to compute $A_{1\ldots n}$would thus be $m[1,n]$.
- $m[i,i] = 0$, when i=j, problem is trivial. Chain consist of just one matrix $A_{i\ldots i} = A_i$, so that no scalar multiplications are necessary to compute the product.
- The optimal solution of $A_i \times A_j$must break at some point, k, with $\leq k < j$. Each matrix A_i is $p_{i-1} \times p_i$ and computing the matrix multiplication $A_{i\ldots k}\, A_{k+1\ldots j}$ takes $p_{i-1}p_k p_j$ scalar multiplication.

  Thus, $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$
  Equation 1

3. Computing the optimal costs

It is a simple matter to write a recursive algorithm based on above recurrence to compute the minimum cost $m[1,n]$ for multiplying $A_1 A_2 \ldots A_n$.

MATRIX-CHAIN-ORDER (P)

1. $n \leftarrow length[p] - 1$
2. let $m[1..n, 1..n]$ and $s[1..n-1, 2..n]$ be new tables
3. for i←1 to n
4. do $m[i,i] \leftarrow 0$
5. For l← 2 to n
6. do for i←1 to n-1+1
7. do j←i+l-1
8. $m[i,j] \leftarrow \infty$
9. for k←l to j-1
10. do $q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$
11. if q<$m[i,j]$
12. then $m[i,j] \leftarrow q$
13. $s[i,j] \leftarrow k$
14. return m and s

The following pseudo code assumes that matrix $\boldsymbol{A_i}$ has dimensions $\boldsymbol{p_{i-1} \times p_i}$ for **i=1,2,…,n**. The input is a sequence$\boldsymbol{p = p_0, p_1, \ldots, p_n}$, where **length[p]=n+1**. The procedure uses an auxiliary table $\boldsymbol{m[1 \ldots n, 1 \ldots n]}$ for storing the $\boldsymbol{m[i,j]}$ costs and the auxiliary table $\boldsymbol{s[1 \ldots n, 1 \ldots n]}$ that records which index of k achieved the optimal cost in computing $m[i,j]$. We will use the table **s** to construct an optimal solution.

- In line 3-4 algorithm first computes $m[i,i] = 0$ for $i = 1, 2, \ldots, n$ (the minimum costs for chains of length 1).
- During the first execution of for loop in line 5-13, it uses Equation (1) to computes$m[i, i+1]$ for $i = 1, 2, \ldots, n-1$ (the minimum costs for chains of length 2).
- The second time through the loop, it computes $m[i, i+2]$ for $i = 1, 2, \ldots, n-2$ (the minimum costs for chains of length 3, and so forth.
- At each step, the $m[i,j]$ cost computed in lines 10-13 depends only on table entries $m[i,k]$ and $m[k+1,j]$ already computed.

4. Constructing an Optimal Solution:

The MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplication needed to compute a matrix-chain product. To obtain the optimal solution of the matrix multiplication, we call **PRINT_OPTIMAL_PARES(s,1,n)** to print an optimal parenthesization of $\boldsymbol{A_1A_2, \ldots, A_n}$**.** Each entry $s[i, j]$ records a value of **k** such that an optimal parenthesization of $\boldsymbol{A_1A_2, \ldots, A_j}$splits the product between A_k and A_{k+1}.

```
PRINT_OPTIMAL_PARENS(s,i,j)
1. If i==j
2. Then print "A"i
3. else print "("
4. PRINT_OPTIMAL_PARENS(s, i, s[i,j])
5. PRINT_OPTIMAL_PARENS(s, s[i,j]+1, j)
6. print ")"
```

Example: Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is as follows:

Matrix	Dimension
A1	30 × 35
A2	35 × 15
A3	15 × 5
A4	5 × 10
A5	10 × 20

Solution:

Table m

j→ 1	2	3	4	5	i↓
0	15750	7875	9375	11875	1
	0	2625	4375	7125	2
		0	750	2500	3
			0	1000	4
				0	5

Table s

j→2	3	4	5	i↓
1	1	3	3	1
	2	3	3	2
		3	3	3
			4	4

Step 1: l=2

m[1 2]= m[1 1]+m[2 2] + $p_0p_1p_2$ = 0 + 0 + 30 ×35 × 15 = 15750 and k = 1 s[1 2]=1

m[2 3]= m[2 2] + m[3 3] + $p_1p_2p_3$ = 0 + 0 + 35 ×15 × 5 = 2625 and k=2 s[2 3]= 2

m[3 4]= m[3 3] + m[4 4] + $p_1p_3p_4$ = 0 + 0 + 15 ×5 × 10 = 750 and k=3 s[3 4]= 3

m[4 5]= m[4 4] + m[5 5]+ $p_3p_4p_5$ = 0 + 0 + 5 ×10 × 20 = 1000 and k =4 s[4 5] = 4

Step 2: l=3

m[1 3] = min (m[1 1] + m[2 3] + $p_0p_1p_3$, m[1 2] + m[3 3] + $p_0p_2p_3$)

= min (0 + 2625+ 30 ×35 × 5 , 15750+0+ 30 ×15 × 5)

= min(7875, 18000) = 7875 and k = 1 gives minimum s[1 3]= 1

m[2 4] = min (m[2 2] + m[3 4] + $p_1p_2p_4$, m[2 3] + m[4 4] + $p_1p_3p_4$)

= min (0 + 750+ 3 ×15 × 10 , 2625+0+ 35 ×5 × 10)

= min(6000, 4375) = 4375 and k = 3 gives minimum s[2 4]= 3

m[3 5] = min (m[3 3] + m[4 5] + $p_2p_3p_5$, m[3 4] + m[5 5] + $p_2p_4p_5$)

= min (0 + 1000+ 15 ×5 × 20 , 750+0+ 15 ×10 × 20)

= min(2500, 3750) = 2500 and k = 3 gives minimum s[3 5]= 3

Step 3: l=4

m[1 4] = min (m[1 1] + m[2 4] + $p_0p_1p_4$, m[1 2] + m[3 4] + $p_0p_2p_4$, $m[1\ 3] + m[4\ 4] + p_0p_3p_4$)

= min(0 + 4375 + 30 ×35 × 10, 15750 + 750 + 30 ×15 × 10, 7875+0+ 30 ×5 × 10)

= min(14875, 21900, 9375) = 9375 and k = 3 gives minimum s[1 4] = 3

m[1 4] = min (m[1 1] + m[2 4] + $p_0p_1p_4$, m[1 2] + m[3 4] + $p_0p_2p_4$, $m[1\ 3] + m[4\ 4] + p_0p_3p_4$)

= min(0 + 4375 + 30 ×35 × 10, 15750 + 750 + 30 ×15 × 10, 7875+0+ 30 ×5 × 10)

= min(14875, 21900, 9375) = 9375 and k = 3 gives minimum s[1 4] = 3

m[2 5] = min (m[2 2] + m[3 5] + $p_1p_2p_5$, m[2 3] + m[4 5] + $p_1p_3p_5$, $m[2\ 4] + m[5\ 5] + p_1p_4p_5$)

= min(0 + 2500 + 35 ×15 × 20, 2625 + 1000 + 35 ×5 × 20, 4375+0 + 35 ×10 × 20)

= min (13000, 7125, 11375) = 7125 and k = 3 gives minimum s[2 5] = 3

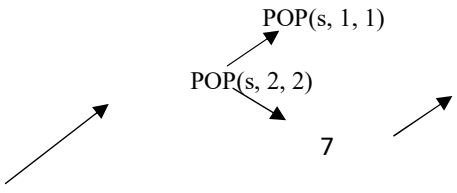
Step 4: l=5

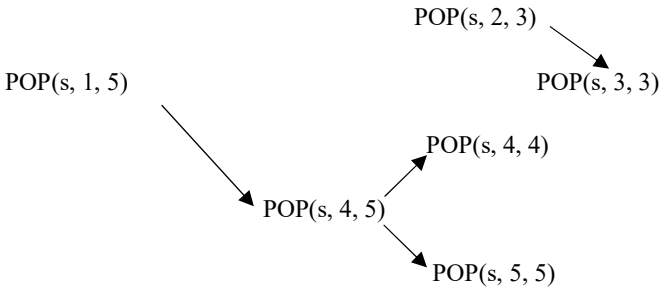
m[1 5] = min (m[1 1] + m[2 5] + $p_0p_1p_5$, m[1 2] + m[3 5] + $p_0p_2p_5$, $m[1\ 3] + m[4\ 5] + p_0p_3p_5$, m[1 4] + m[5 5] + $p_0p_4p_5$)

= min(0 + 7125 + 30 ×35 × 20, 15750+2500 + 30 ×15 × 20, 7875+ 1000 + 30 ×5 × 20, 9375+ 0 +30 ×10 × 20)

= min (28125, 27250, 11875, 15375) = 11875 and k = 3 gives minimum s[1 5] = 3

Now, print optimal parenthesis (POP) algorithm is runs:

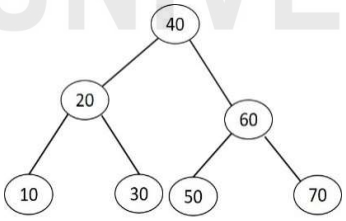




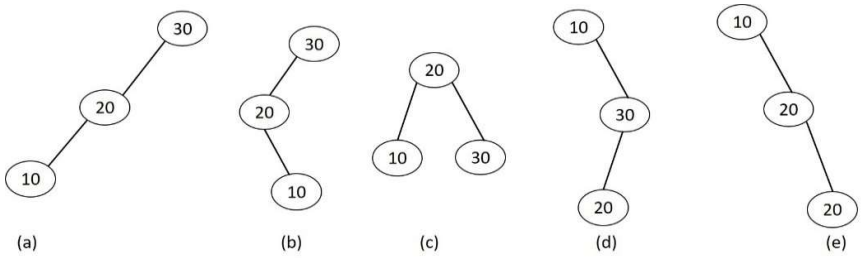
Hence, the final solution is $(A_1(A_2A_3)(A_4A_5))$.

2.5 OPTIMAL BINARY SEARCH TREE

Binary Search Tree: A binary tree is binary search tree such that all the key smaller than root are on left side and all the keys greater than root or on right side. Figure shown below is a binary search tree in which 40 is root node all the node in the left sub tree of 40 is smaller and all the node in the right sub tree of 40 is greater. No questions is why keys are arrange in order. Because it gives and advantage of searching. Suppose I want to search 30, we will start searching from the root node. Check whether root is a 30, no, then 30 is smaller than 40, definitely 30 will be in left sub tree of 40. Go in the left sub tree, check left sub tree root is 30, no, 20 is smaller than 30, definitely 30 will be in the right hand side. Yes now 30 is found. So to search the 30 how many comparison are need. Out of 7 elements to search 30 it, takes 3 comparison. No of comparison required to search any element in a binary search tree is depend upon the number of levels in a binary search tree. We are searching for the key that are already present in the binary search tree, is successful search. Now let's assume that key is 9 and search is an unsuccessful search because 9 is not found in the binary search tree. Total number of comparison it takes is also 3. There are a two possibility to search any element successful search and unsuccessful search. In both the cases we need to know the amount of comparison we are doing. That amount of comparison is nothing but the cost of searching in a binary search tree.



Let's take an example: Keys: 10, 20, 30 How many binary search tree possible. T(n)= $\frac{2n_{C_n}}{n+1}$ So there are 3 keys number of binary search tree possible = 5. Let's draw five possible binary search tree shown in below figure (a)-(e).



Now the cost of searching an element in the above binary tree is the maximum number of comparison. In tree (a), (b), (d), and (e) takes maximum three comparison whereas in tree (c) it takes only two comparison. It means that if you have set of n keys, there is a possibility that any tree gives you less number of comparison compare to other trees in all possible binary search tree of n keys. It means that if a height of binary search tree is less, number of comparison will be less. Height play an important role in searching any key in a binary search tree. So we want a binary search tree that requires less number of comparison for searching any key elements. This is called an optimal binary search tree.

The problem of optimal binary search tree is, given a keys and their probabilities, we have to generate a binary search tree such that the searching cost should be minimum. Formally we are given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order, and we wish to build a binary search tree from these keys. For each key k_i, we have a probability p_i that a search will be for k_i. Some searches may be for the values that is not in K, so we have also n+1 dummy keys $d_0, d_1, d_3, \ldots, d_n$ representing values not in K. In particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n, and for $i = 1, 2, 3, \ldots, n-1$, the dummy key d_i represents all values between k_i and k_{i+1}. For each dummy key d_i, we have a probability q_i that a search corresponds to d_i. Below Figure 2 shows the binary search tree for set of 5 keys. Each k_i is a internal node and each d_i is leaf node.

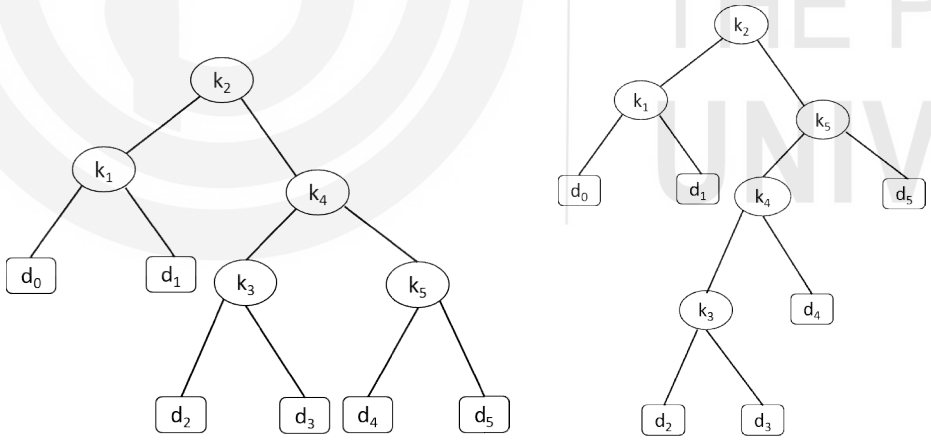


Figure 2: (a) (b)
Two Binary search tree with dummy keys with the following probabilities:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

(a) A binary search tree with expected search cost 2.80. (b) A binary search tree with expected search cost 2.75. This tree is optimal

Every search is either successful or unsuccessful, and so we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

Because we have probabilities of searches for each key and each key and each dummy key, we can determine the expected cost of a search is a given binary search tree T. Let we assume that actual cost of search equals the number of nodes examined, i.e., depth of the node found by the search in T, plus 1. Then the expected cost of a search in T is

$$E(search\ cost\ in\ T) = \sum_{i=1}^{n} (depth_T(k_i) + 1).p_i + \sum_{i=0}^{n} (depth_T(d_i) + 1).q_i$$

$= 1 + \sum_{i=1}^{n} depth_T(k_i).p_i + \sum_{i=0}^{n} depth_T(d_i).q_i,$

Where $depth_T$ denotes a node depth in the tree T. Expected search cost node by node given in following table:

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.20	0.80
Total			2.80

For a given set of probabilities, we wish to construct a binary search tree whose expected search cost is smallest. We call such tree an optimal binary search tree. Figure 2(b) shows an optimal binary search tree for the probabilities given in figure caption; its expected cost is 2.75. This example shows that an optimal binary search tree is not necessarily a tree whose overall height is smallest. Now question is we have n number of keys finding minimum cost means draw all possible trees and picking the tree having minimum cost is an optimal binary search tree. But this approach is not an efficient approach. Using dynamic programming we can find the optimal binary search tree without drawing each tree and calculating the cost of each tree. Thus, dynamic programming gives better, easiest and fastest method for trying out all possible binary search tree and picking up best one without drawing each sub tree.

Dynamic Programming Approach:

- Optimal Substructure: if an optimal binary search tree T has a subtree T` containing keys $k_i, \ldots, k_j$, then this sub tree T' must be optimal as well for the sub problem with keys $k_i, \ldots, k_j$and dummy keys $d_{i-1}, \ldots, d_j$.
- Algorithm for finding optimal tree for sorted, distinct keys $k_i, \ldots, k_j$:
    - For each possible root k_r for $i \leq r \leq j$

- Make optimal subtree for $k_i, \ldots, k_{r-1}$.
- Make optimal subtree for $k_{r+1}, \ldots, k_j$.
- Select root that gives best total tree

- Recursive solution: We pick our sub problem domain as finding an optimal binary search tree containing the keys $k_i, \ldots, k_j$, where $i \geq 1, j \leq n$, and $j \geq i - 1$. Let us define $e[i, j]$ as the expected cost of searching an optimal binary search tree containing the keys $k_i, \ldots, k_j$. Ultimately, we wish to compute $e[1, n]$.

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j=i-1 \\ \min_{i \leq r \leq j}\{e[i, r-1] + e[r+1, j] + w(i,j)\} & \text{if } i \leq j \end{cases} \quad \text{Equation 2}$$

where, $w(i,j) = \sum_{k=i}^{j} p_i$ is the increase in cost if $k_i, \ldots, k_j$ is a subtree of a node. When j=i-1, there are no actual keys; we have just the dummy key d_{i-1}.

Computing the Optimal Cost:

We store the $e[i, j]$ values in a table $e[1..n+1, 0..n]$. The first index needs to run to n+1 rather than n because in order to have a sub tree containing only the dummy key d_n, we need to compute and store $e[n+1, n]$. Second index needs to start from 0 because in order to have a sub tree containing only the dummy key d_0, we need to compute and store $e[1,0]$. We use only the entries $e[i, j]$ for which $j \geq i - 1$. We also use a table $root[i, j]$, for recording the root of the sub tree containing keys $k_i, \ldots, k_j$. This table uses only entries for which $1 \leq i \leq j \leq n$. We will need one other table for efficiency. Rather than compute the value of $w(i, j)$ from scratch every time we are computing $e[i, j]$ which would take $\theta(j - 1)$ additions- we store these values in a table $w[1..n+1, 0..n]$. For the base case, we compute $w[i, i-1] = q_{i-1}$ for $1 \leq i \leq n + 1$. For $j \geq i$, we compute

$$w[i,j] = w[i,j-1] + p_j + q_j \quad \text{Equation 3}$$

The pseudo code that follows takes as inputs the probabilities $p_1, \ldots, p_n$ and $q_0, \ldots, q_n$ and the size n, and returns the table e and root.

```
OPTIMAL_BST(p,q,n)
1.  let e[1..n+1,0..n], w[1..n+1,0..n] and root[1..n,1..n] be new tables.
2.  for i=1 to n+1
3.  e[i, i − 1] = q_{i−1}
4.  w[i, i − 1] = q_{i−1}
5.  for l=1 to n
6.      for i=1 to n-l+1
7.          J=i+l-1
8.  e[i, j] = ∞
9.  w[i, j] = w[i, j − 1] + p_j + q_j
10.         for r=i to j
11.             t= e[i,r-1]+e[r+1,j]+w[i, j]
12.             if t < e[i, j]
13. e[i, j] = t
14. root[i, j] = r
15. return e and root.
```

In the above algorithm:

- The for loop of lines 2-4 initializes the values of $e[i, i\text{-}1]$ and $w[i, i-1]$.
- The for loop of lines 5-14 then uses the recurrence in equation 2 to compute $e[i, j]$ and $w[i, j]$ for all $1 \leq i \leq j \leq n$.
- In the first iteration, when l=1, the loop computes $e[i, i]$ and $w[i, i]$ for $i = 1,2, \ldots, n$. The second iteration, when l=2, the loop computes $e[i, i+1]$ and $w[i, i+1]$ for $i = 1,2, \ldots, n-1$ and so forth.
- The innermost for loop, in lines 10-14, tries each candidates index r to determine which key $k_r$to use as the root of an optimal binary search tree containing keys $k_i, \ldots k_j$. This for loops saves the current value of the index r in $root[i, j]$whenever it finds a better key to use as the root.

Example of running the Algorithm:

Find the optimal binary search tree for N=4, having successful search and unsuccessful given in p_i and q_i rows.

Keys		10	20	30	40
$\mathbf{p_i}$		3	3	1	1
$\mathbf{q_i}$	2	3	1	1	1

Row 1

Let's fill up the values of **w**.

w[0 0]=q_0 w[1 1]=q_1 w[2 2]=q_2 w[3 3]=q_3 w[4 4]=q_4

Let's fill the cost **e**

Initial cost $\mathbf{e_{00}, e_{11}, e_{22}, e_{33}, e_{44}}$ will be 0. Similarly root **r** will be 0.

Row 2

Using equation 3**w** can be filled as:

$w[0\ 1]= w[0\ 0] + p_1 + q_1 \quad = \quad 2+3+3 = 8$

$w[1\ 2]= w[1\ 1] + p_2 + q_2 \quad = \quad 3+ 3 +1 = 7$

$w[2\ 3]= w[2\ 2] + p_3 + q_3 \quad =1+ \ 1 + 1=3$

$w[3\ 4]= w[3\ 3] + p_4 + q_4 \quad = \quad 1 + 1 +1 =3$

$e[0\ 1] = \min(e[0\ 0] + e[\ 1\ 1]) + w[0\ 1] = \min(0 + 0) + 8 = 8$

$e[1\ 2] = \min(e[1\ 1] + e[\ 2\ 2]) + w[1\ 2] = \min(\ 0 + 0) + 7 = 7$

$e[2\ 3] = \min(e[2\ 2] + e[\ 3\ 3]) + w[2\ 3] = \min(\ 0 + 0) + 3 = 3$

$e[3\ 4] = \min(e[3\ 3] + e[4\ \ 4]) + w[3\ 4] = \min(\ 0 + 0) + 3 = 3$

$r[0\ 1] = 1, \quad r[1\ 2] = 2, \quad r[2\ 3]= 3, \quad r[\ 3\ 4]=4$

j	**0**	**1**	**2**	**3**	**4**
j-i=0	$w_{00}= 2$ $e_{00}=0$ $r_{00}=0$	$w_{11}= 3$ $e_{11}=0$ $r_{11}=0$	$w_{22}= 1$ $e_{22}=0$ $r_{22}=0$	$w_{33}= 1$ $e_{33}=0$ $r_{33}=0$	$w_{44}= 1$ $e_{44}=0$ $r_{44}=0$
j-i=1	$w_{01}=8$ $e_{01}= 8$ $r_{01}= 1$	$w_{12}= 7$ $e_{12}= 3$ $r_{12}=2$	$w_{23}= 3$ $e_{23}= 3$ $r_{23}= 3$	$w_{24}= 3$ $e_{34}= 3$ $r_{34}= 4$	
j-i=2	$w_{02}= 12$ $e_{02}= 19$ $r_{02}=1$	$w_{13}= 9$ $e_{13}=12$ $r_{13}=2$	$w_{24}= 5$ $e_{24}=8$ $r_{24}=3$		
j-i=3	$w_{03}= 14$ $e_{03}=25$ $r_{03}=2$	$w_{14}= 11$ $e_{14}= 19$ $r_{14}=2$			
j-i=4	$w_{04}= 16$ $e_{04}=32$ $r_{04}=2$				

Table 1 shows the calculation of cost of the optimal binary search tree

Row 3

$w[0\ 2]= w[0\ 1] + p_2 + q_2 = 8 + 3 +1 = 12$

$w[1\ 3]= w[1\ 2] + p_3 + q_3 = 7 + 1 +1 =9$

w[2 4]= w[2 3] + p_4 + q_4 = 3 + 1 +1 =5

e[0 2] = k= 1, 2 = min(e[0 0] + e[1 2], e[0 1]+ e[2 2])+w[0 2] = min(0+7,8+0)+12 = 7+12=19 here k=1 has given minimum value thus, r[0 2]= 1

e[1 3]= k = 2, 3 = min(e[1 1]+ e[2 3], e[1 2]+e[3 3]) + w[1 3] = min(0+3, 7+0)+9 = 3+9= 12, here k=2 has given minimum value thus, r[1 3]=2

e[2 4]= k = 3, 4 = min(e[2 2]+e[3 4], e[2 3]+e[3 4])+w[2 4]= min(0+3, 3+3)+5 = 8, here k=3 has given minimum value thus, r[2 4]=3

Row 4

w[0 3]= w[0 2] + p_3 + q_3 = 12 + 1 + 1 = 14

w[1 4]= w[1 3] + p_4 + q_4 = 9 + 1 + 1 = 11

e[0 3] = k= 1, 2, 3 = min(e[0 0] + e[1 3], e[0 1]+ e[2 3], e[0 2]+ e[3 3])+w[0 3] = min(0+12,8+3, 19+0)+14 = 11+14=25 here k=2 has given minimum value thus, r[0 3]= 2

e[1 4]= k = 2, 3, 4 = min(e[1 1]+e[2 4], e[1 2]+e[3 4], e[1 3]+e[4 4])+w[1 4]= min(0+8, 7+3, 12+0)+11 = 19, here k=2 has given minimum value thus, r[1 4]= 2

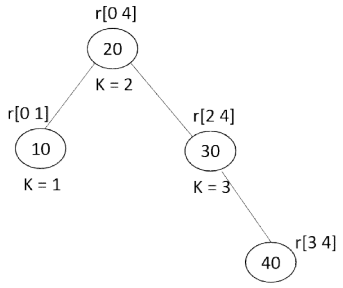
Row 5

w[0 4]= w[0 3] + p_4 + q_4 = 14 + 1 + 1 = 16

e[0 4]= k = 1, 2, 3, 4= min(e[0 0]+e[1 4], e[0 1]+ e[2 4], e[0 2]+e[3 4], e[0 3]+e[4 4] + w[0 4]=

min(0+19, 8+8, 19+3, 25+0]+16= 16+16=32 here k =2 has given minimum value thus, r[0 4]=2

Table 1 shows the calculation of cost matrix, weight matrix and root. While calculating the matrix, we have tried all possible binary search tree and select the minimum cost each time, thus it gives the minimum cost of the optimal binary search tree. Now we can draw the optimal binary search tree from the table 1. Below is a optimal binary search tree whose cost is minimum that is 32/16 = 2.



2.6 BINOMIAL COEFFICIENT COMPUTATION

Computing binomial coefficients is non optimization problem but can be solved using dynamic programming. **Binomial Coefficient** is the coefficient in the Binomial Theorem which

is an arithmetic expansion. It is denoted as c(n, k) which is equal to $\frac{n!}{k! \times (n-k)!}$where ! denotes factorial.This follows a recursive relation using which we will calculate the n binomial coefficient in linear time $O(n \times k)$using Dynamic Programming.

What is Binomial Theorem?

Binomial is also called as Binomial Expansion describe the powers in algebraic equations. Binomial Theorem helps us to find the expanded polynomial without multiplying the bunch of binomials at a time. The expanded polynomial will always contain one more than the power you are expanding.
Following formula shows the General formula to expand the algebraic equations by using Binomial Theorem:

$$(x+a)^n = \sum_{k=0}^{n} \binom{n}{k} x^k a^{n-k}$$

Where, n= positive integer power of algebraic equation and$\binom{n}{k}$= read as "n choose k".
According to theorem, expansion goes as following for any of the algebraic equation containing any positive power,

$$(a+b)^n = \binom{n}{0} a^n b^0 + \binom{n}{1} a^{n-1} b^1 + \binom{n}{2} a^{n-2} b^2 + \ldots + \binom{n}{n-1} a^1 b^{n-1} + \binom{n}{n} a^n b^n$$

Where, $\binom{n}{k}$ are binomial coefficients and$\binom{n}{k} = \boldsymbol{n_{c_k}}$gives combinations to choose k elements from n-element set. The expansion of binomial coefficient can be calculated as:$\frac{n!}{k! \times (n-k)!}$.

We can compute $\boldsymbol{n_{c_k}}$ for any n and k using the recurrence relation as follows:

$$\boldsymbol{n}_{C_k} = \begin{cases} 1, & \textit{if } k=0 \textit{ or } n=k \\ n-1 C_{k-1} + n-1 C_k, & \textit{for } n>k>0 \end{cases}$$

Computing C(n, k): Pseudo code:

BINOMIAL(n, k)

Input: A pair of nonnegative integers $n \geq k \geq 0$

Output: The value of $c(n, k)$

```
1.  for i←0 to n  do
2.    for j← 0 to min(i, k) do
3.      if  j=0  or  j=i
4.          c[i, j]←1
5.      else
6.          c[i, j]← c[i-1, j-1] + c[i-1, j]
7.  return c[n, k]
```

Let's consider an example for calculating binomial coefficient:

Compute binomial coefficient for n=5 and k=5

n/k	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

Table 3: gives the binomial coefficient

According to the formula when k=0 corresponding value in the table will be 1. All the values in column1 will be 1 when k=0. Similarly when n=k, value in the diagonal index will be 1.

c[2, 1] = c[1, 0]+c[1,1] = 1+1= 2 c[3, 1]= c[2,0] + c[2,1] = 1+2 = 3

c[4, 1] = c[3,0]+ c[3,1]= 1+3 = 4c[5, 1] = c[4, 0] + c[4, 1] = 1 + 4 = 5

c[3, 2] = c[2,1] + c[2, 2] = 2+1=3 c[4, 2] = c[3, 1] + c[3, 2] = 3 + 3 = 6

c[5, 2] = c[4, 1]+ c[4, 2] = 4 + 6 =10 c[4, 3] = c[3,2] + c[3, 3] = 3 + 1= 4

c[5, 3] = c[4, 2] + c[4, 3] = 6 + 4 = 10 c[5, 4] = c[4, 3] + c[4, 4]= 4 + 1= 5

Table 3 represent the binomial coefficient of each term.

Time Complexity:

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. Time complexity of the binomial coefficient is the size of the table i.e, O(n*k)

2.7 ALL PAIRSHORTEST PATH

Till now we have seen single source shortest path algorithms to find the shortest path from a given source S to all other vertices of graph **G**. But what if we want to know the shortest distance among all pair of vertices? Suppose we require to design a railway network where any two stations are connected with an optimum route. The simple answer to this problem would be to apply single source shortest path algorithms i.e. Dijkstra's or Bellman Ford algorithm for each vertex of the graph. The other approach may be to apply all pair shortest path algorithm which gives the shortest path between any two vertices of a graph at one go. We would see which approach will give the best result for different types of graph later on. But first we will discuss one such All Pair Shortest Path Algorithm i.e. Floyd Warshall Algorithm.

2.7.1 Floyd Warshall Algorithm

Floyd Warshall Algorithm uses the Dynamic Programming (DP) methodology. Unlike Greedy algorithms which always looks for local optimization, DP strives for global optimization that means DP does not relies on immediate best result. It works on the concept of recursion i.e. dividing a bigger problem into similar type of sub problems and solving them recursively. The only difference is- DP does save the outputs of the smaller subproblem and avoids the task of repetition in calculating the same subproblem. First it searches for the output of the sub problem in the table, if the output is not available then only it calculates the sub problem. This reduces the time complexity to a great extent. Due to its nature, DP is useful in solving the problems which can be divided into similar subproblems If each subproblem is different in nature, DP does not help in reducing the time complexity.

Now when we have the basic understanding of how the Dynamic Programming works and we have found out that the main crux of using DP in any problem is to identify the recursive function which breaks the problem into smaller subproblems. Let us see how can we create the recursive function for Floyd Warshall Algorithm (thereafter refers as FWA) which manages to find all pair shortest path in a graph.

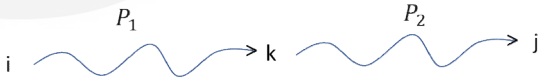
Note- Floyd Warshall Algorithm works on directed weighted graph including negative edge weight. Although it does not work on graph having negative edge weight cycle.

FWA uses the concept of intermediate vertex in the shortest path. Intermediate vertex in any simple path $\mathbf{P} = \{v_1, v_2 \ldots v_l\}$ is any vertex in **P** except source vertex v_1 and destination vertex v_l. Let us take a graph **G (V, E)** having vertex set **V= {1,2,3.....n}**. Take a subset of **V** as V_{s1}**={1,2...k}** for some value of **k**. For any pair of vertices $i, j \in V$ lists all paths from **i** to **j** such that any intermediate vertex in those paths belongs to V_{s1}. Find the shortest path P_s among them. Now there is one possibility between the two cases listed below-

1. The vertex **k** does not belong to the shortest path P_s.
2. The vertex **k** belongs to the shortest path P_s.

Case 1- P_s will also be the shortest path from **i** to **j** for another subset of **V** i.e. V_{s2}**={1,2...k-1}**

Case 2- We can divide path P_s into two paths P_1 and P_2 as below-



Where P_1 is the shortest path between vertices **i**and **k** and P_2 is the shortest path between vertices **k** and **j**. P_1 and P_2 both have the intermediate vertices from the set V_{s2}**= {1,2…k-1}**.

Based on the above understanding the recursive function for FWA can be derived as below-

$$d_{i,j}^{(k)} = \begin{cases} w(i,j) & \text{if } k = 0 \\ \min \{d_{i,j}^{(k-1)},\ d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\} & \text{if } k > 0 \end{cases}$$

Where,

$d_{i,j}^{(k)}$ - Weight of the shortest path from **i** to **j** for which all intermediate vertices ∈ **{1, 2...k}**

w(i,j) - Edge weight from vertex **i**to **j**

The recursive function implies that if there is no intermediate vertex from **i** to **j** then the shortest path shall be the weight of the direct edge. Whereas, if there are multiple intermediate vertices involved, the shortest path shall be the minimum of the two paths, one including a vertex k and another without including the vertex **k**.

2.7.2 Working Strategy for FWA

- Initial Distance matrix $\boldsymbol{D^0}$of order $\boldsymbol{n \times n}$ consisting direct edge weight is taken as the base distance matrix.
- Another distance matrix $\boldsymbol{D^1}$of shortest path $\boldsymbol{d_{i,j}^{(1)}}$including one (1) intermediate vertex is calculated where **i**, **j** ∈ **V**.
- The process of calculating distance matrices $\boldsymbol{D^2, D^3 \ldots D^n}$by including other intermediate vertices continues until all vertices of the graph is taken.
- The last distance matrix $\boldsymbol{D^n}$ which includes all vertices of a matrix as intermediate vertices gives the final result.

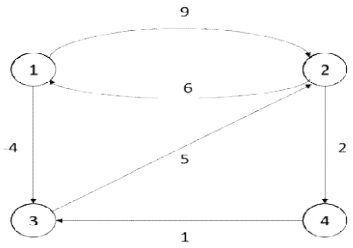
2.7.3 Pseudo-code for FWA

- **n** is the number of vertices in graph **G (V, E).**
- $\boldsymbol{D^k}$is the distance matrix of order $\boldsymbol{n \times n}$ consisting matrix element $\boldsymbol{d_{i,j}^{(k)}}$as the weight of shortest path having intermediate vertices from **1,2…k** ∈ **V**
- **M** is the initial matrix of direct edge weight. If there is no direct edge between two vertices, it will be considered as ∞.

```
FWA(M)
1. D^0 = M
2. For (k=1 to n)
3. For (i=1 to n)
4.                 For (j=1 to n)
5. d_{i,j}^{(k)} = min {d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}}
6. Return D^n
```

Since there are three nested for loops and inside which there is one statement which takes constant time for comparison and addition, the time complexity of FWA turns out to be $\boldsymbol{O(n^3)}$**.**

Example- Apply Floyd-Warshall Algorithm on the following graph-



We will create the initial matrix $\boldsymbol{D^0}$ from the given edge weights of the graph-

$$D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & \infty \\ 6 & 0 & \infty & 2 \\ \infty & 5 & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix} \end{array}$$

For creating distance matrices$\boldsymbol{D^1}$,$\boldsymbol{D^2}$ etc. We use two simple tricks for avoiding the calculations involved in each step-

1. Weight of shortest path from one vertex to itself will always be zero since we are dealing with no negative weight cycle. So we will put diagonal elements as zero in every iteration.
2. For the intermediate vertex **j**, we will put row and column elements of $\boldsymbol{D^j}$as it is for $\boldsymbol{D^{j-1}}$.

If we don't use the above tricks, still the result will be same.

Now for creating $\boldsymbol{D^1}$, we have to find distance between every two vertices via vertex 1. We will consider $\boldsymbol{D^0}$as base matrix as below-

$\boldsymbol{d^1[2,3]}$ = min $\{\boldsymbol{d^0[2,3]}, \boldsymbol{d^0[2,1] + d^0[1,3]} = \text{min } \{\infty, 6+(-4)\} = \text{min } \{\infty, 2\} = 2$

Note - *We have used* $\boldsymbol{d^k[i,j]}$ *instead of* $\boldsymbol{d_{i,j}^{(k)}}$ *for simplicity here.*

$\boldsymbol{d^1[2,4]}$ = min $\{\boldsymbol{d^0[2,4]}, \boldsymbol{d^0[2,1] + d^0[1,4]} = \text{min } \{2, 6+\infty\} = \text{min } \{2, \infty\} = 2$

$\boldsymbol{d^1[3,2]}$ = min $\{\boldsymbol{d^0[3,2]}, \boldsymbol{d^0[3,1] + d^0[1,2]} = \text{min } \{5, \infty+9\} = \text{min } \{5, \infty\} = 5$

Similarly, we will calculate for other vertices.

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & \infty \\ 6 & 0 & 2 & 2 \\ \infty & 5 & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix} \end{array}$$

For creating $\boldsymbol{D^1}$, we have to find distance between every two vertices via vertex 2. We will consider $\boldsymbol{D^1}$ as base matrix as below-

$\boldsymbol{d^2[1,3]}$ = min $\{\boldsymbol{d^1[1,3]}, \boldsymbol{d^1[1,2]} + \boldsymbol{d^1[2,3]}$ = min $\{-4, 9+2\}$ = min $\{-4, 11\} = -4$

$\boldsymbol{d^2[1,4]}$ = min $\{\boldsymbol{d^1[1,4]}, \boldsymbol{d^1[1,2]} + \boldsymbol{d^1[2,4]}$ = min $\{\infty, 9+2\}$ = min $\{\infty, 11\} = 11$

$\boldsymbol{d^2[3,1]}$ = min $\{\boldsymbol{d^1[3,1]}, \boldsymbol{d^1[3,2] + d^1[2,1]}$ = min $\{\infty, 5+6\}$ = min $\{\infty, 11\} = 11$

Similarly, we will calculate for other vertices.

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & 11 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ \infty & \infty & 1 & 0 \end{bmatrix} \end{array}$$

Create distance matrix $\boldsymbol{D^3}$ by taking vertex 3 as intermediate vertex and $\boldsymbol{D^2}$ as base matrix as below-

$$\boldsymbol{d^3[1,2]} = \min\{\boldsymbol{d^2[1,2]}, \boldsymbol{d^2[1,3]} + \boldsymbol{d^2[3,2]} = \min\{\mathbf{9}, -\mathbf{4} + \mathbf{5}\} = \min\{9,1\} = 1$$

$$\boldsymbol{d^3[1,4]} = \min\{\boldsymbol{d^2[1,4]}, \boldsymbol{d^2[1,3]} + \boldsymbol{d^2[3,4]} = \min\{\mathbf{11}, -\mathbf{4} + \mathbf{7}\} = \min\{11,3\} = 3$$

Similarly calculate for other pairs.

$$D^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & -4 & 3 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ 12 & 6 & 1 & 0 \end{bmatrix} \end{array}$$

Create distance matrix $\boldsymbol{D^4}$ by taking vertex 4 as intermediate vertex and $\boldsymbol{D^3}$ as base matrix as below-

$$\boldsymbol{d^4[1,2]} = \min\{\boldsymbol{d^3[1,2]}, \boldsymbol{d^3[1,4]} + \boldsymbol{d^3[4,2]} = \min\{\mathbf{1}, \mathbf{3} + \mathbf{6}\} = \min\{\mathbf{1}, \mathbf{9}\} = \mathbf{1}$$

Similarly calculate for other pairs-

$$D^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & -4 & 3 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ 12 & 6 & 1 & 0 \end{bmatrix} \end{array}$$

$\boldsymbol{D^4}$ gives the shortest path between any two vertices of given graph. e.g. weight of shortest path from vertex **1** to vertex **2** is **1**.

2.7.4 When FWA gives the best result

Do you remember the starting point of this topic where we stated that one can use Floyd Warshall Algorithm for finding all pair shortest path at one go or one can also use Dijkstra's or Bellman Ford algorithm for each vertex? We have analyzed the time complexity of FWA. Now we will find out the time complexities of running Dijkstra's and Bellman Ford algorithm for finding all pair shortest path to see which gives the best result-

Using Dijkstra's Algorithm - The time complexity of running Dijkstra's Algorithm for single source shortest path problem on graph **G(V,E)** is **O(E+VlogV)** using Fibonacci heap. For **G** being complete graph, running Dijkstra's Algorithm on each vertex will result in time complexity of **O($\boldsymbol{V^3}$).** For graph other than complete, it shall be **O(n*n log n),** less than FWA but since Dijkstra's Algorithm does not work with negative edge weight for single source shortest path problem it will also not work for all pair shortest path problem given negative edge weight.

Using Bellman Ford Algorithm – The time complexity of running Bellman Ford algorithm for single source shortest path problem on Graph **G (V,E)** is **O(VE).** If **G** is complete graph then this complexity turns out to be **O(V*$\boldsymbol{V^2}$)** i.e. **O($\boldsymbol{V^3}$).** Therefore,

the time complexity of running Bellman Ford Algorithm on each vertex of graph **G** shall be **O(V^4).**

From the above two points it can be concluded that FWA is the best choice for all pair shortest path problem when the graph is dense whereas Dijkstra's Algorithm is suitable when the graph is sparse and no negative edge weight exist. For graph having negative edge weight cycle the only choice among the three is Bellman Ford Algorithm.

2.8 SUMMARY

- The Dynamic Programming is a technique for solving optimization Problems, using bottom-up approach. The underlying idea of dynamic programming is to avoid calculating the same thing twice, usually by keeping a table of known results that fills up as substances of the problem under consideration are solved.
- In order that Dynamic Programming technique is applicable in solving an optimization problem, it is necessary that the principle of optimality is applicable to the problem domain.
- The principle of optimality states that for an optimal sequence of decisions or choices, each subsequence of decisions/choices must also be optimal.
- **The Chain Matrix Multiplication Problem:** The problem of how to parenthesize the pairs of matrices within the expression A1A2 … An, a product of n matrices which is defined; so as to minimize the number of scalar multiplications in the computation of the product A1A2 … An. The product is known as Chained Matrix Multiplication.

♣Check Your Progress:

Q1. Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is <5, 10, 3, 12, 5, 50, 6>.

Q2. Show that a full parenthesization of an n-element has exactly n-1 pairs of paranthesis.

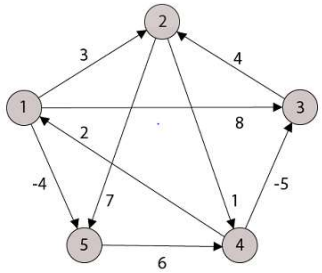
Q3. Determine the cost and structure of an optimal binary search tree for a set of n=7 keys with the following properties:

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

Q4.Determine the cost and structure of an optimal binary search tree for a set of n=7 keys with the following properties:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Q5. Apply Floyd-Warshall algorithm on the following graph. Show the matrix D^5 of the graph.



Q6. When graph is dense and have negative edge weight cycle. Which of the following is a best choice to find the shortest path

a) Bellman-ford algorithm
b) Dijakstra's algorithm
c) Floyd-Warshall algorithm
d) Kruskal's Algorithm

Q7. What procedure is being followed in Floyd Warshall Algorithm?
a) Top down
b) Bottom up
c) Big bang
d) Sandwich

Q8. What happens when the value of k is 0 in the Floyd Warshall Algorithm?
a) 1 intermediate vertex
b) 0 intermediate vertex
c) N intermediate vertices
d) N-1 intermediate vertices

2.9 SOLUTION / ANSWERS

♣Check Your Progress Answer:

Q1. Multiplication sequence is (A1A2)(A3A4)(A5A6)

m	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

s	1	2	3	4	5	6
1	0	1	2	2	4	2
2		0	2	2	2	2
3			0	3	4	4
4				0	4	4
5					0	5
6						0

Q5
.

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

Q6.Floyd-warshall algorithm

Q7. Bottom up

Q8. When k=0, a path from vertex i to vertex j has no intermediate vertices at all. Such a path has at most one edge and hence $d_{ij}^{0} = w_{ij}$