ITDO 5014 ADSA

Module 4

Dynamic Algorithms

ITDO 5014 ADSA

Module 4 : Dynamic Algorithms (6 Hrs)

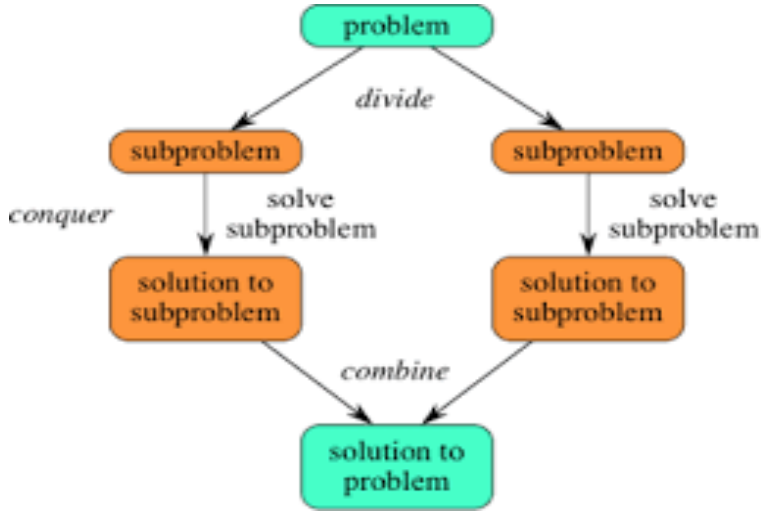
Reference : Horowitz, Sahani

1. Introduction Dynamic algorithms
2. All pair shortest path
3. 0/1 knapsack
4. Travelling salesman problem
5. Matrix Chain Multiplication
6. Optimal binary search tree (OBST)
7. Analysis of All algorithms and problem solving.

Self-learning Topics: Implementation of All pair shortest path, 0/1 Knapsack and OBST.

1. Introduction Dynamic algorithms :

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to sub problems.



Divide-and-conquer algorithms :

partition the problem into independent sub problems,
solve the sub problems recursively, and
then combine their solutions to solve the original problem.

In contrast, dynamic programming is applicable when the sub problems are not independent,

that is, when sub problems share sub problems.

In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub problems.

A dynamic-programming algorithm :

solves every sub problem just once ,

then saves its answer in a table / array and

reuses it to solve the overlapping sub problems

thereby avoiding the work of re-computing the answer every time the sub problem is encountered.

Like Greedy method, Dynamic programming is typically applied to *optimization problems.*

In such problems there can be many possible solutions.

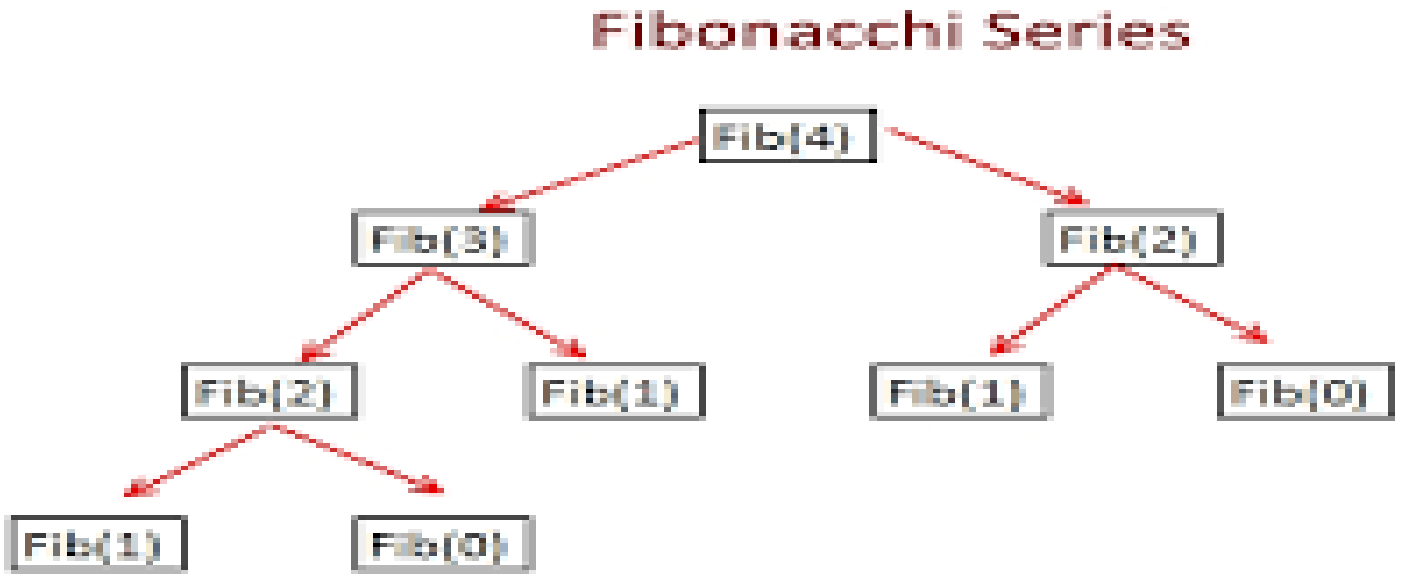
Each solution has a value, and

we wish to find a solution with the optimal (minimum or maximum) value.

We call such a solution *an* optimal solution to the problem.

```
Fibonacci(n)
{
        if (n==0) return 0;
        else if (n==1) return 1;
        else return ( Fibonacci(n-1) + Fibonacci(n-2);
}
```

n	0	1	2	3	4	5	6	7	8	9
Fibo(n)	0	1	1	2	3	5	8	13	21	34



Fib(0)	2
Fib(1)	3
Fib(2)	2
Fib( )	

Time Complexity of Recursive Fibonacci
= F(n) = F(n-1) + F(n-2) +1
⇒ O(2^n) ➔ Time required grows exponentially with input size n
(which is undesirable).

As shown in the table, in order to compute Fib (4), Fib(0) is called 2 times.
Fib(1) is called 3 times.
Fib(2) is called 2 times.

Fib(0)	2
Fib(1)	3
Fib(2)	2
Fib( )	

For very large n, Fib (0) ,Fib(2) ,….. are be called very large number of times.

Instead of re-computing the values of Fib (0) ,Fib(2) ,…..
by making time and space (for stack) consuming recursive calls ,

In dynamic programming,
we store their result in an array as : A[0] = 0, A[1] = 1, A[2] = 1 , ……
and
Reuse these values stored in an array so as to save the time and space required for re-executing the same function with the same input value again and again.

Thus, a dynamic-programming algorithm solves every sub problem just once and then saves its answer in a table/ array and reuses it to solve the overlapping sub problems, thereby avoiding the work of re computing the answer every time the sub problem is encountered.

Dynamic Algorithm for finding Fibonacci series is :

```
Fibo(n)
{
    Int A[n];
    A[0] = 0;
    A[1] = 1;
    For (i =  2 ; i<= n ; i++ )
        A[i] = A[i-1] +A[i-2];
    Return A[n];
}
```

n	0	1	2	3	4	5	6	7	8	9
Fibo(n)	0	1	1	2	3	5	8	13	21	34

Time Complexity of Recursive Fibonacci

= F(n) = F(n-1) + F(n-2) +1 Using Substitution method,

$\Rightarrow O(2^n)$ ➔ Time required grows exponentially with input size n (which is undesirable).

This requires Time = O (n) -for loop

Space = O (n) -for array A

Thus, dynamic programming usually uses an iterative approach whereas divide and conquer uses recursive method.

The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

1. Fibo(n) = 0 For n =0
   Fibo(n) = 1 For n =1
   Fibo(n) = Fibo (n-1) + Fibo (n-2) For n >= 2

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 in a bottom-up fashion.

4. Construct an optimal solution from computed information.

Steps 1-3 form the basis of a dynamic-programming solution to a problem.

Step 4 can be omitted if only the value of an optimal solution is required.

When we do perform step 4, we sometimes maintain additional information during the computation in step 3 to ease the construction of an optimal solution.

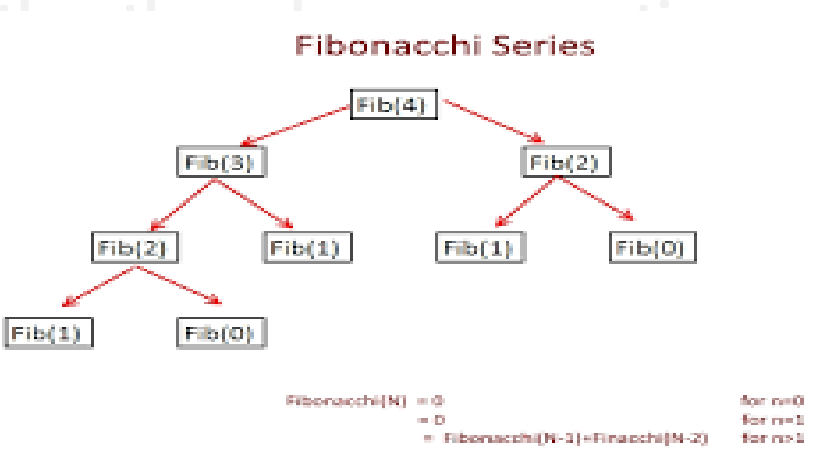
There are two key attributes that divide and conquer problem must have in order for dynamic programming to be applicable:

1. **Optimal substructure** — **optimal solution can be constructed from optimal solutions of its sub problems.**

2. **Overlapping sub-problems** — **problem can be broken down into sub problems which are reused several times or**

a recursive algorithm for the problem solves the same sub problem again and again

new sub problems



Applications of Dynamic Algorithms :

Problems that can be solved using Dynamic algorithms are :

1. All pair shortest path
2. 0/1 knapsack
3. Travelling salesman problem
4. Coin Changing Problem
5. Matrix Chain Multiplication
6. Flow shop scheduling
7. Optimal binary search tree (OBST)

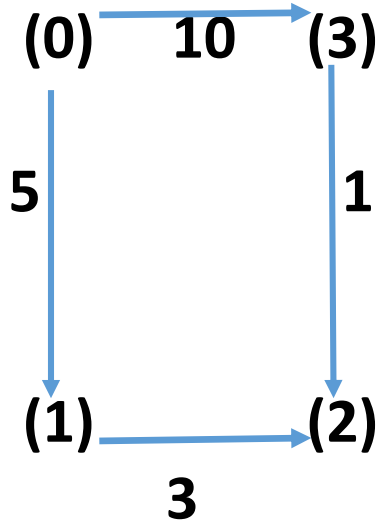
1. All pair shortest path : The Floyd Warshall Algorithm : is used for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Q. Apply all pair shortest path on the following graph. (10M)

Example: Input:

```
graph[][] = { { 0,   5,   INF,  10 },
              {INF,  0,    3,   INF},
              {INF,  INF,  0,    1  },
              {INF,  INF, INF,  0  } }
```

which represents the following graph



if i is equal to j :

the value of graph[i][j] is 0

if there is no edge from vertex i to j

graph[i][j] is INF (infinite).

Output:

Shortest distance matrix :

	0	1	2	3
0	0	5	8	9
1	INF	0	3	4
2	INF	INF	0	1
3	INF	INF	INF	0

Floyd Warshall Algorithm :

Q1. Explain All Pairs Shortest Path algorithm with suitable example. (10 M)

We initialize the solution matrix same as the input graph matrix as a first step.

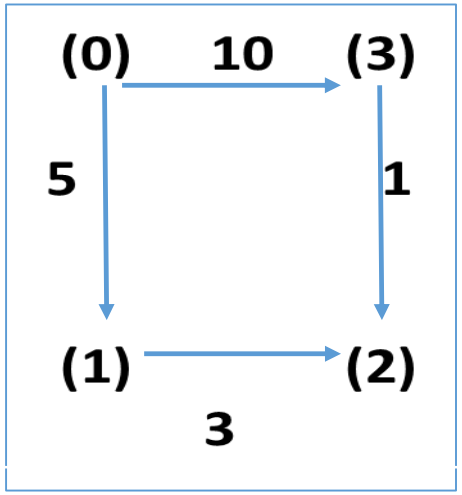
Then we update the solution matrix by considering all vertices as an intermediate vertex.

The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.

When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices.

For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

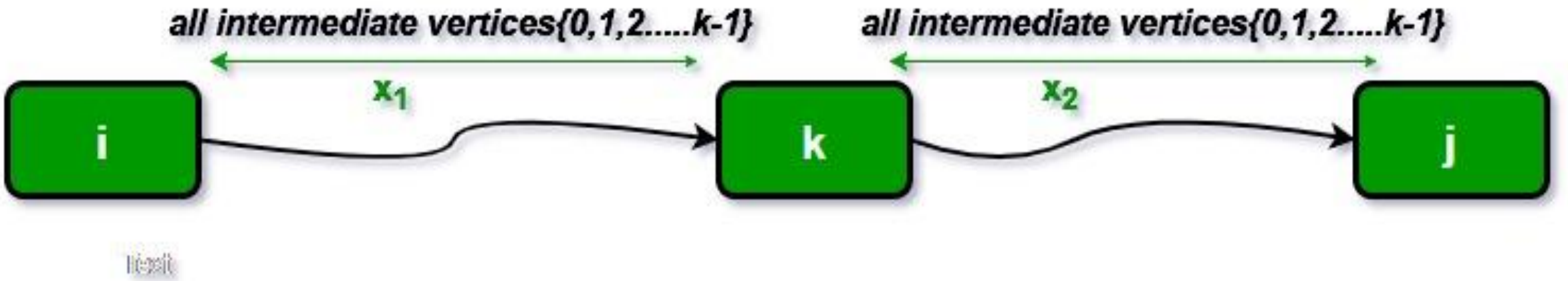
For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.



1) If k is not an intermediate vertex in shortest path from i to j then,
We keep the value of dist[i][j] as it is.

2) If k is an intermediate vertex in shortest path from i to j then,
We update the value of dist[i][j] as dist[i][k] + dist[k][j].

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.



Implementation of the Floyd Warshall algorithm for APSP :

```
// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough value.
This value will be used  for vertices not
 connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path problem
using Floyd Warshall algorithm
void floydWarshall (int graph[][V])
{
   /* dist[][] will be the output matrix that
will finally have the shortest
     distances between every pair of vertices
*/
   int dist[V][V], i, j, k;

   /* Initialize the solution matrix same as
      input graph matrix.  */

  for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)          O(n^2)
       dist[i][j] = graph[i][j];
```

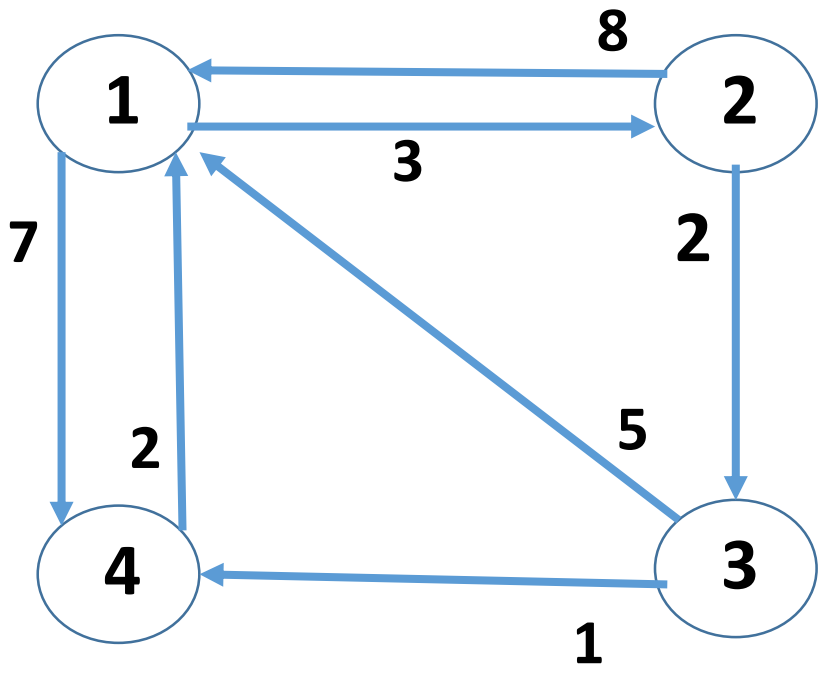
```
/* Add all vertices one by one to the set
of intermediate vertices.
---> Before start of an iteration, we have  shortest
 distances between all pairs of vertices such that
 the shortest distances consider only  the vertices
 in set {0, 1, 2, .. k-1} as intermediate vertices.
----> After the end of an iteration, vertex no. k is
 added to the set of intermediate vertices and
the set becomes {0, 1, 2, .. k} */
 for (k = 0; k < V; k++) // Pick every vertex k as intermediate vertex
 {
   for (i = 0; i < V; i++) // Pick all vertices i as source one by one
   {   // Pick all vertices j as destination for the
          // above picked source i
        for (j = 0; j < V; j++)                    O(n^3)
        {   // If vertex k is on the shortest path from
            // i to j, then update the value of dist[i][j]
if ( ( dist[i][k] + dist[k][j] ) < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
          }
       }
    }
     // Print the shortest distance matrix
    printSolution(dist);
}

 int graph[V][V] = { {0,   5,  INF, 10},
                     {INF, 0,   3, INF},
                     {INF, INF, 0,   1},
                     {INF, INF, INF, 0}
                   };
```

Function Call: `floydWarshall(graph);`

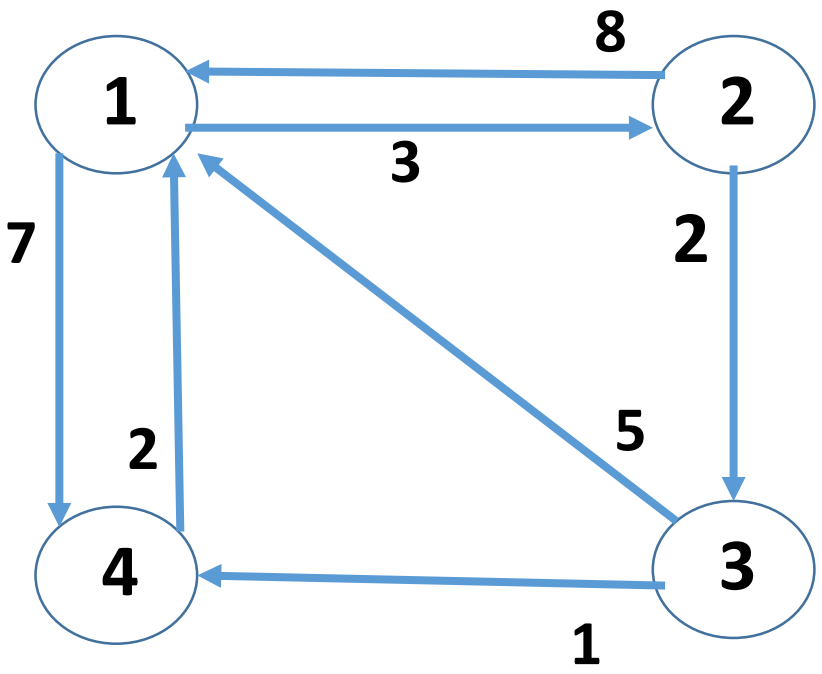
Time Complexity: $O(n^2) + O(n^3)$

$= O(n^3)$

Example : **Apply all pair shortest path on the following graph. (10M)**



Q. Apply all pair shortest path on the following graph. (10M)



0➔No Loop. Path from same vertex to itself.

∞ = INF = Absence of an edge between a pair of vertices.

A0=

	1	2	3	4
1	0	3	INF	7
2	8	0	2	INF
3	5	INF	0	1
4	2	INF	INF	0

Q… g graph. (10M)

A0[2,3] ? A0[2,1] + A0[1,3]
2 < 8 + INF

A0[2,4] ? A0[2,1] + A0[1,4]
INF > 8 + 7 (=15)

A0[3,2] ? A0[3,1] + A0[1,2]
INF > 5 + 3 (=8)

A0[3,4] ? A0[3,1] + A0[1,4]
1 < 5 + 7

A0[4,2] ? A0[4,1] + A0[1,2]
INF < 2 + 3 (=5)

A0[4,3] ? A0[4,1] + A0[1,3]
INF < 2 + INF (=INF)

1. Write graph in matrix form

A0=

	1	2	3	4
1	0	3	INF	7
2	8	0	2	INF
3	5	INF	0	1
4	2	INF	INF	0

3. Shortest path going via vertex 1 = A1

A1=

	1	2	3	4
1	0	3	INF	7
2	8	0	2	15
3	5	8	0	1
4	2	5	INF	0

Q… graph. (10M)

A1[1,3] ? A1[1,2] + A1[2,3]
INF > 3 + 2 (=5)

A1[1,4] ? A1[1,2] + A1[2,4]
7 < 3 + 15 (=18)

A1[3,1] ? A1[3,2] + A1[2,1]
5 < 8 + 8 (=16)

A1[3,4] ? A1[3,2] + A1[2,4]
1 < 8 + 15

A1[4,1] ? A1[4,2] + A1[2,1]
2 < 5 + 8 (=13)

A1[4,3] ? A1[4,2] + A1[2,3]
INF > 5 + 2 (=7)

1. Write graph in matrix form

A1=

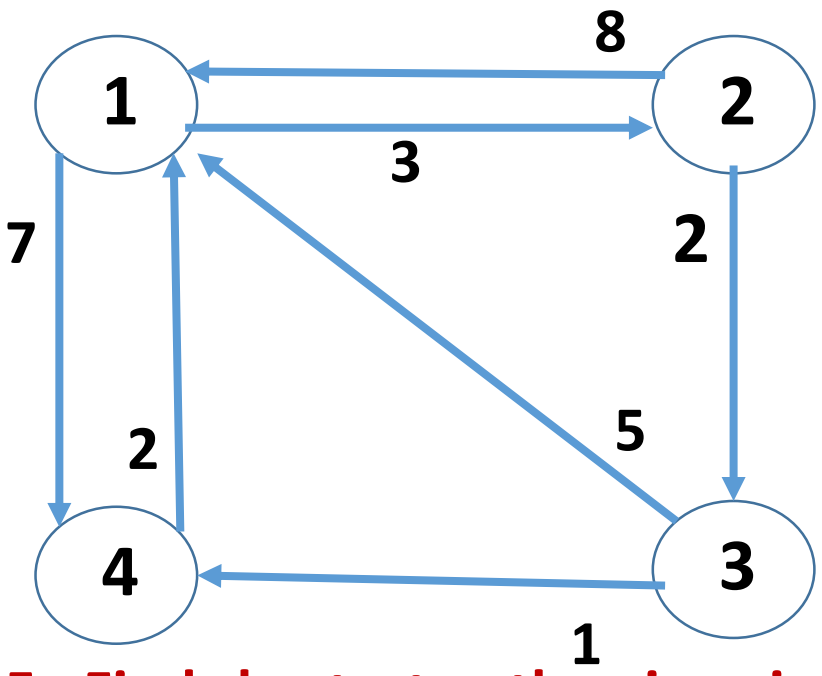
	1	2	3	4
1	0	3	INF	7
2	8	0	2	15
3	5	8	0	1
4	2	5	INF	0

4. Shortest path going via vertex 2 = A2

A2=

	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	5	8	0	1
4	2	5	7	0

Q. Apply all pair shortest path on the following graph. (10M)



5. Find shortest path going via vertex 3 = A3

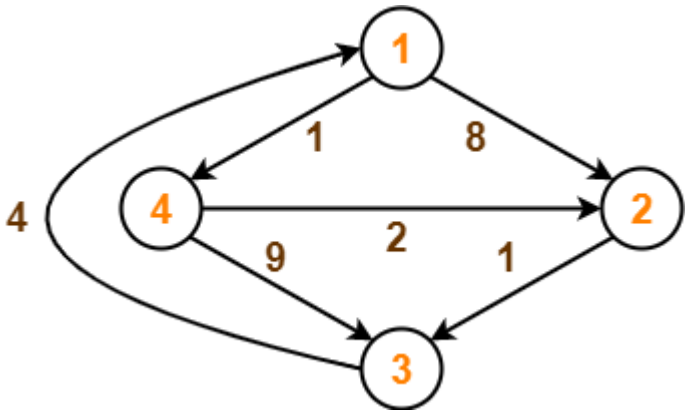
A3=

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

6. Find shortest path going via vertex 4 = A4

A4=

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



•Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph.

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

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

$$D_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$D_3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$D_4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

Source : www…

Example: Input:

graph[][] =
{ {0, 5, INF, 10},
{INF, 0, 3, INF},
{INF, INF, 0, 1},
{INF, INF, INF, 0} }

which represents the following graph
Note that the value of graph[i][j] is 0 if i is equal to j

And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

Output: Shortest distance matrix

0 5 8 9
INF 0 3 4
INF INF 0 1
INF INF INF 0

Describe the advantages of dynamic programming . How does it differ from divide and conquer. (10 M)

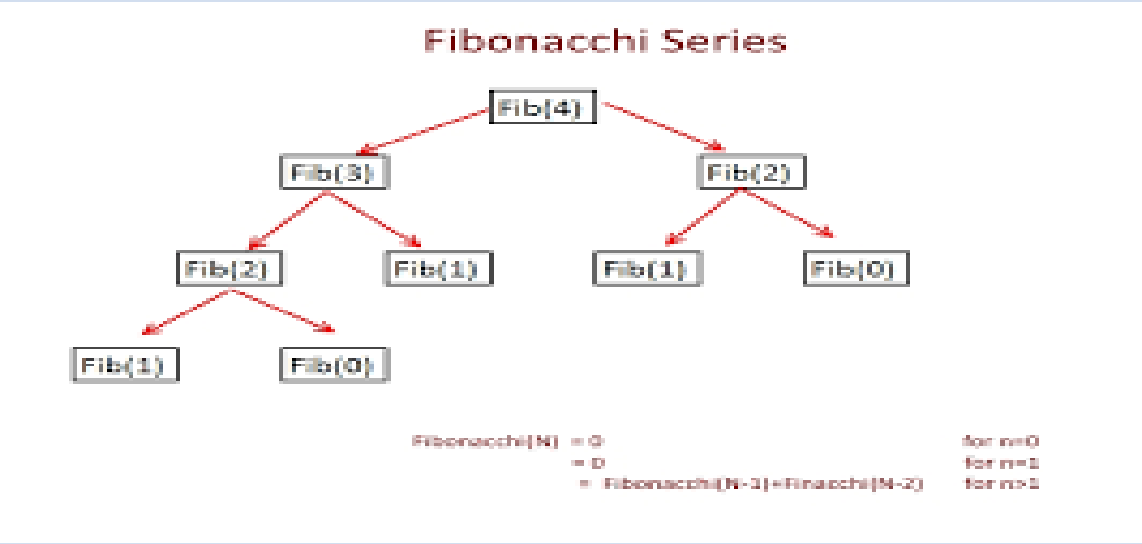
Divide and Conquer	Dynamic Programming
Partitions the problem into independent smaller sub problems.	Partitions the problem into overlapping sub problems
Does not store solution of smaller subpoblems	Stores the solution of smaller sub problems.
May solve the same sub problem multiple times.	Solves every sub problem just once and reuses it to save time and space required for re-computing the same sub problem.
Uses recursive approach.	Usually uses iterative approach.
Uses Top –down approach of problem solving. Divides larger problems into smaller sub problems thereby decreasing the problem size as we go down.	Uses bottom –up approach of problem solving. Solves smaller sub problems first and then combines their solution to solve the larger problem.
Example : Code and Explanation of nth Fibonacci number using D & C	Example : Code and Explanation of nth Fibonacci number using Dynamic Programming

Divide and Conquer

```
Fibonacci(n)
{
        if (n==0) return 0;
        else if (n==1) return 1;
        else return ( Fibonacci(n-1) + Fibonacci(n-2);
}
```

In order to compute Fib (4), D&C calls Fib(0) 2 times.
Fib(1) 3 times.
Fib(2) 2 times….

i.e. D&C re-computs the values of Fib (0) ,Fib(1), Fib(2)… by making time and space (for stack) consuming recursive calls.



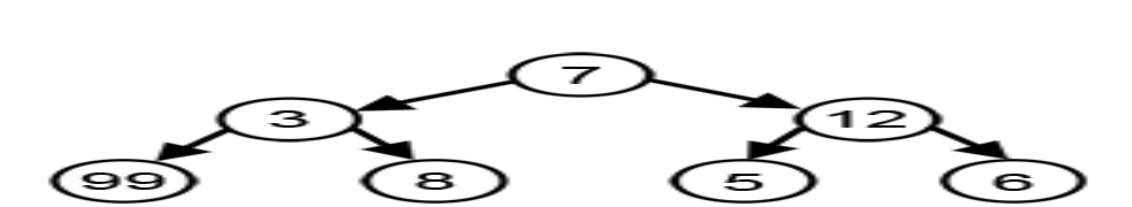
Dynamic Programming

```
Fibo(n)
{       Int A[n];
        A[0] = 0;
        A[1] = 1;
        For (i =  2 ; i<= n ; i++ )
              A[i] = A[i-1] +A[i-2];
        Return A[n];
}
```

n	0	1	2	3	4	5	6	7	8	9
Fibo(n)	0	1	1	2	3	5	8	13	21	34

In dynamic programming,
we store their result in an array as : A[0] = 0, A[1] = 1, A[2] = 1 , ……
and
Reuse these values stored in an array so as to save the time and space required for re-executing the same function with the same input value again and again.

Q. Compare Greedy and Dynamic Programming approach for an algorithm design. Explain how both can be used to solve knapsack problem.
10M (SH 2018)

Greedy	Dynamic Programming
A greedy algorithm is one that at a given point in time, makes a local optimization. 	Dynamic programming can be thought of as 'smart' recursion. It often requires one to break down a problem into smaller components that can be cached.
Greedy algorithms have a local choice of the subproblem that will lead to an optimal answer	Dynamic programming solves all dependent sub problems and then select one that would lead to an optimal solution.
A greedy algorithm is one which finds optimal solution at each and every stage with the hope of finding global optimum at the end.	A Dynamic algorithm is applicable to problems that exhibit Overlapping sub problems and Optimal substructure properties.
More efficient as compared,to dynamic programming	Less efficient as compared to greedy approach
Example : Fractional Knapsack	Example : 0/1 Knapsack

0/1 or Integer Knapsack Problem :

Q1. What is 0/1 knapsack and fractional knapsack problem? Solve the following using 0/1 knapsack method (10 M)

Q2. Write an algorithm for 0/1 knapsack problem using DP approach. (10 M)

Given weights and price/values of n items, put these items in a knapsack of capacity W to get the maximum total price/value in the knapsack.

In other words, given two integer arrays price/val[0..n-1] and wt[0..n-1] which represent price/values and weights associated with n items respectively.

Also given an integer W which represents knapsack capacity, find out the maximum value subset of price/val[] such that sum of the weights of this subset is smaller than or equal to W.

You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

0-1 Knapsack Problem

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets.
Consider the only subsets whose total weight is smaller than W.
From all such subsets, pick the maximum value subset.

1) Optimal Substructure:

To consider all subsets of items, there can be two cases for every item:
(1) the item is included in the optimal subset,
(2) not included in the optimal set.

So, the maximum value that can be obtained from n items is max of following two values.
1) Maximum value obtained by n-1 items and W weight (excluding nth item).
2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

```
// A Dynamic Programming based solution
for 0-1 Knapsack problem

// A utility function that returns maximum
of two integers
int max(int a, int b) { return (a > b)? a : b; }
// No. of items given = n
// Returns the maximum value that can be
 put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
   int i, w;
   int K[n+1][W+1];
// Build table K[][] in bottom up manner
  for (i = 0; i <= n; i++)                    O(n)
  {
     for (w = 0; w <= W; w++)                 O(W)
     {
        if (i==0 || w==0)
           K[i][w] = 0;
        else if (wt[i-1] <= w)
              K[i][w] = max(val[i-1] +
              K[i-1][w-wt[i-1]],  K[i-1][w]);
        else
              K[i][w] = K[i-1][w];
     }
  }
 return K[n][W];
}
```

```
int main()
{
   int val[] = {60, 100, 120};
   int wt[] = {10, 20, 30};
   int  W = 50;
   int n = sizeof(val)/sizeof(val[0]);
   printf("%d", knapSack(W, wt, val, n));
   return 0;
}
```

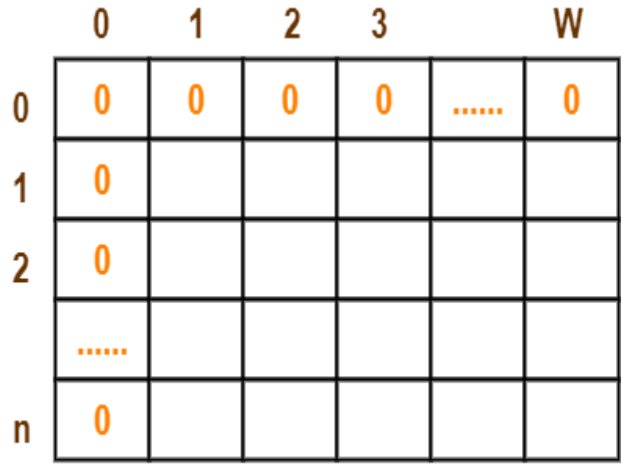
Time Complexity: O(nW) where n is the number of objects and W is the capacity of knapsack.

Procedure for solving 0/1 knapsack prolem:

Step-01:

n=no of objects, w=Capacity of knapsack

- Draw a table say 'T' with (n+1) number of rows and (w+1) number of columns.
- Fill all the boxes of 0^{th} row and 0^{th} column with zeroes as shown-



T-Table

Step-02:

Start filling the table row wise top to bottom from left to right.

Use the following formula-

$$T(i, j) = \max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}$$

Here, T(i , j) = maximum value of the selected items if we can take items 1 to i and have weight restrictions of j.

This step leads to completely filling the table.

Then, value of the last box represents the maximum possible value that can be put into the knapsack.

For the given set of items and knapsack capacity = 5 kg, find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach.

Knapsack capacity (w) = 5 kg
Number of items (n) = 4

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

Step-01:

Draw a table say 'T' with (n+1) = 4 + 1 = 5 number of rows and (w+1) = 5 + 1 = 6 number of columns.
Fill all the boxes of 0th row and 0th column with 0.

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T(i, j) = \max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}$$

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

$T(i, j) = \max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}$

Finding T(1,1)- $i = 1, j = 1$

$(value)_i = (value)_1 = 3$ $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$T(1,1) = \max \{ T(1-1, 1), 3 + T(1-1, 1-2) \}$

$T(1,1) = \max \{ T(0,1), 3 + T(0,-1) \}$

$T(1,1) = T(0,1)$ { Ignore T(0,-1) }

$T(1,1) = 0$

Finding T(1,2)- $i = 1, j = 2$

$(value)_i = (value)_1 = 3$ $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$T(1,2) = \max \{ T(1-1, 2), 3 + T(1-1, 2-2) \}$

$T(1,2) = \max \{ T(0,2), 3 + T(0,0) \}$

$T(1,2) = \max \{0, 3+0\}$

$T(1,2) = 3$

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

	0	1	2	3	4	5
0	0	0	0	0	0	0
✓ 1	0	0	3	3	3	3
✓ 2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

Finding T(1,3)- **i = 1, j = 3** $T(i, j) = \max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}$

$(value)_i = (value)_1 = 3$ $(weight)_i = (weight)_1 = 2$

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

Substituting the values, we get-

$T(1,3) = \max \{ T(1-1, 3), 3 + T(1-1, 3-2) \}$

$T(1,3) = \max \{ T(0,3), 3 + T(0,1) \}$

$T(1,3) = \max \{0, 3+0\}$

$T(1,3) = 3$

Finding T(1,4)- **i = 1** **j = 4**

$(value)_i = (value)_1 = 3$ $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$T(1,4) = \max \{ T(1-1, 4), 3 + T(1-1, 4-2) \}$

$T(1,4) = \max \{ T(0,4), 3 + T(0,2) \}$

$T(1,4) = \max \{0, 3+0\}$

$T(1,4) = 3$

	0	1	2	3	4	5
0	0	0	0	0	0	0
✓ 1	0	0	3	3	3	3
✓ 2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

Finding T(1,5)- i = 1 j = 5

$(value)_i = (value)_1 = 3$ $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,5) = max { T(1-1 , 5) , 3 + T(1-1 , 5-2) }

T(1,5) = max { T(0,5) , 3 + T(0,3) }

T(1,5) = max {0 , 3+0}

T(1,5) = 3

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

Finding T(2,1)- i = 2 j = 1

$(value)_i = (value)_2 = 4$ $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,1) = max { T(2-1 , 1) , 4 + T(2-1 , 1-3) }

T(2,1) = max { T(1,1) , 4 + T(1,-2) }

T(2,1) = T(1,1) { Ignore T(1,-2) }

T(2,1) = 0

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

T-Table

Finding T(2,2)- **i = 2** **j = 2**

$(value)_i = (value)_2 = 4$ **$(weight)_i = (weight)_2 = 3$**

Substituting the values, we get-

T(2,2) = max { T(2-1 , 2) , 4 + T(2-1 , 2-3) }

T(2,2) = max { T(1,2) , 4 + T(1,-1) }

T(2,2) = T(1,2) **{ Ignore T(1,-1) }**

T(2,2) = 3

Finding T(2,3)- **i = 2** **j = 3**

$(value)_i = (value)_2 = 4$ **$(weight)_i = (weight)_2 = 3$**

Substituting the values, we get-

T(2,3) = max { T(2-1 , 3) , 4 + T(2-1 , 3-3) }

T(2,3) = max { T(1,3) , 4 + T(1,0) }

T(2,3) = max { 3 , 4+0 }

T(2,3) = 4

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

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

T-Table

Finding T(2,4)- **i = 2 j = 4**

$(value)_i = (value)_2 = 4$ **$(weight)_i = (weight)_2 = 3$**

Substituting the values, we get-

T(2,4) = max { T(2-1 , 4) , 4 + T(2-1 , 4-3) }

T(2,4) = max { T(1,4) , 4 + T(1,1) }

T(2,4) = max { 3 , 4+0 }

T(2,4) = 4

Finding T(2,5)- **i = 2 j = 5**

$(value)_i = (value)_2 = 4$ **$(weight)_i = (weight)_2 = 3$**

Substituting the values, we get-

T(2,5) = max { T(2-1 , 5) , 4 + T(2-1 , 5-3) }

T(2,5) = max { T(1,5) , 4 + T(1,2) }

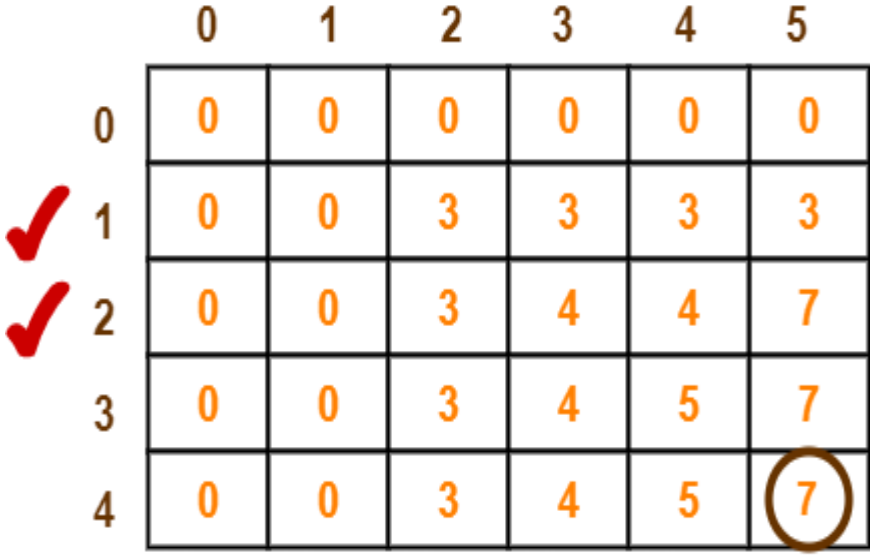
T(2,5) = max { 3 , 4+3 }

T(2,5) = 7

Similarly, compute all the entries.

After all the entries are computed and filled in the table, we get the following table-

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6



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

T-Table

The last entry represents the maximum possible value that can be put into the knapsack.

So, maximum possible value that can be put into the knapsack = 7.

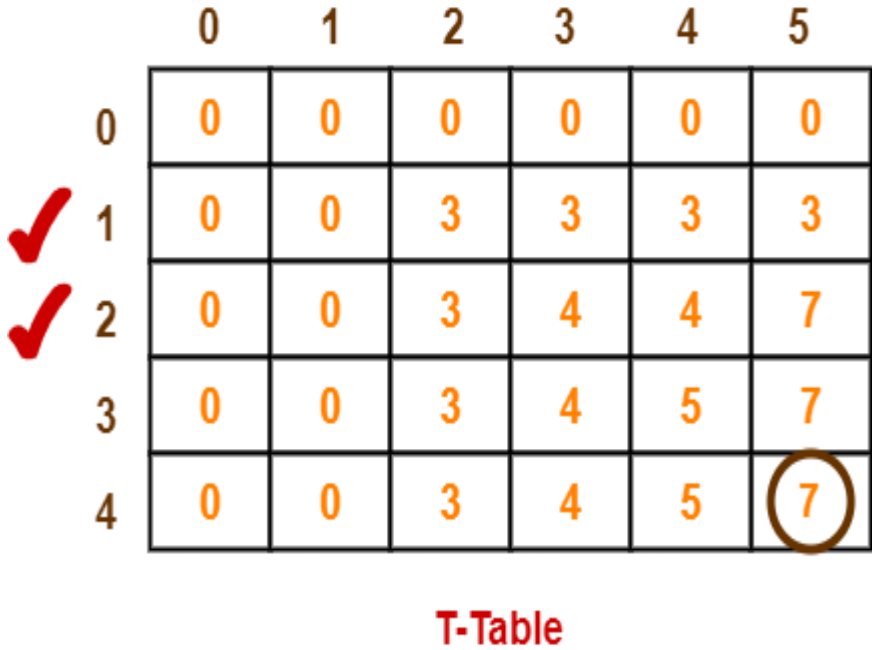
Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

Identifying Items To Be Put Into Knapsack-

Following Step-04,

We mark the rows labelled “1” and “2”.

Thus, items that must be put into the knapsack to obtain the maximum value 7 are



	0	1	2	3	4	5
0	0	0	0	0	0	0
1 ✓	0	0	3	3	3	3
2 ✓	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

How to find actual Knapsack Items :

$B[n,W]$ is the maximal value of items that can be placed in the Knapsack.

Let i=n and k=W

if $B[i,k] \neq B[i-1,k]$ then

add the i^{th} item in the knapsack

mark the i^{th} item as added in the knapsack

$i = i-1, k = k-w_i$

else $i = i-1$ // i^{th} item is not added in the knapsack

// Could it be in the optimally packed knapsack?

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

k =	0	1	2	3	4	5
i = 0	0	0	0	0	0	0
✓ 1	0	0	3	3	3	3
✓ 2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

Knapsack capacity (w) = 5 kg

Number of items (n) = 4 ➔

i = 4 , k = 5 :

if $B[i,k] \neq B[i-1,k]$ ➔ B[4,5] == ? B[3, 5]

➔ 7=7 ➔ i = i – 1 ➔ i = 3

i = 3, k = 5 :

if $B[i,k] \neq B[i-1,k]$ ➔ B[3,5] == B[2, 5] ➔

i = i – 1 ➔ i = 2

B[3, 5] ? B[2, 5] => 7 ? 7 ➔ 7 = 7

i = i – 1 ➔ i = 2

i = 2 k = 5 :

if $B[i,k] \neq B[i-1,k]$ ➔ B[2,5] ≠ B[1, 5] ➔ 7 ≠ 3

➔ Add ith item i.e. item 2 in knapsack and mark ad added

➔ i = i – 1 ➔ i = 2-1 = 1 ,

➔ $k = k-w_i$ $k = 5 – w2 = 5 – 3 = 2$

i = 1, k = 2

if $B[i,k] \neq B[i-1,k]$ ➔ B[1,2] ≠ B[0, 2]

➔ 3 == 0 ? 3 ≠ 0 ➔ Add and Mark ith i.e. item 1 in knapsack, i = i – 1 ➔ i = 0 ,

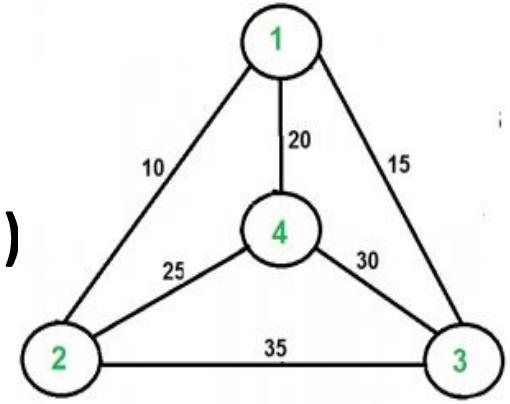
➔ $k = k-w_i$ $k = 2 – w1 = 2 – 2 = 0$

Ans : Total value i.e profit = 7 (∑xiPi = 1*3 + 1*4 = 7)

Items added in knapsack xi = (1, 1, 0, 0)

∑xiwi = 1*2 + 1*3 = 5 <= w = 5

Travelling salesman problem (TSP) (NP Hard) :



Q1. Write a note on Travelling salesman problem. (10 M)
Q2. Find the path of Travelling salesman problem for the given graph. (10 M)

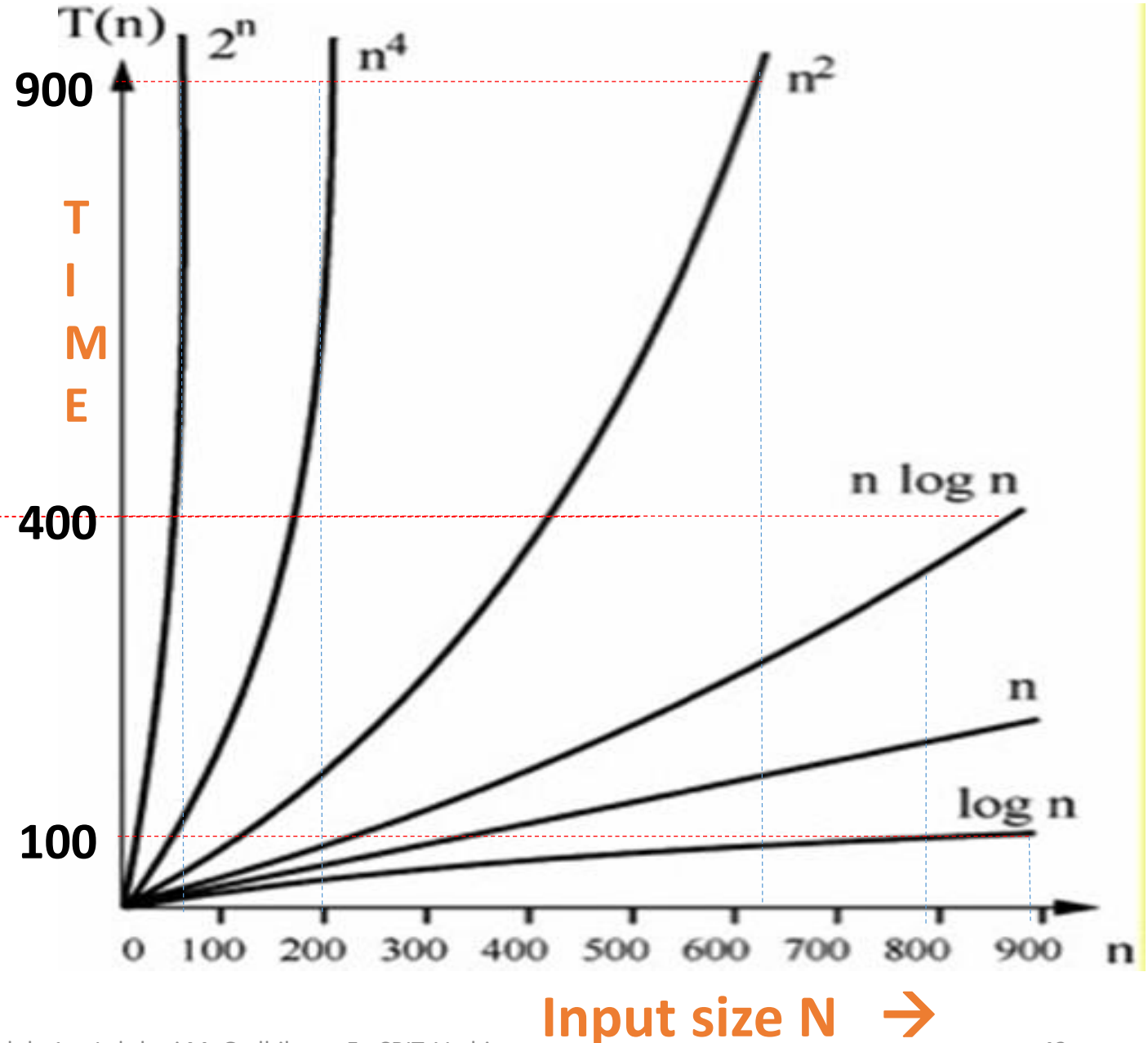
Given a set of cities and distance between every pair of cities,
the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

The difference between Hamiltonian Cycle and TSP is :

The Hamiltonian cycle problem is to find if there exist a tour that visits every city exactly once.

In TSP, we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

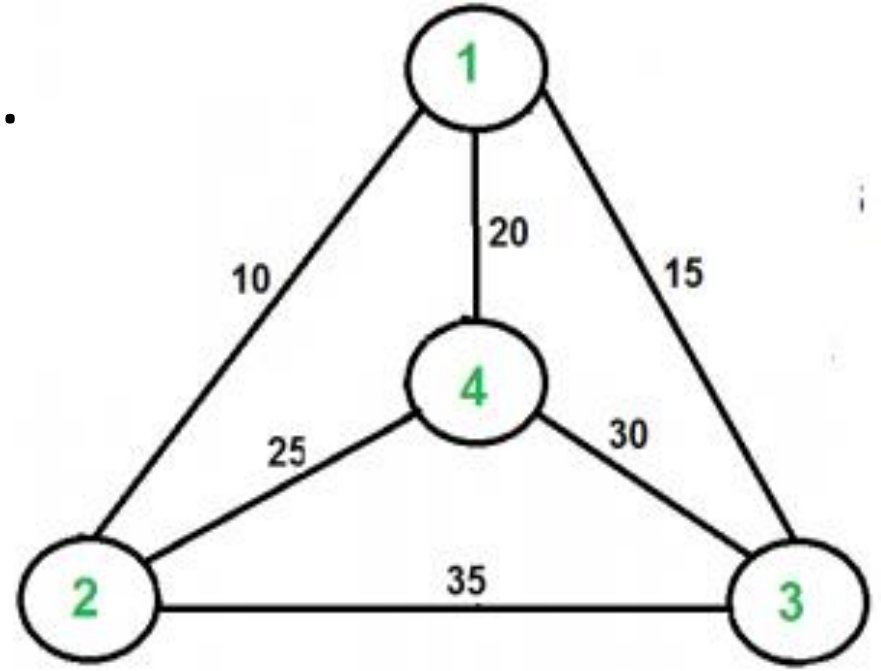
is proportional to:	Complexity:
$T(n) \propto \log n$	logarithmic
$T(n) \propto n$	linear
$T(n) \propto n \log n$	linearithmic
$T(n) \propto n^2$	quadratic
$T(n) \propto n^3$	cubic
$T(n) \propto n^k$	Polynomial
$T(n) \propto 2^n$	exponential
$T(n) \propto k^n; k > 1$	exponential



For example, consider the graph shown in figure on right side.

A TSP tour in the graph is 1-2-4-3-1.

The cost of the tour is 10+25+30+15 which is 80.



The problem is a famous NP hard problem.

There is no polynomial time (n^k) known solution for this problem.

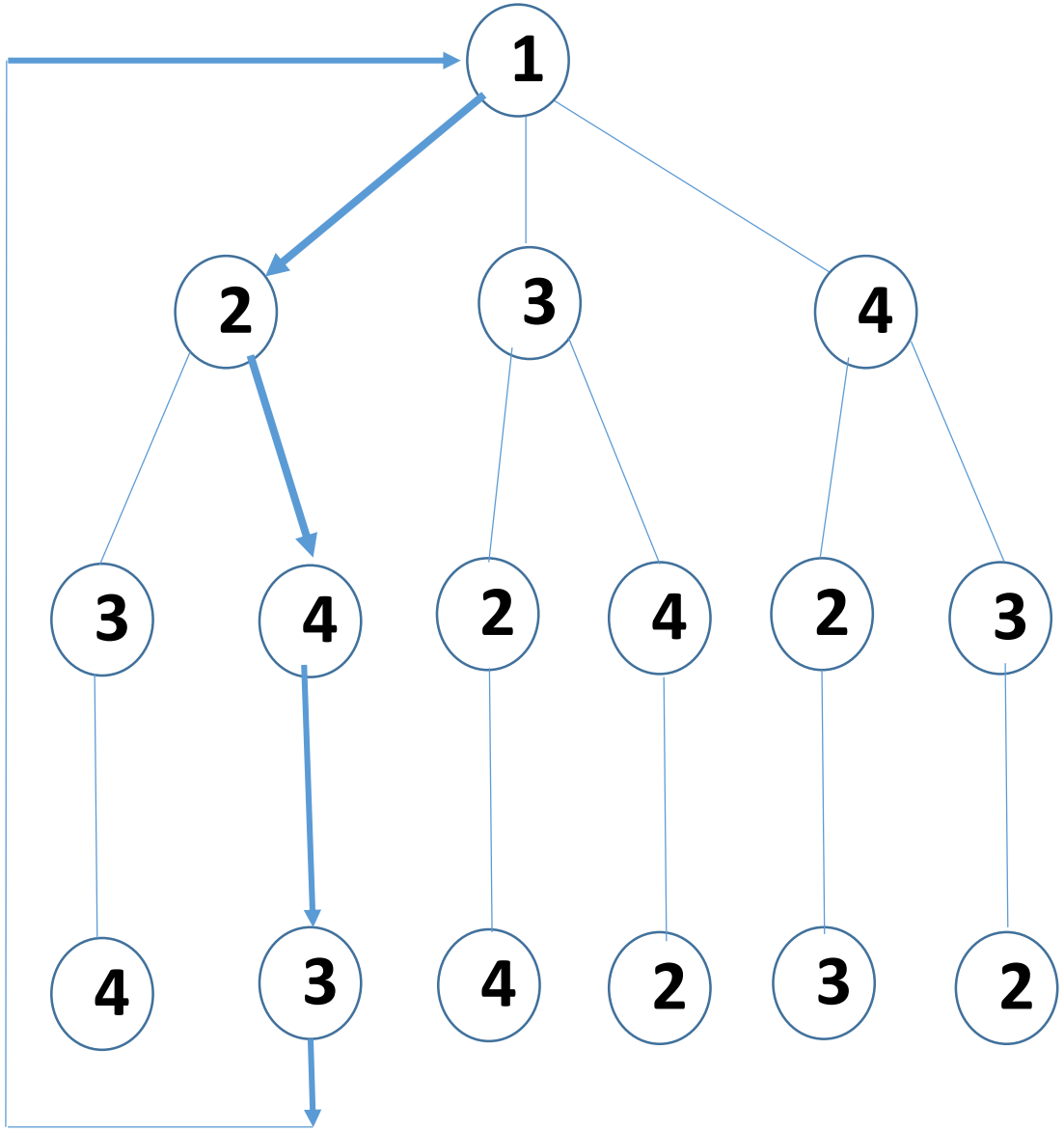
Following are different methods for traveling salesman problem.

1. Brute force method / Naïve solution
2. Dynamic Programming method

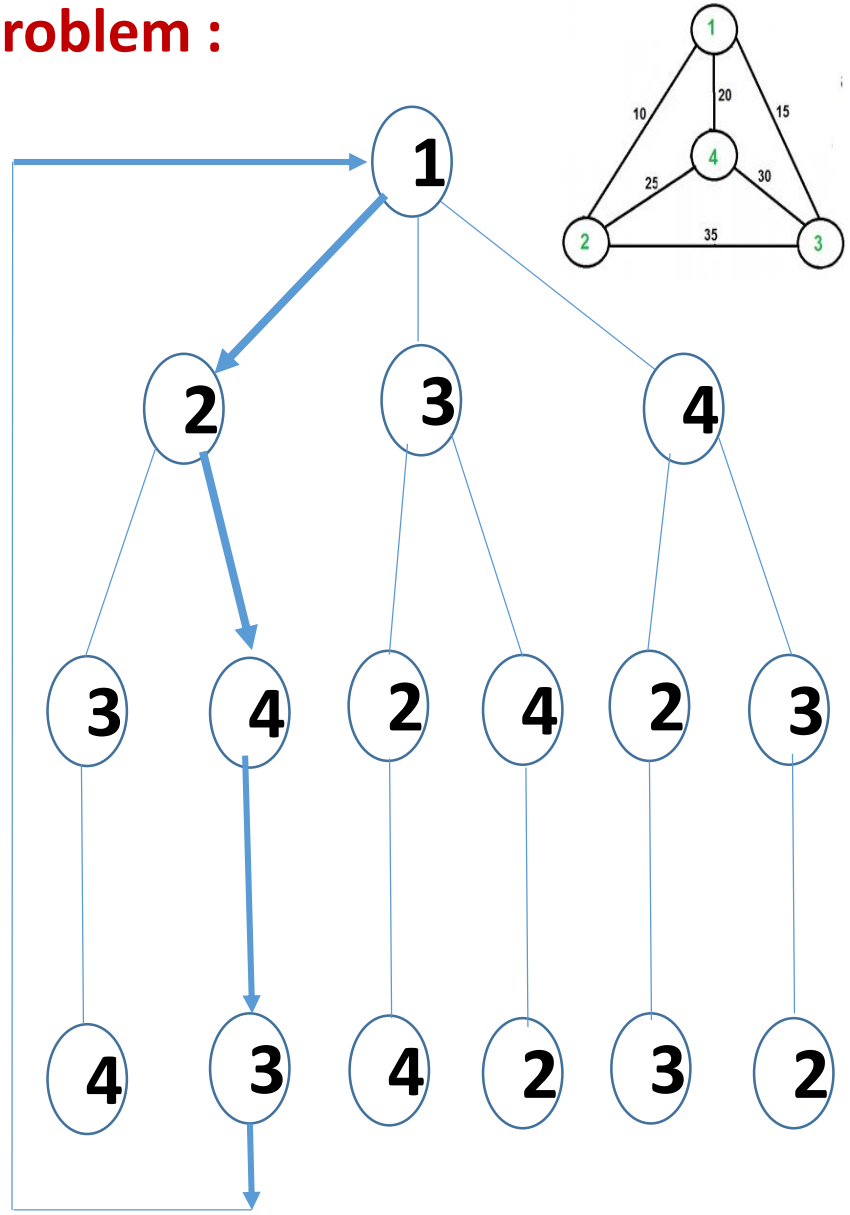
Brute force method / Naive Solution:

1) Consider city 1 as the starting and ending point.
2) Generate all (n-1)! Permutations of cities.
3) Calculate cost of every permutation and keep track of minimum cost permutation.
4) Return the permutation with minimum cost.

Time Complexity: Θ(n!)



Dynamic Programming Method for solving traveling salesman problem :



Let the given set of vertices be {1, 2, 3, 4,….n}.

Let us consider 1 as starting and ending point of output.

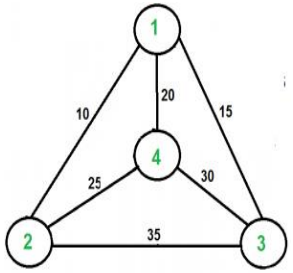
For every other vertex i (other than 1),

find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once.

Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1.

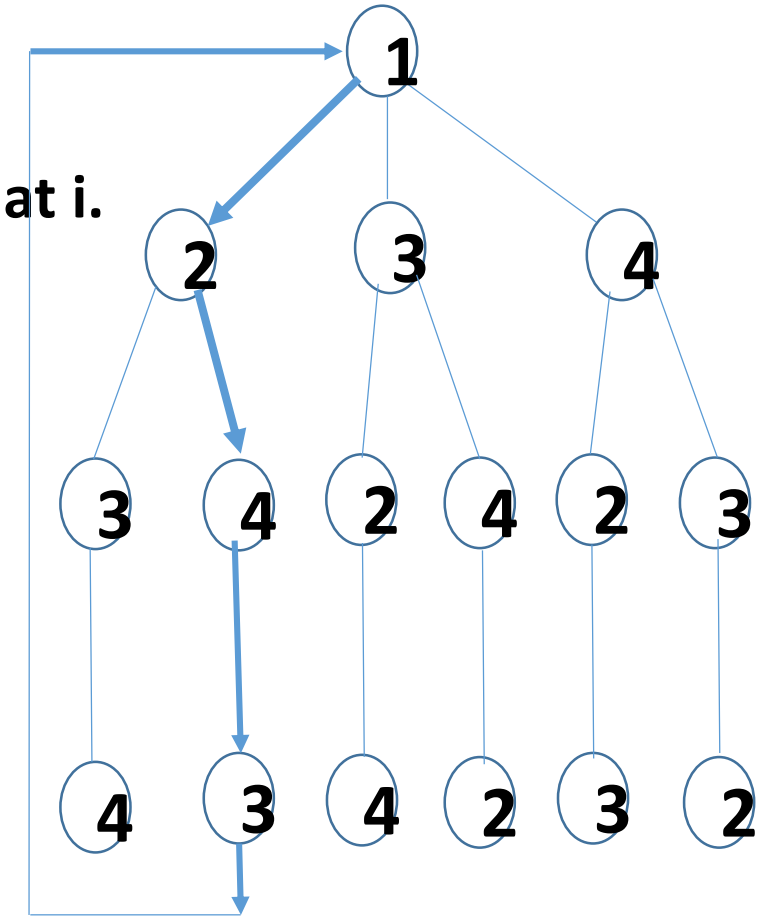
Finally, we return the minimum of all [cost(i) + dist(i, 1)] values.

Finding cost(i) :



To calculate cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems.

Let us define a term C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i.



We start with all subsets of size 2 and
calculate C(S, i) for all subsets where S is the subset, then
we calculate C(S, i) for all subsets S of size 3 and so on.
Note that 1 must be present in every subset.

If size of S is 2, then S must be {1, i},
C(S, i) = dist(1, i)
Else if size of S is greater than 2.
C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.

For a set of size n, we consider n-2 subsets each of size n-1 such that all subsets don't have nth in them.

Using the above recurrence relation, we can write dynamic programming based solution.

There are at most O(n*2n) sub problems, and each one takes linear time to solve.

The total running time is therefore O(n2*2n).

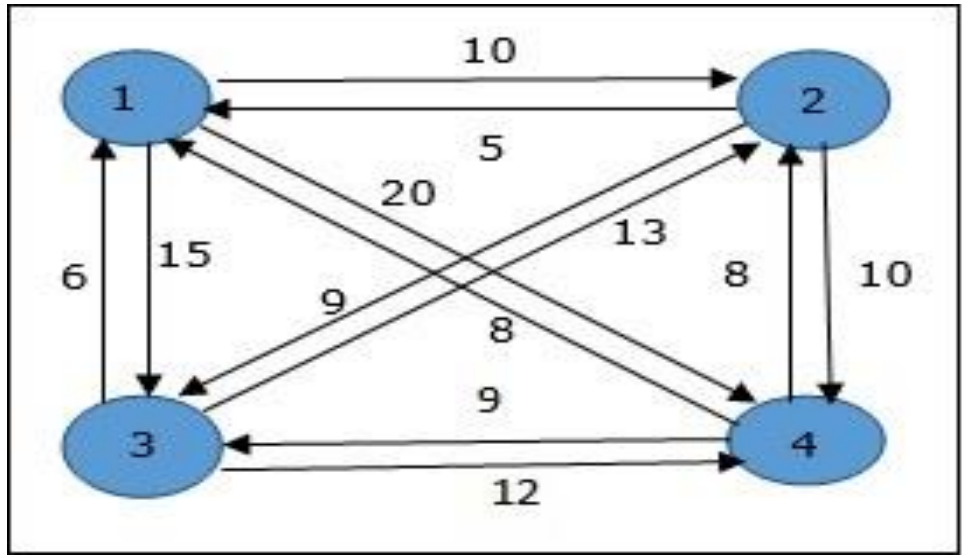
The time complexity is much less than O(n!), but still exponential.

Space required is also exponential.

So this approach is also infeasible even for slightly higher number of vertices.

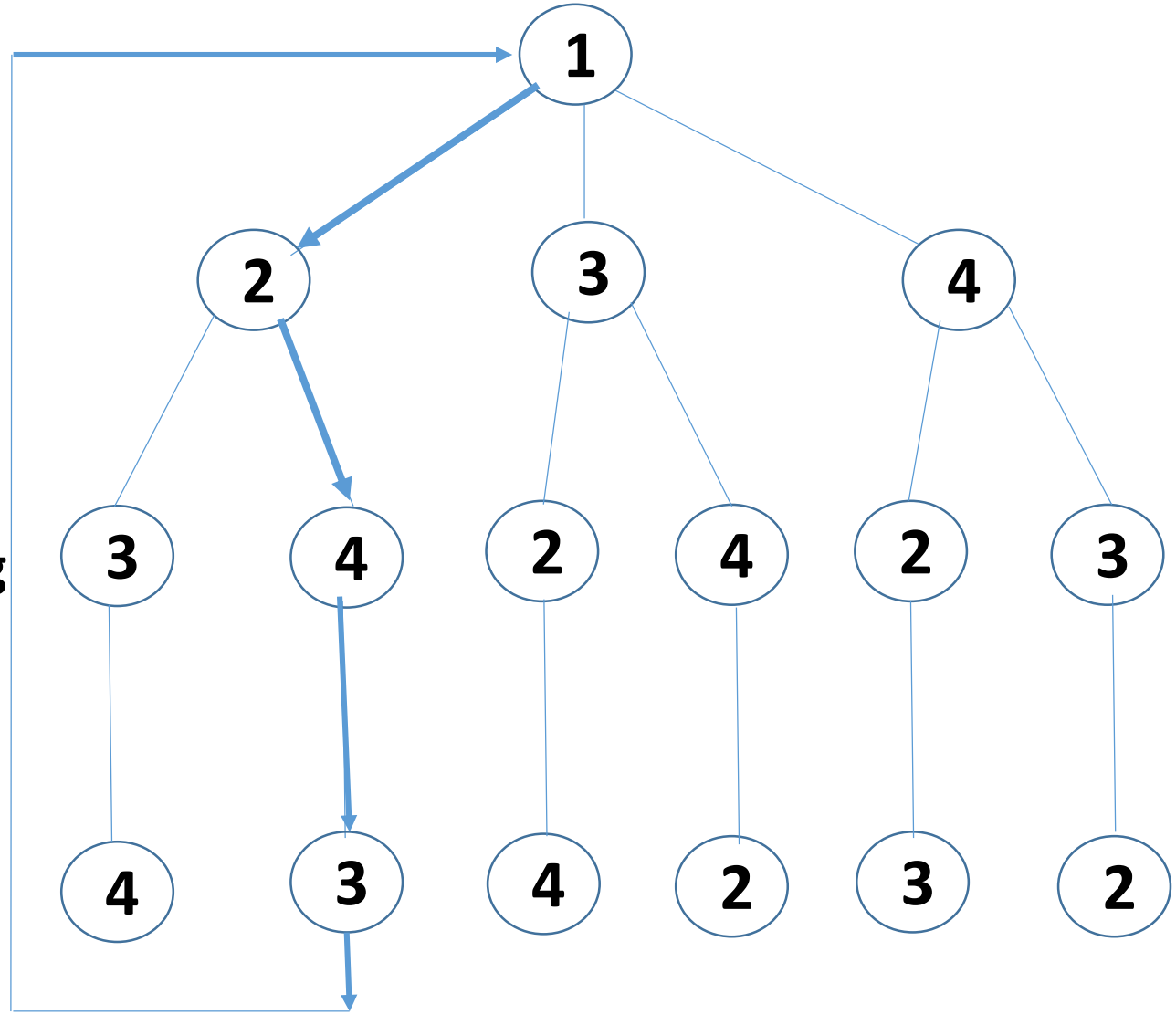
So, we may use some approximate algorithms for travelling salesman problem…….

Q2. Find the path of Travelling salesman problem for the given graph. (10 M)SH18



Step1: From the above graph, the following matrix is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

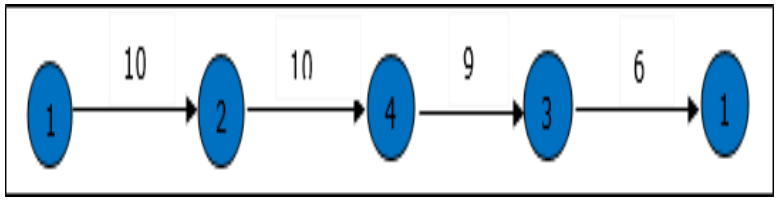


S = Φ

Cost(2,Φ,1)=d(2,1)=5

Cost(3,Φ,1)=d(3,1)=6

Cost(4,Φ,1)=d(4,1)=8



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

If size of S is 2, then S must be {1, i},
C(S, i) = dist(1, i)
Else if size of S is greater than 2.
C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.

S=1

Cost(i,s)=min{Cost(j,s–(j))+d[i,j]}

Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1)=9+6=15

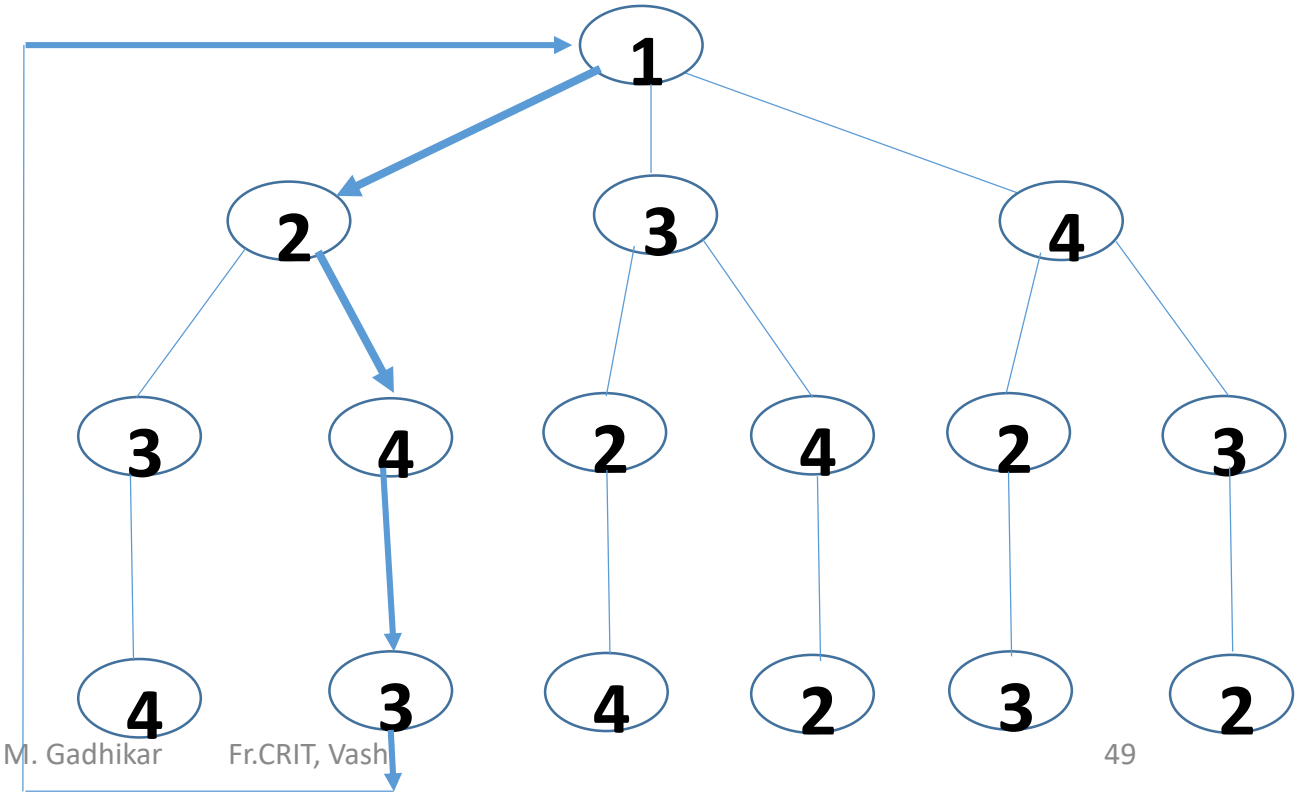
Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=18

Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18

Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20

Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13

Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15



S = 2

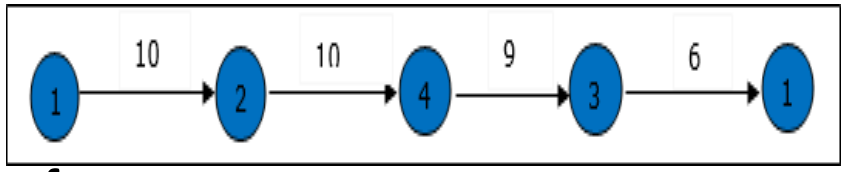
Cost(2,{3,4},1) = MIN {

d[2,3]+Cost(3,{4},1)=9+20=29 ,

d[2,4]+Cost(4,{3},1)=10+15=25

}

Cost(2,{3,4},1) = 25



Cost(3,{2,4},1) = MIN {

d[3,2]+Cost(2,{4},1)=13+18=31 ,

d[3,4]+Cost(4,{2},1)=12+13=25

}

Cost(3,{2,4},1) = 25

Cost(4,{2,3},1) = MIN {

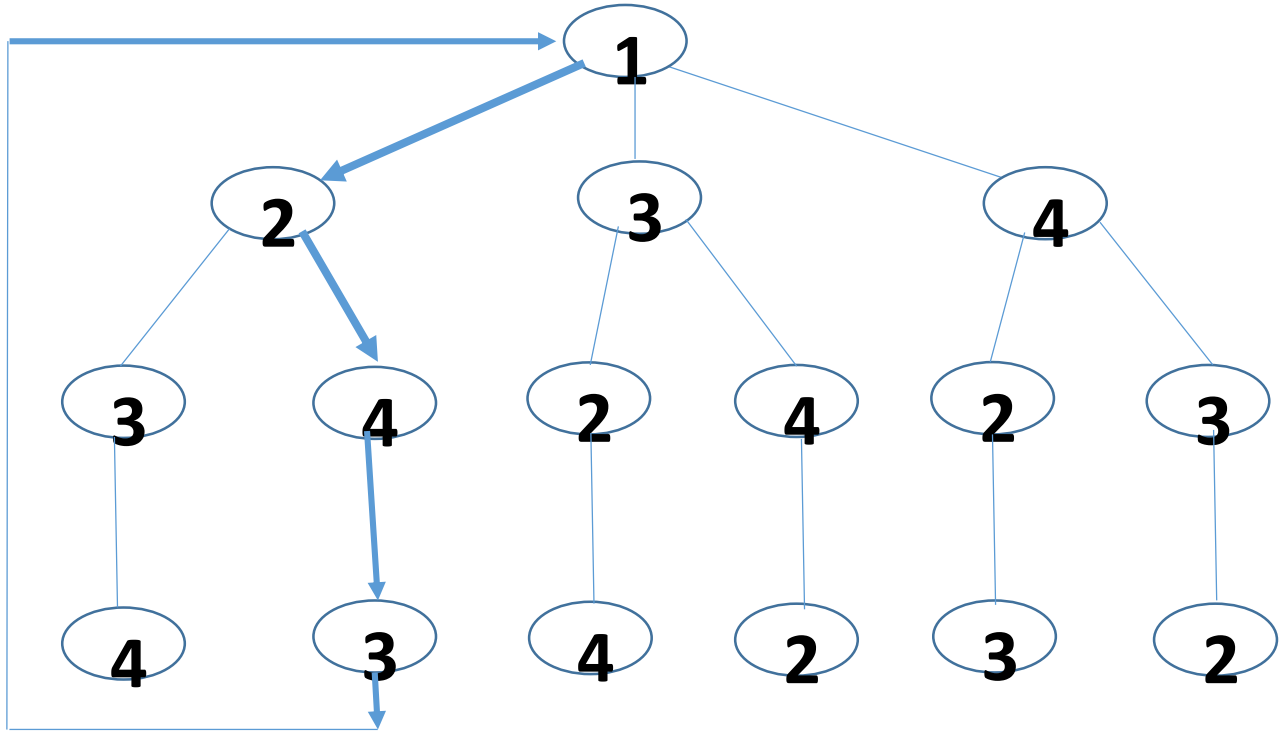
d[4,2]+Cost(2,{3},1)=8+15=23 ,

d[4,3]+Cost(3,{2},1)=9+18=27

}

Cost(4,{2,3},1) = 23

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0



S = 2 ➔

Cost(2,{3,4},1) = 25

Cost(3,{2,4},1) = 25

Cost(4,{2,3},1) = 23

S = 3

Cost(1, {2,3,4}, 1) = MIN {

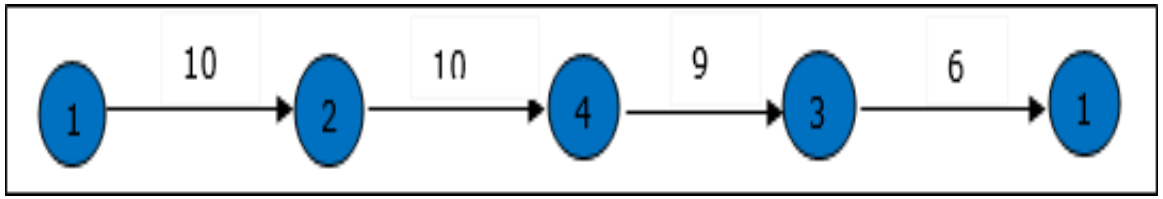
d[1,2]+Cost(2,{3,4},1)=10+25=35 ,

d[1,3]+Cost(3,{2,4},1)=15+25=40 ,

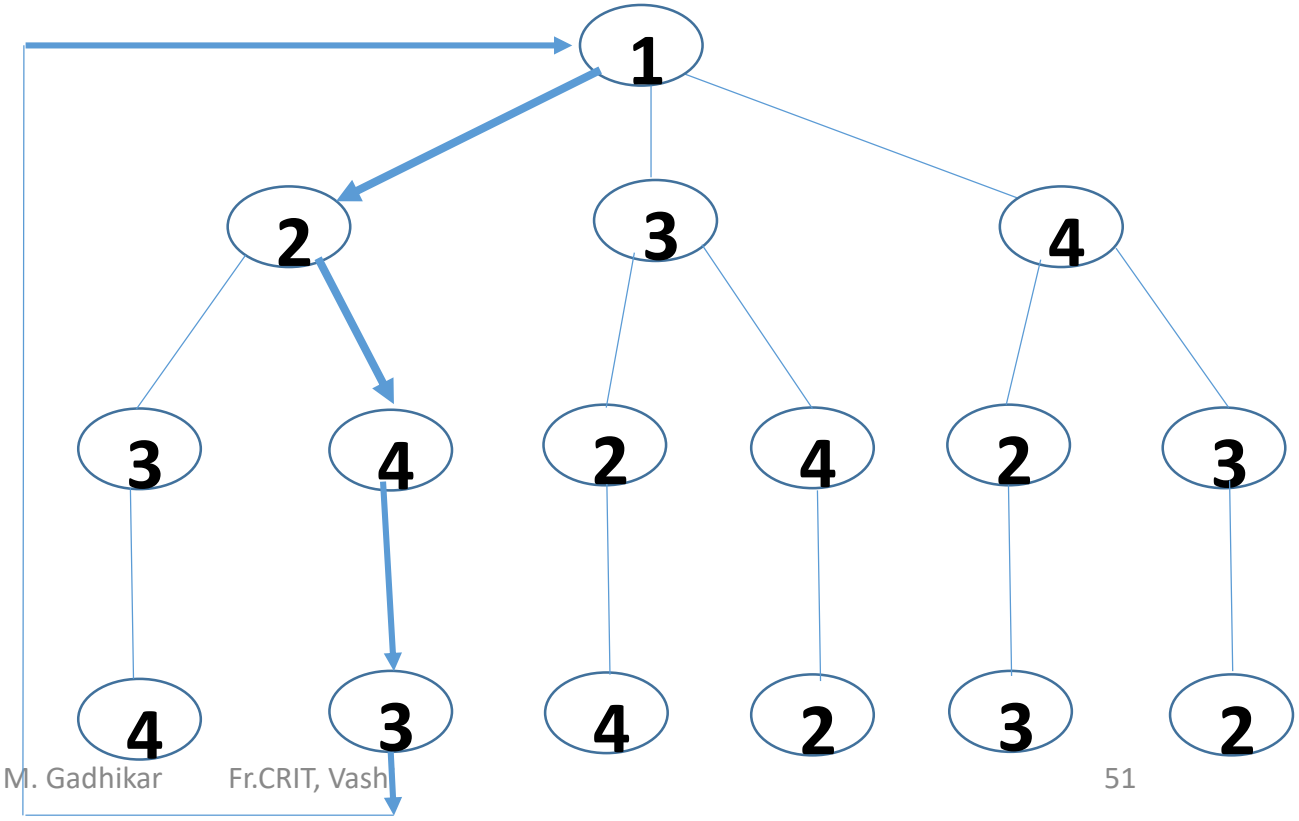
d[1,4]+Cost(4,{2,3},1)=20+23=43

}

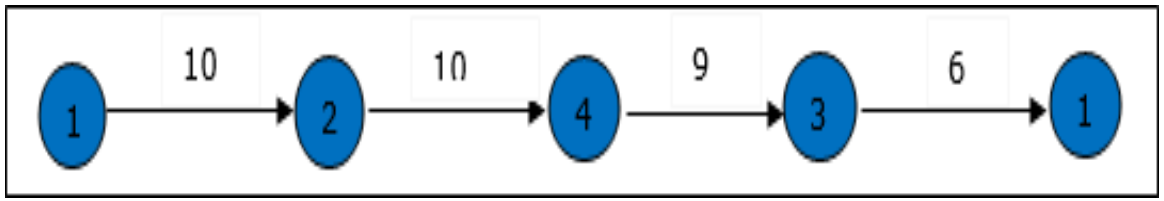
Min Cost(1,{2,3,4},1) = 35, Min cost Path =



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0



The minimum cost path is 35.

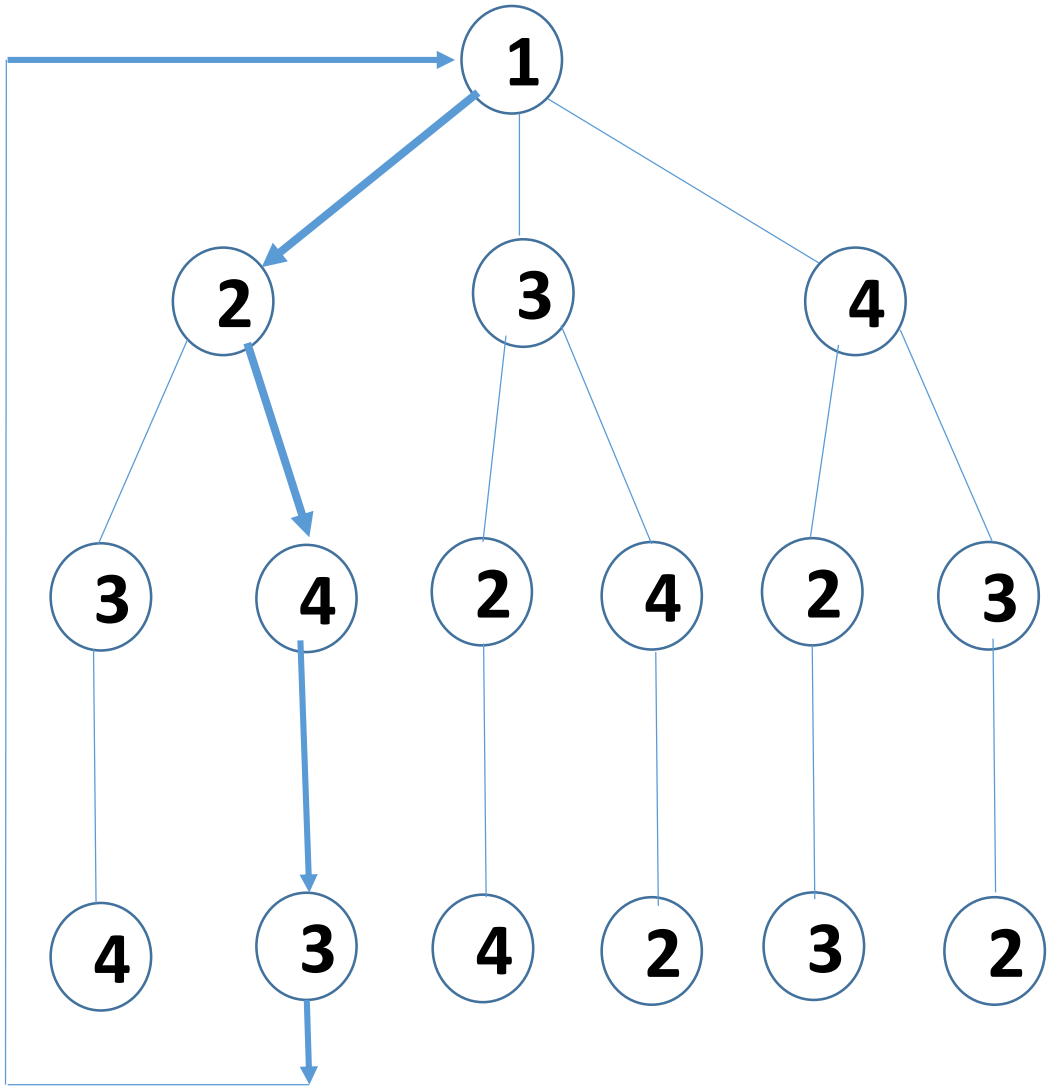


Start from cost {1, {2, 3, 4}, 1}, we get the minimum value for d [1, 2].

When s = 3, select the path from 1 to 2 (cost is 10) then go backwards.

When s = 2, we get the minimum value for d [4, 2]. Select the path from 2 to 4 (cost is 10) then go backwards.

When s = 1, we get the minimum value for d [4, 3]. Selecting path 4 to 3 (cost is 9), then we shall go to 3 and then go to s = Φ step.
We get the minimum value for d [3, 1] (cost is 6).



Matrix Chain Multiplication :

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative.
In other words, no matter how we parenthesize the product, the result will be the same.
For example, if we had four matrices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ...

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency.
For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations
A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.
In both cases result matrix is 10*60.

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Clearly the first parenthesization requires less number of operations.

Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i].

We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.

Input: p[] = {40, 20, 30, 10, 30} Output: 26000

There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.

Let the input 4 matrices be A, B, C and D.

The minimum number of multiplications are obtained by putting parenthesis in following way

(A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30

Input: p[] = {10, 20, 30, 40, 30} Output: 30000

There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.

Let the input 4 matrices be A, B, C and D.

The minimum number of multiplications are obtained by putting parenthesis in following way

((AB)C)D --> 10*20*30 + 10*30*40 + 10*40*30

```
/* A naive recursive implementation that simply
  follows the above optimal substructure property */
#include<stdio.h>
#include<limits.h>

// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int i, int j)
{
   if(i == j)
      return 0;
   int k;
   int min = INT_MAX;
   int count;

   // place parenthesis at different places between first
   // and last matrix, recursively calculate count of
   // multiplications for each parenthesis placement and
   // return the minimum count
   for (k = i; k <j; k++)
     {
        count = MatrixChainOrder(p, i, k)
        +  MatrixChainOrder(p, k+1, j) +
              p[i-1]*p[k]*p[j];

        if (count < min)
           min = count;
     }
     // Return minimum count
     return min;
}
```

Input: p[] = {40, 20, 30, 10, 30}

40x20, 20x30, 30x10 and 10x30.

A, B, C and D.

(A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30

```
// Driver program to test above function
int main()
{
   int arr[] = {1, 2, 3, 4, 3};
   int n = sizeof(arr)/sizeof(arr[0]);

   printf("Minimum number of multiplications
     is %d ", MatrixChainOrder(arr, 1, n-1));

   getchar();
   return 0;
}
```

Time complexity of the above naive recursive approach is exponential.

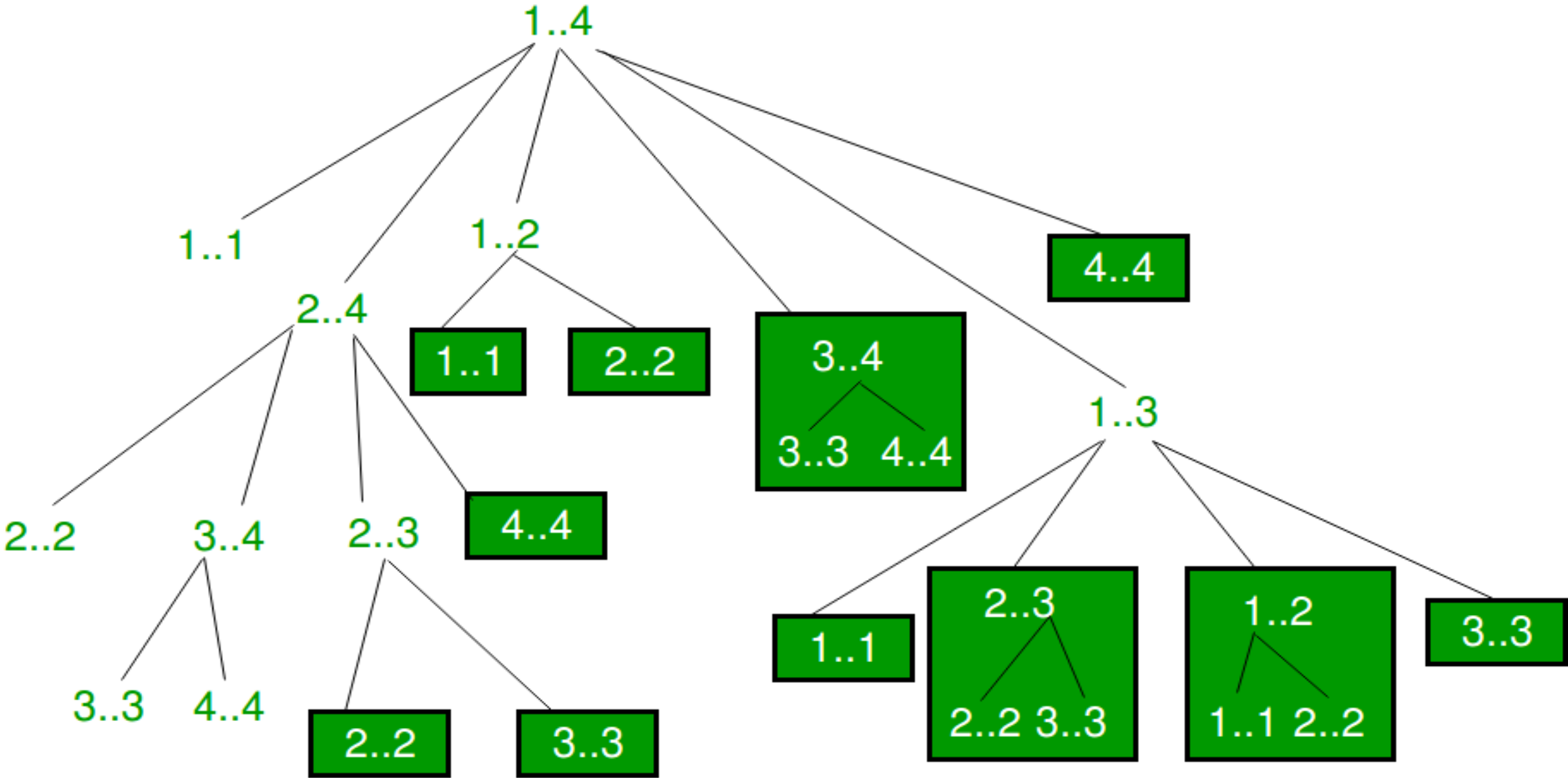
It should be noted that the above function computes the same sub problems again and again.

See the following recursion tree for a matrix chain of size 4.

The function MatrixChainOrder(p, 3, 4) is called two times.

We can see that there are many sub problems being called more than once.

2) Overlapping Subproblems :



Since same sub problems are called again, this problem has Overlapping Sub problems property.

So Matrix Chain Multiplication problem has both properties of a dynamic programming problem.

Like other typical Dynamic Programming(DP) problems, re-computations of same sub problems can be avoided by constructing a temporary array m[][] in bottom up manner.

Dynamic Programming Solution :

```
// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int n)
{
/* For simplicity of the program, one extra row
 and one extra column are allocated in m[][].
 0th row and 0th column of m[][] are not used */
   int m[n][n];
   int i, j, k, L, q;

/* m[i,j] = Minimum number of scalar multiplications
needed to compute matrix A[i]A[i+1]...A[j] = A[i..j]
where dimension of A[i] is p[i-1] x p[i] */

   // cost is zero when multiplying one matrix.
   for (i=1; i<n; i++)
      m[i][i] = 0;
// L is chain length.
   for (L=2; L<n; L++)
   {
      for (i=1; i<n-L+1; i++)
      {
         j = i+L-1;
         m[i][j] = INT_MAX;
         for (k=i; k<=j-1; k++)
         {
      // q = cost/scalar multiplications
            q = m[i][k] + m[k+1][j] +
                     p[i-1]*p[k]*p[j];
            if (q < m[i][j])
               m[i][j] = q;
         }
      }
   }
     return m[1][n-1];
}
```

```
int main()
{
   int arr[] = {1, 2, 3, 4};
   int size = sizeof(arr)/sizeof(arr[0]);

   printf("Minimum number of multiplications is %d ",
                  MatrixChainOrder(arr, size));

   getchar();
   return 0;
}
```

Time Complexity: $O(n^3)$

Auxiliary Space: O(n^2)

m	1	2	3	4
1	0	120		
2		0	48	
3			0	84
4				0

S	1	2	3	4
1		1		
2			2	
3				3
4				

A1 . A2 . A3 . A4

5*4 4*6 6*2 2*7

1. Multiplying matrices in pairs of two :

3 possible combinations :

i. M[1,2]

A1 . A2 So, cost of multiplication of A1*A2 is :

5*4 4*6 = 5 * 4 * 6 = 120

For M[1,2] i.e. A1.A2 , split is at A1 , so S[1,2] = 1

ii. M[2,3]

A2 . A3

4*6 6*2 = 4 * 6 * 2 = 48

For M[2,3] i.e. A2.A3 , split is at A2 , so S[2,3] = 2

iii. M[3,4]

A3 . A4

6*2 2*7 = 6 * 2 * 7 = 84

For M[3,4] i.e. A3.A4 , split is at A3 , so S[3,4] = 3

m	1	2	3	4
1	0	120	88	
2		0	48	
3			0	84
4				0

S	1	2	3	4
1		1		
2			2	
3				3
4				

A1 . A2 . A3 . A4

5*4 4*6 6*2 2*7

2. Multiplying matrices in combinations of three: M[1,3] → A1 . A2 . A3 OR

M[2,4] → A2 . A3 . A4

M[1,3] → A1.A2.A3 :2 possible combinations:

MIN { A1 . (A2 . A3) or (A1 . A2) . A3 }

So, cost of multiplication A1 . (A2 . A3) is :

5 * 4 . 4 * 6 . 6 * 2

m[1,1] + m[2,3] + 5 *4 *2

0 + 48 + 40 = 88 = q

cost of multiplication (A1 . A2) . A3 is :

5 * 4 . 4 * 6 . 6 * 2

m[1,2] + m[3,3] + 5 *6 *2

120+ 0 + 60 = 180 = q

M[1,3] = min (q)

MIN { 88, 180 } = 88 given by A1 : A1 . (A2 . A3)

So, we write 1 in s[1,3]

M[2,4] : A2 . A3 . A4 Two possibilities :

A2 . (A3 . A4)	or	(A2 . A3) . A4
4 * 6 6 * 2 2 * 7		4 * 6 6 * 2 2 * 7
M[2,2] + m[3,4] + 4 * 6 * 7		m[2,3] + m[4,4] + 4 * 2 * 7
0 + 84 + 168		48 + 0 + 56
MIN { 252	,	104 } ➔

M[2,4] = 104 ➔

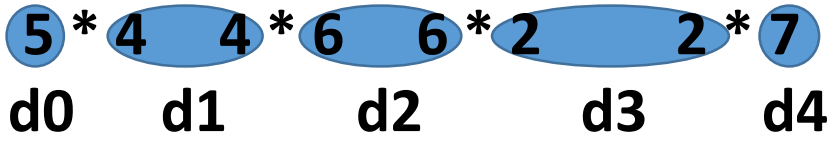
Min is from ({A2 * A3) * A4 = >

Min value has come from A3 i.e split is at A3 ➔ S[2,4] = 3.

m	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

S	1	2	3	4
1		1	1	
2			2	3
3				3
4				

3. Multiplying matrices in combinations of four: A1 . A2 . A 3 . A4



M[1,4] = A1 . (A2 . A3 . A4) or (A1 . A2) . (A3 . A4) or (A1 . A2 . A3) . A4 or A1 . (A2 . A3) . A4

MIN { m[1,1] + m[2,4] + 5 * 4 * 7 , m[1,2] + m [3,4] + 5 * 6 * 7 , m [1,3] + m [4,4] + 5 * 2 * 7,
m[1,1] + m [2 ,3] + m [4, 4] + 5*4*2 + 5*2*7 }

MIN { 0 + 104 + 140 , 120 + 84 + 210 , 88 + 0 + 70 , 0 +48 +0 + 40 + 70 }

= MIN {244, 414, 158, 158} ➔

m	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

S	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

Formula : for (k=i; k<=j-1; k++)

M[i , j] = min { m [i , k] + m [k+1 , j] + di-1 * dk * dj }

M[1, 4] = min { m [1,1] + m[2, 4] + d0 * d1 * d4 = 0 + 104 + 5*4*7

m	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

S	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

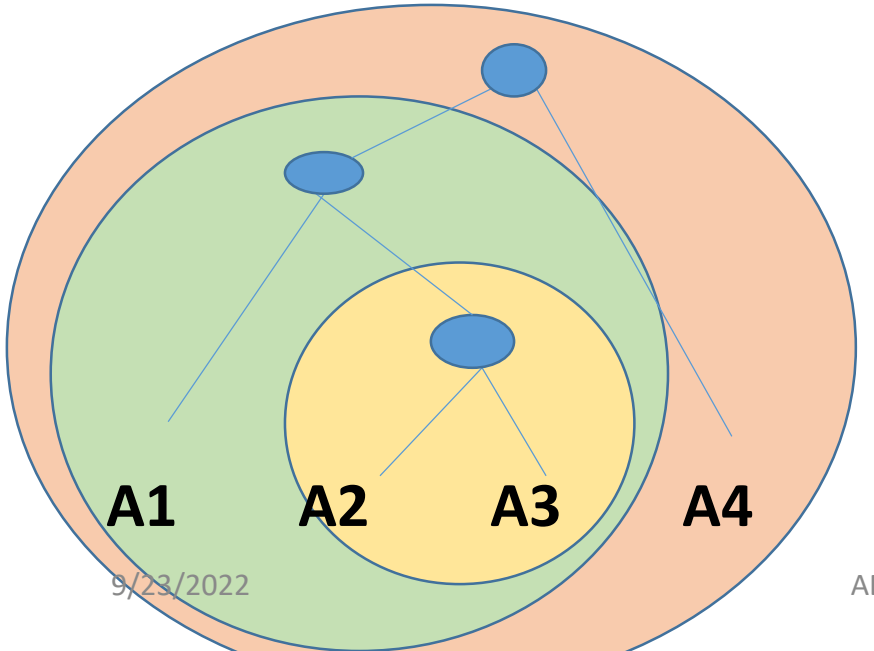
Observe table S, multiplication of m[1, 4] is split at 3 ➔
(A1 . A2 . A3) . (A4)

From table S, it is clear that A[1 , 3] is split at 1 ➔

((A1) . (A2 . A3))

(A1 . A2 . A3) . (A4)
((A1) . (A2 . A3)) . (A4)

https://www.javatpoint.com/matrix-chain-multiplication-example



Thus order of Matrix Chain Multiplication is :

((A1) . (A2 . A3)) . (A4)

Cost of above multiplication = 158 comparisons.

Dynamic Programming Solution :

```
// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int n)
{
/* For simplicity of the program, one extra row
 and one extra column are allocated in m[][].
 0th row and 0th column of m[][] are not used */
   int m[n][n];
   int i, j, k, L, q;

/* m[i,j] = Minimum number of scalar multiplications
needed to compute matrix A[i]A[i+1]...A[j] = A[i..j]
where dimension of A[i] is p[i-1] x p[i] */

   // cost is zero when multiplying one matrix.
   for (i=1; i<n; i++)
      m[i][i] = 0;
// L is chain length.
   for (L=2; L<n; L++)
   {
      for (i=1; i<n-L+1; i++)
      {
         j = i+L-1;
         m[i][j] = INT_MAX;
         for (k=i; k<=j-1; k++)
         {
      // q = cost/scalar multiplications
            q = m[i][k] + m[k+1][j] +
                     p[i-1]*p[k]*p[j];
            if (q < m[i][j])
               m[i][j] = q;
         }
      }
   }
     return m[1][n-1];
}
```

```
int main()
{
   int arr[] = {1, 2, 3, 4};
   int size = sizeof(arr)/sizeof(arr[0]);

   printf("Minimum number of multiplications is %d ",
                  MatrixChainOrder(arr, size));

   getchar();
   return 0;
}
```

Time Complexity: $O(n^3)$

Auxiliary Space: O(n^2)

Optimal Binary Search Tree (OBST):

Q. What is Binary Search Tree? How to generate Optimal Binary Search Tree? (10M)

Q. Write a note on OBST. (5M OR 10M)

Given a sorted array keys[0.. n-1] of search keys and an array freq[0.. n-1] of frequency counts, where freq[i] is the number of searches to keys[i].

Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.

Let us first define the cost of a BST.

The cost of a BST node is level of that node multiplied by its frequency. Level of root is 1.

Example 1 :

Input: keys[] = {10, 12}, freq[] = {34, 50}

There can be following two possible BSTs

Frequency of searches of 10 and 12 are 34 and 50 respectively.

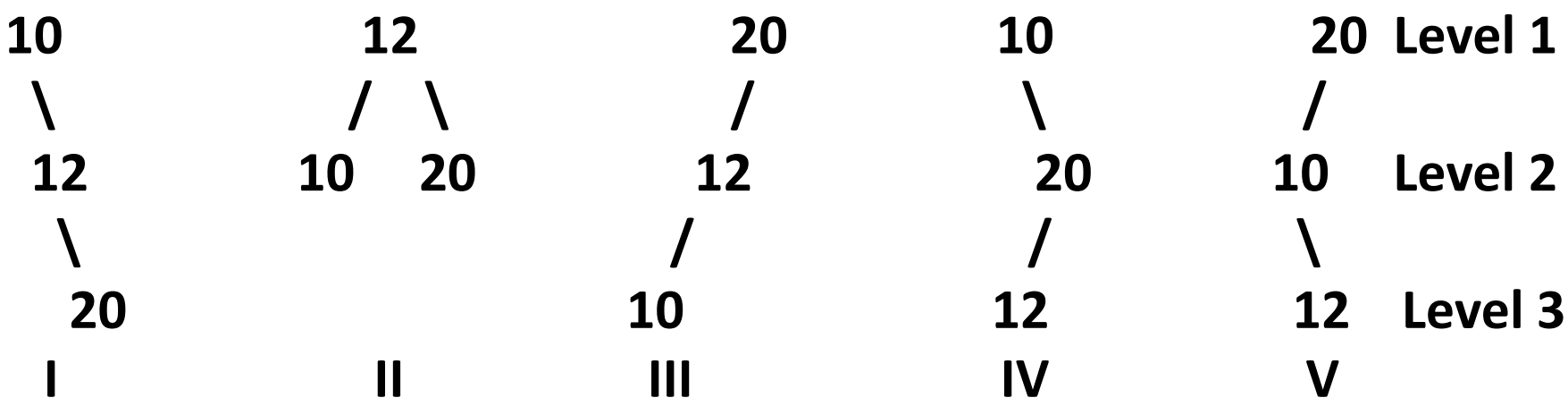
The cost of tree I is 34*1 + 50*2 = 134

The cost of tree II is 50*1 + 34*2 = 118

Example 2 :

Input: keys[] = {10, 12, 20} , freq[] = {34, 8, 50}

There can be following possible BSTs



Among all possible BSTs, cost of the fifth BST is minimum.

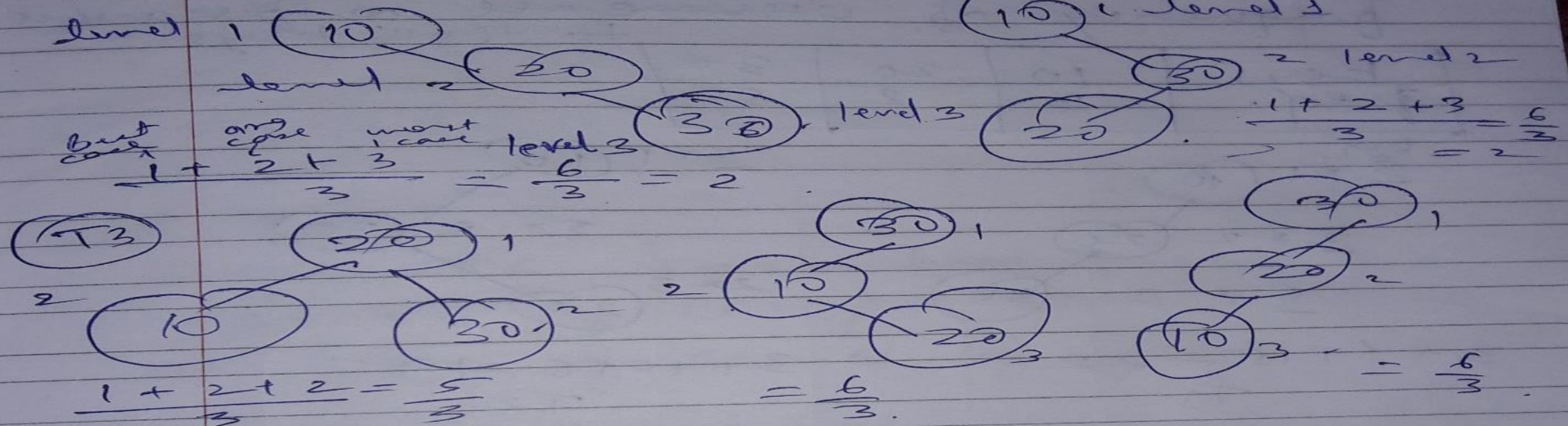
Cost of the fifth BST is 1*50 + 2*34 + 3*8 = 142

Src: Abdul Bari

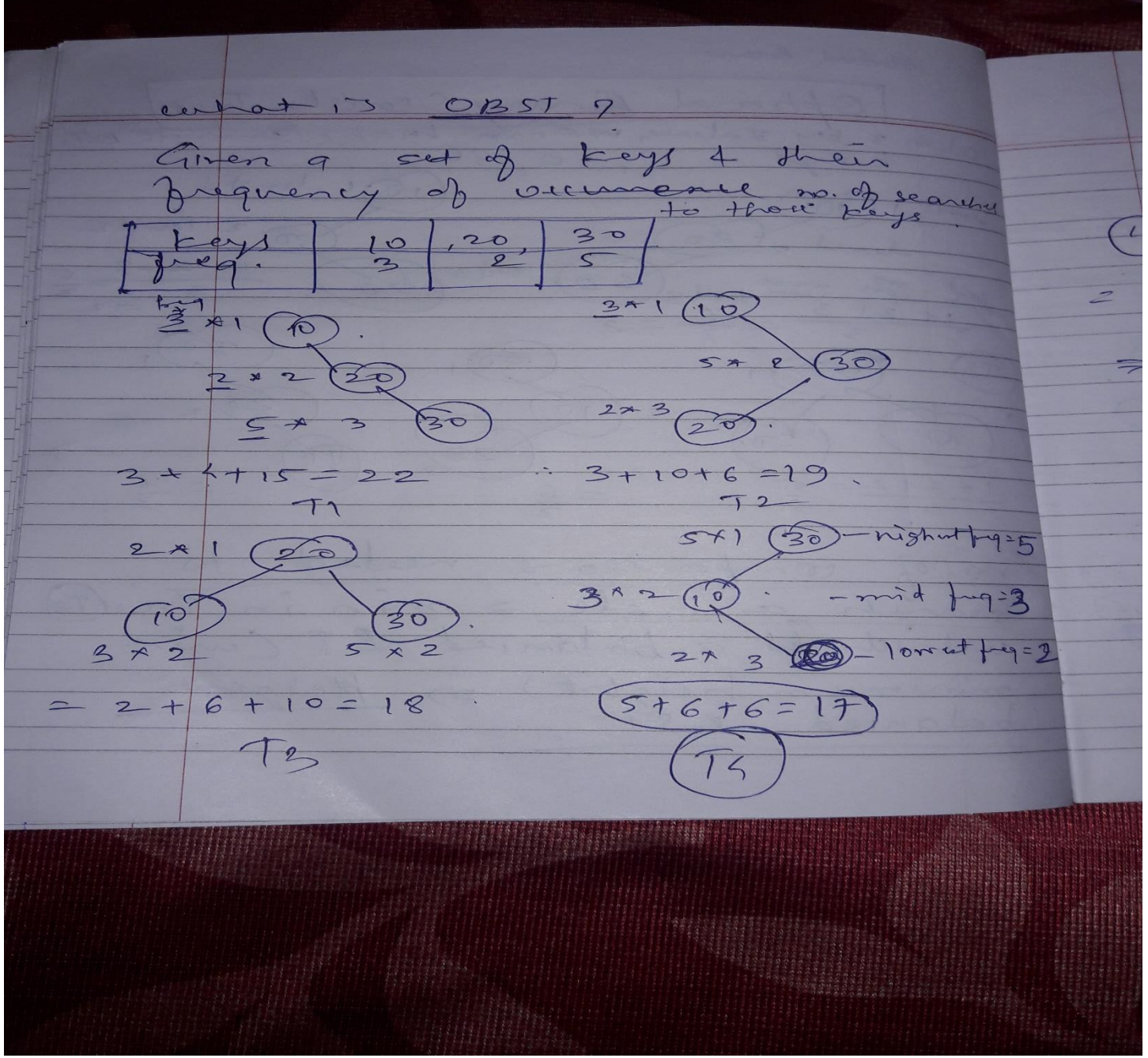
Optimal Binary Search Tree

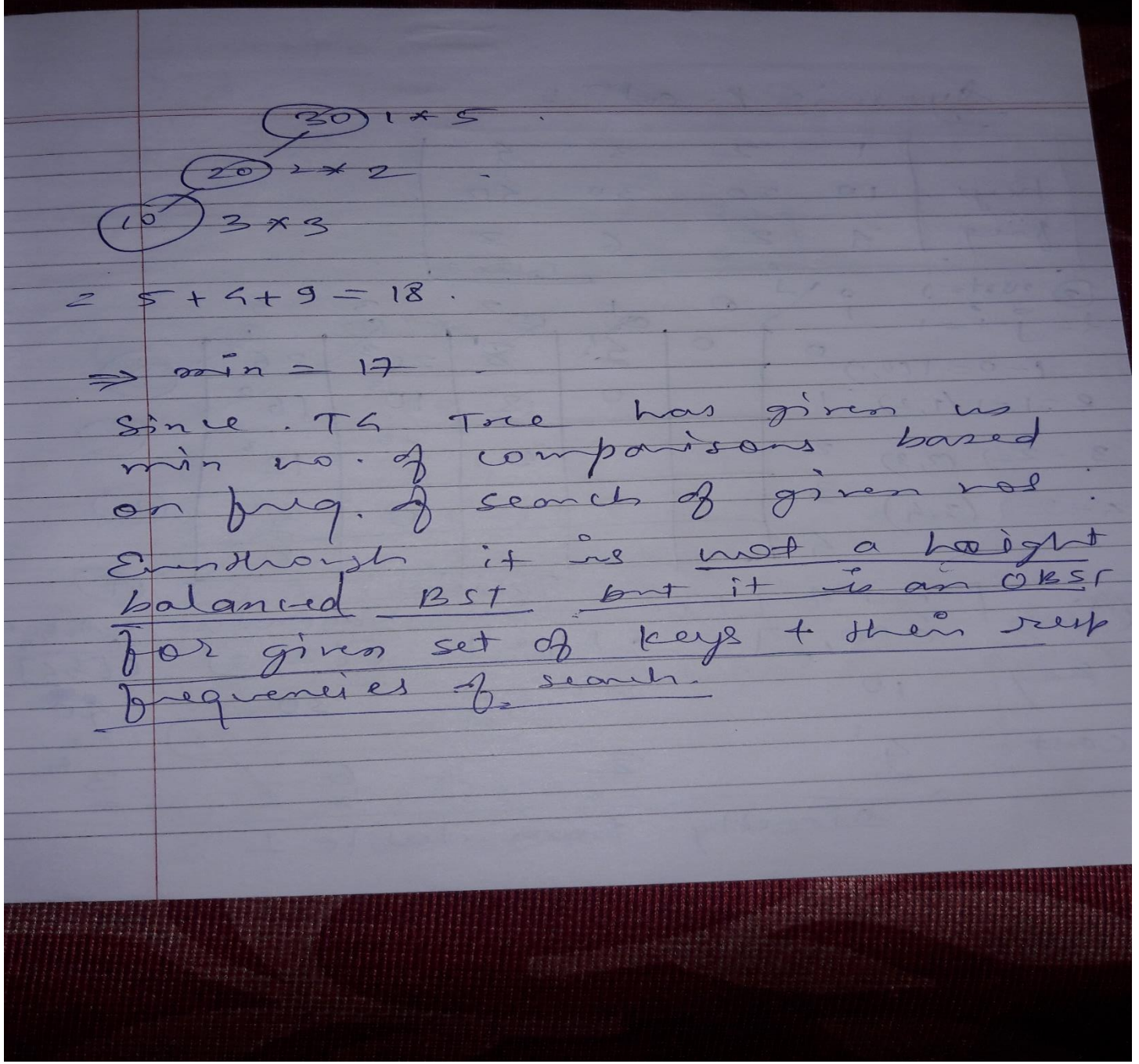
Consider 3 key values, we can create/generate following BSTs

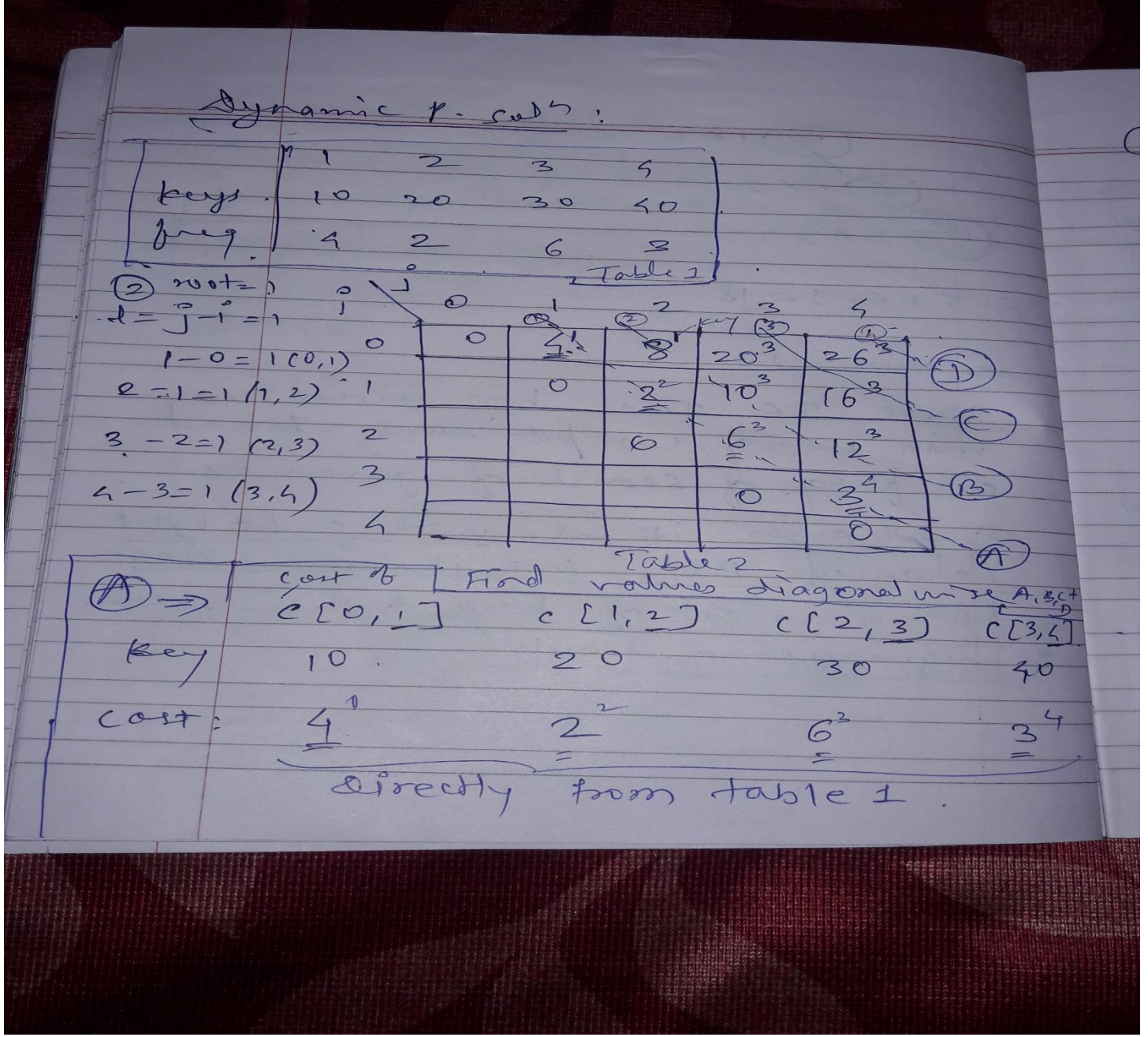
Keys : 10, 20, 30

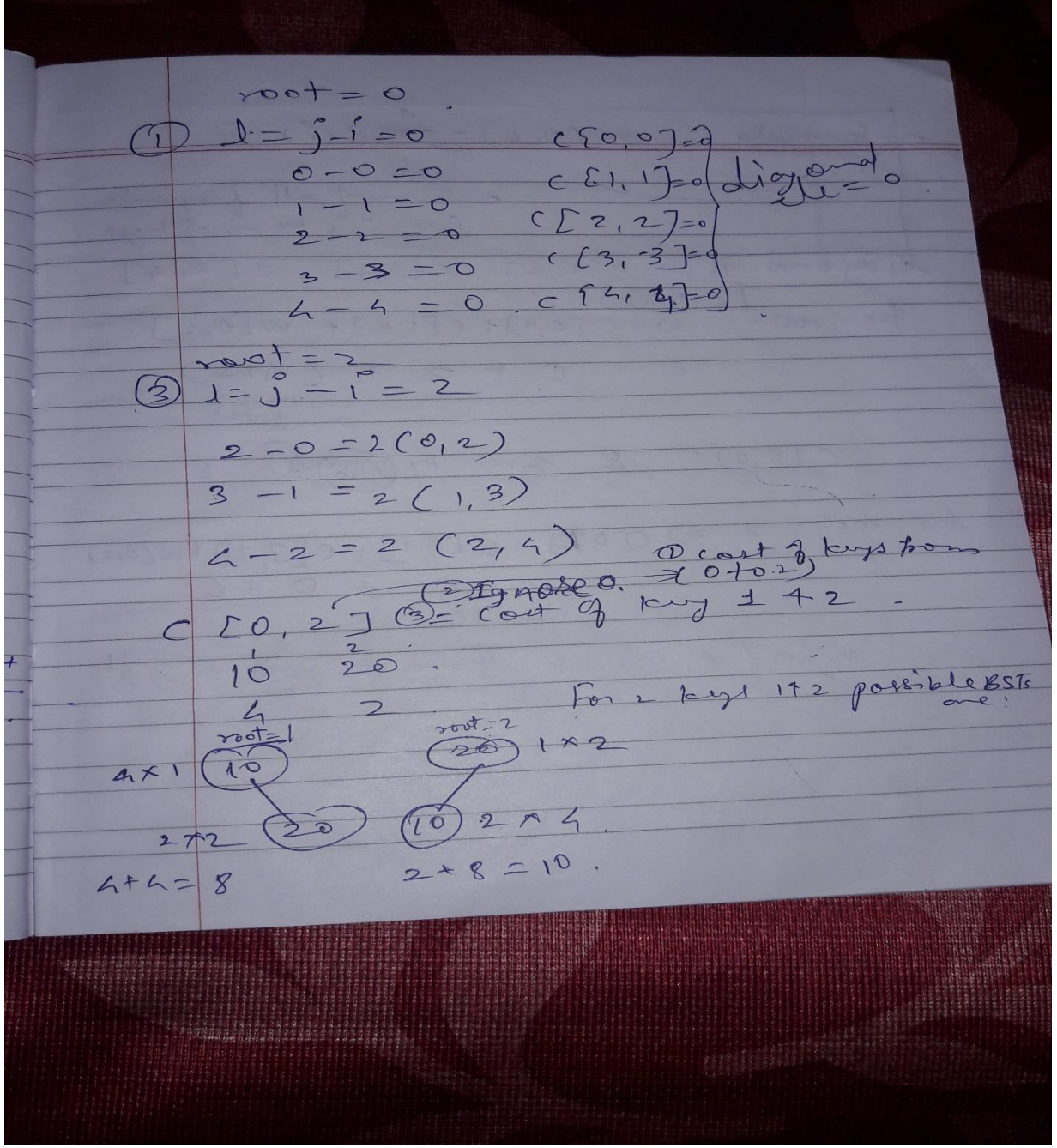


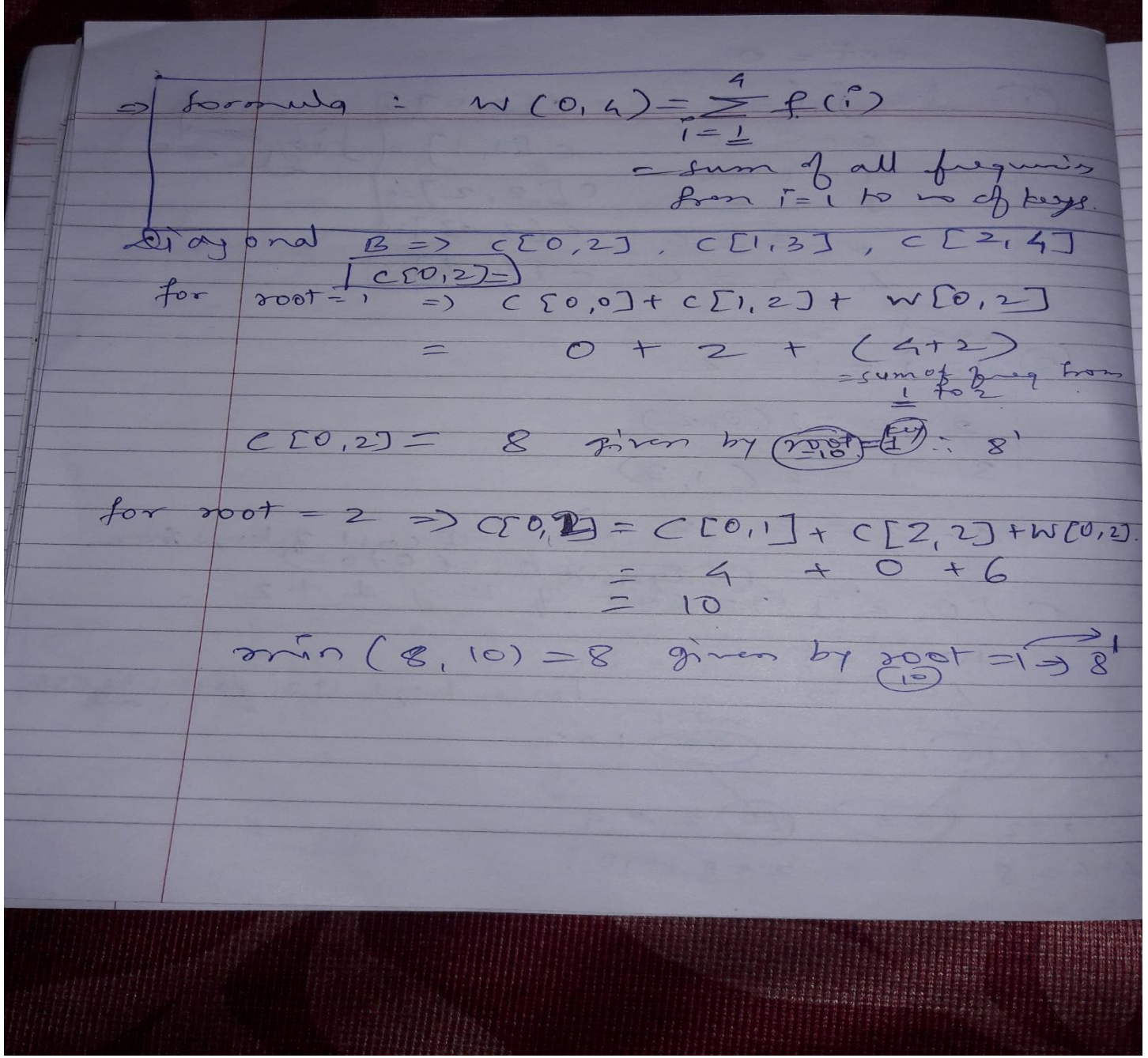
Avg no. of comparisons needed to search an ele = min in Tree (T3) which is a balanced BT (with minimum height) => Height balanced BST.


what is OBST ?
Given a set of keys & their frequency of occurrence i.e. no. of searches to their keys.
Keys | 10 | 20 | 30
freq. | 3 | 2 | 5
3 * 1
2 * 2
5 * 3
3 + 4 + 15 = 22
T1
2 * 1
3 * 2
5 * 2
= 2 + 6 + 10 = 18
T3
3 * 1
5 * 2
2 * 3
∴ 3 + 10 + 6 = 19
T2
5 * 1
3 * 2
2 * 3
- right freq = 5
- mid freq = 3
- lower freq = 2
5 + 6 + 6 = 17
T4


30
20
10
1*5
2*2
3*3
= 5 + 4 + 9 = 18.
⇒ min = 17
Since . T4 Tree has given us min no. of comparisons based on freq. of search of given nos.
Even though it is not a height balanced BST but it is an OBST for given set of keys & their resp frequencies & search.


Dynamic p. coding:
keys 10 20 30 40
freq. 4 2 6 3
Table 1
② root = i
d = j - i = 1
1 - 0 = 1 (0,1)
2 - 1 = 1 (1,2)
3 - 2 = 1 (2,3)
4 - 3 = 1 (3,4)
Table 2
cost & Find values diagonal wise A, B, C, D
c[0,1] c[1,2] c[2,3] c[3,4]
Key 10 20 30 40
cost: 4^1 2^2 6^3 3^4
directly from table 1.

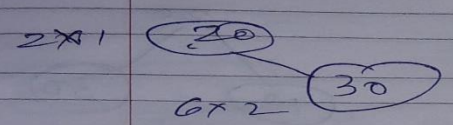

root = 0
① l = j - i = 0
0 - 0 = 0
1 - 1 = 0
2 - 2 = 0
3 - 3 = 0
4 - 4 = 0
c[0,0] = 0
c[1,1] = 0
c[2,2] = 0
c[3,3] = 0
c[4,4] = 0
diagonal = 0
root = 2
③ l = j - i = 2
2 - 0 = 2 (0,2)
3 - 1 = 2 (1,3)
4 - 2 = 2 (2,4)
② Ignore 0.
① cost of keys from 0 to 2
c[0,2] ③ = cost of key 1 + 2
10 20
4 2
For 2 keys 1+2 possible BSTs are!
root = 1
root = 2
4 x 1
1 x 2
2 x 2
2 x 4
4 + 4 = 8
2 + 8 = 10.


formula : $w(0,4) = \sum_{i=1}^{4} f(i)$
= sum of all freq's from i=1 to n of keys.
Diagonal B => c[0,2], c[1,3], c[2,4]
for root = 1 => c[0,0] + c[1,2] + w[0,2]
= 0 + 2 + (4+2)
= sum of freq from 1 to 2
c[0,2] = 8 given by root = 1 ∴ 8'
for root = 2 => c[0,2] = c[0,1] + c[2,2] + w[0,2]
= 4 + 0 + 6
= 10
min (8, 10) = 8 given by root = 1 ⇒ 8'

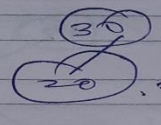
C[1, 3] Ignore 1 start from 2, so, 2+3.

key	2	3
key	20	30
freq	2	6

diff trees possible with this are.



= 2+12 = 14

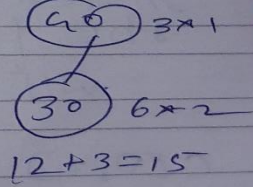


= 6+4 = 10

~~root~~ min (14, 10) = 10 ⇒ given by root = 30.
⇒ key 3
⇒ 10^3

C[2, 4] Ignore 2 + take 3 + 4.

	3	4
keys	30	40
freq	6	3

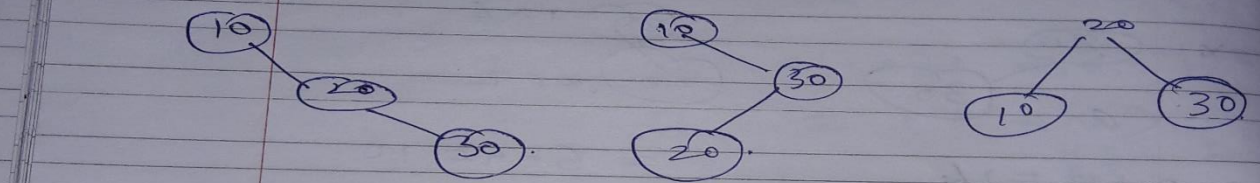


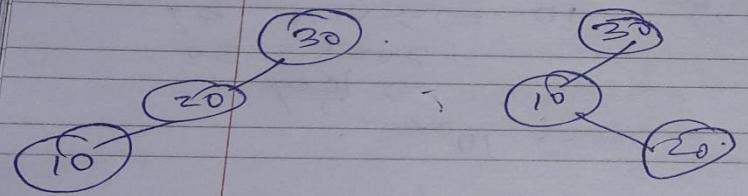
root result = 12 given by root = 30 = key 3
= 12^3

Diagonal ① ⇒ c[0,3] + c[1,4]

c[0,3] Ignore 0 take 1 to 3.

	1	2	3
Key	10	20	30
freq	4	2	6

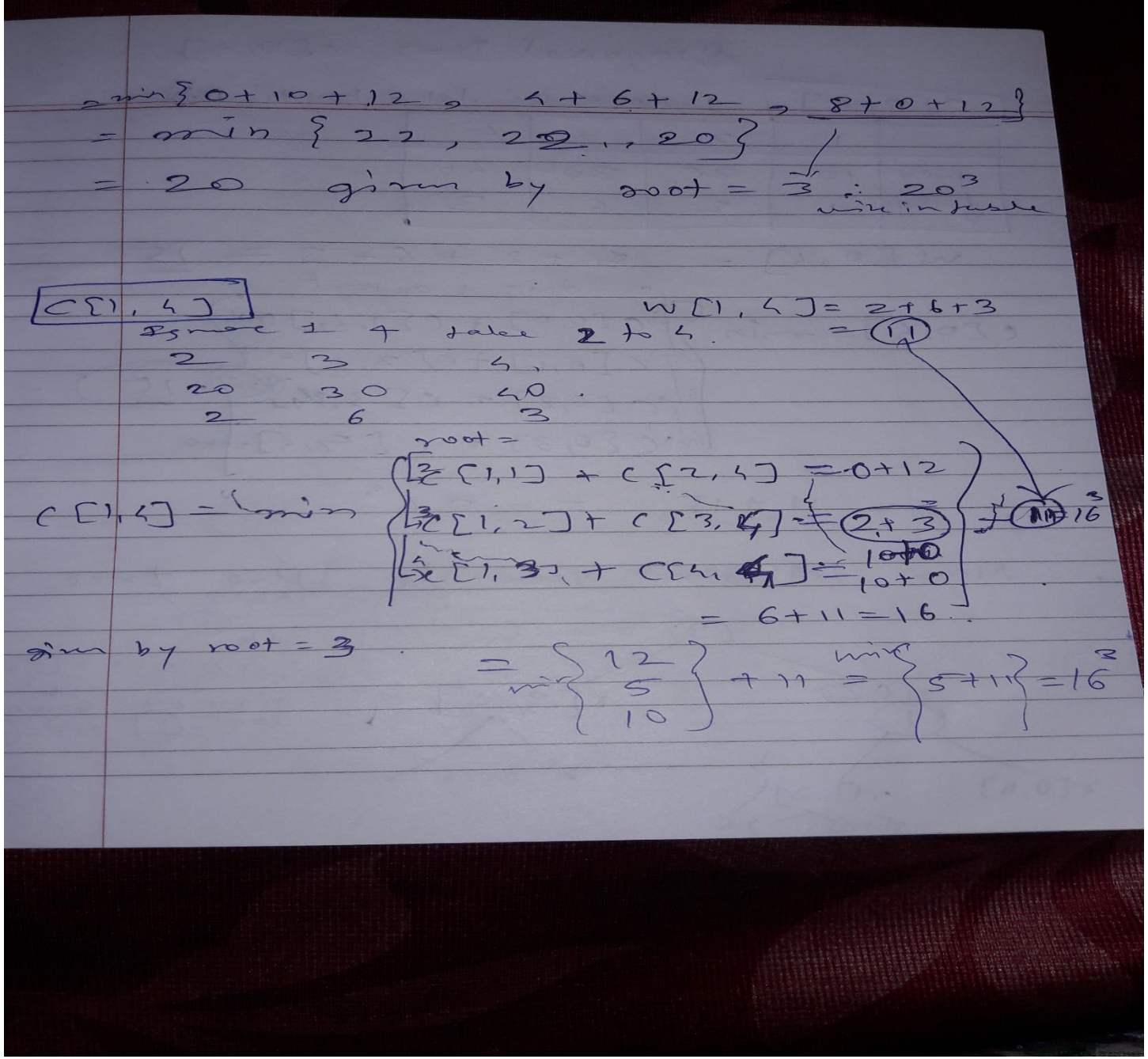


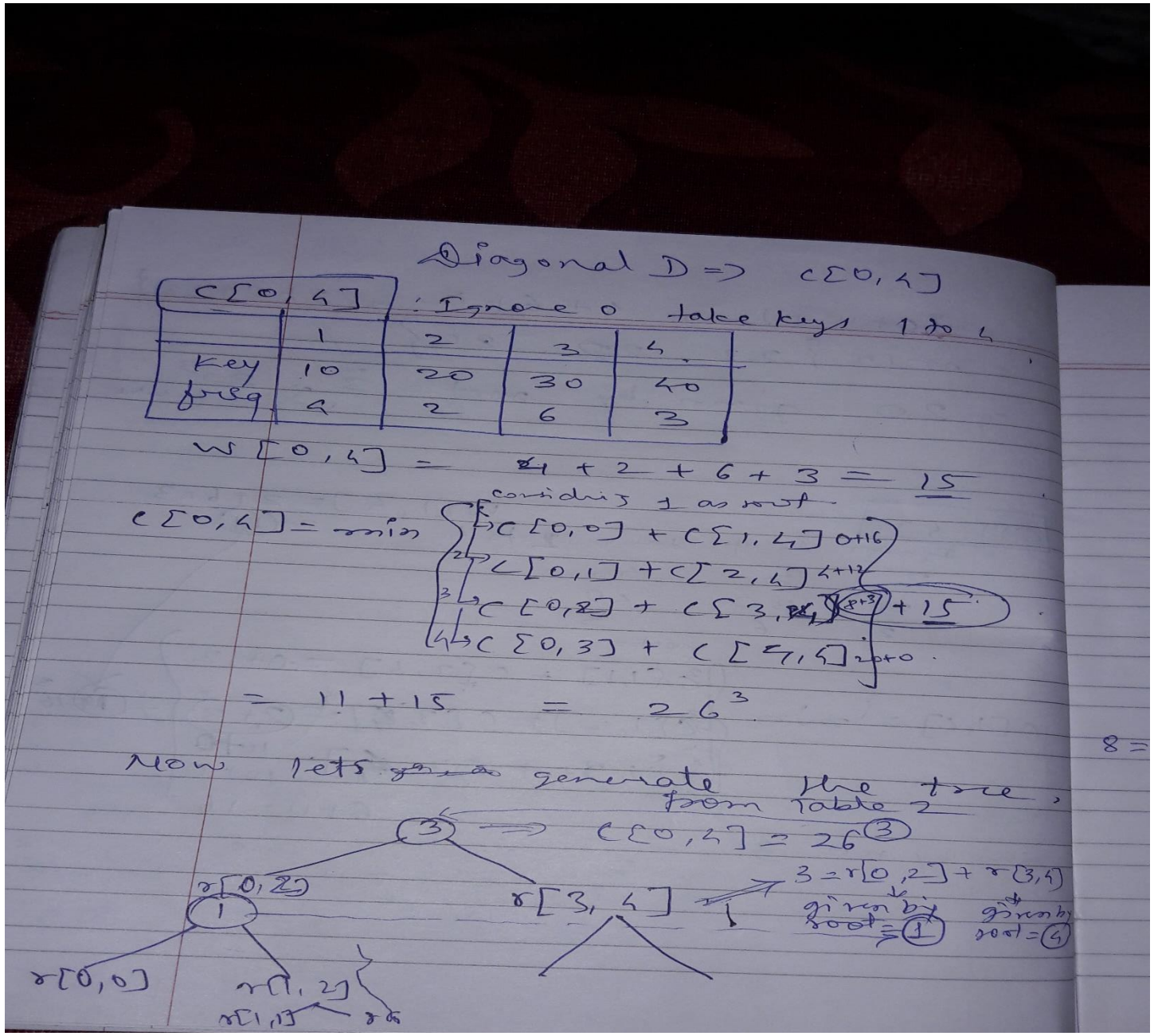


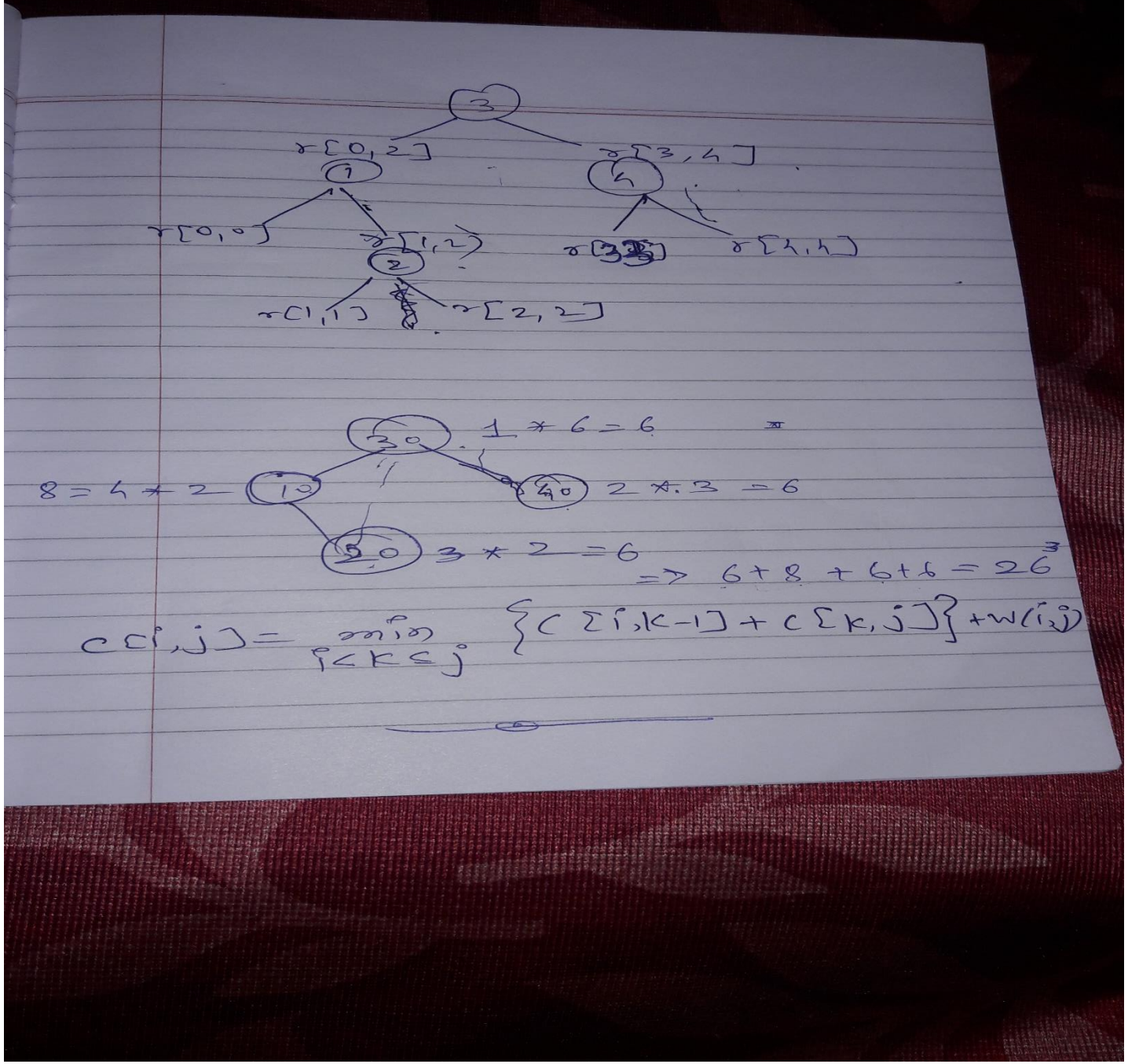
Instead of using above method, use directly the formula:

$$w[0,3] = 12 \cdot = \sum_{i=1}^{3} f(i)$$

$$c[0,3] = \min\{c[0,0] + c[1,3] + 12$$
$$c[0,1] + c[2,3] + 12$$
$$c[0,2] + [c[3,3] + 12\}.$$


= min { 0+10+12 , 4+6+12 , 8+0+12 }
= min { 22 , 22 , 20 }
= 20 given by root = 3 : 20^3 write in table
C[1, 4]
Ignore 1 & take 2 to 4.
2 3 4
20 30 40
2 6 3
W[1, 4] = 2+6+3 = 11
root =
C[1, 4] = min { C[1,1] + C[2, 4] = 0+12 ; C[1, 2] + C[3, 4] = 2+3 ; C[1, 3] + C[4, 4] = 10+0 }
= 6+11 = 16
given by root = 3
= min { 12 ; 5 ; 10 } + 11 = min { 5+11 } = 16


Diagonal D => c[0,4]
c[0,4] : Ignore 0 take keys 1 to 4.
1 2 3 4
key 10 20 30 40
freq 4 2 6 3
w[0,4] = 4 + 2 + 6 + 3 = 15
considering 1 as root.
c[0,4] = min
1 -> c[0,0] + c[1,4] 0+16
2 -> c[0,1] + c[2,4] 4+12
3 -> c[0,2] + c[3,4] 8+3 + 15
4 -> c[0,3] + c[4,4] 26+0
= 11 + 15 = 26
Now lets generate the tree from Table 2
c[0,4] = 26 (3)
r[0,2]
r[3,4]
3 = r[0,2] + r[3,4]
given by root = 1
given by root = 4
r[0,0]
r[1,2]
r[1,1]


r[0,2]
r[3,4]
r[0,0]
r[1,2]
r[3,3]
r[4,4]
r[1,1]
r[2,2]
1 * 6 = 6
8 = 4 * 2
2 * 3 = 6
3 * 2 = 6
=> 6 + 8 + 6 + 6 = 26
c[i,j] = min i<k≤j {c[i,k-1] + c[k,j]} + w(i,j)

Since the recursive implementation calls same suproblems again and again , this problem has Overlapping Subprolems property.

So optimal BST problem has both properties of a dynamic programming problem.

Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array cost[][] in bottom up manner.

Dynamic Programming Solution :

We use an auxiliary array cost[n][n] to store the solutions of subproblems.
cost[0][n-1] will hold the final result.
The challenge in implementation is, all diagonal values must be filled first, then the values which lie on the line just above the diagonal.
In other words, we must first fill all cost[i][i] values, then all cost[i][i+1] values, then all cost[i][i+2] values.
So, to fill the 2D array in such manner, the idea used in the implementation is same as Matrix Chain Multiplication problem, we use a variable 'L' for chain length and increment 'L', one by one. We calculate column number 'j' using the values of 'i' and 'L'.

```
// Dynamic Programming code for Optimal Binary Search  Tree Problem
#include <stdio.h>
#include <limits.h>

// A utility function to get sum of array elements
// freq[i] to freq[j]
int sum(int freq[], int i, int j);

/* A Dynamic Programming based function that calculates
  minimum cost of a Binary Search Tree. */
int optimalSearchTree(int keys[], int freq[], int n)
{
   /* Create an auxiliary 2D matrix to store results
     of subproblems */
   int cost[n][n];

/* cost[i][j] = Optimal cost of binary search tree
      that can be  formed from keys[i] to keys[j].
      cost[0][n-1] will store the resultant cost */

   // For a single key, cost is equal to frequency of the key
   for (int i = 0; i < n; i++)      O(n)
      cost[i][i] = freq[i];

   // Now we need to consider chains of length 2, 3, ... .
   // L is chain length.
   for (int L=2; L<=n; L++)       O(n)

   {
```

```
// i is row number in cost[][]
    for (int i=0; i<=n-L+1; i++)                    O(n)
    {
        // Get column number j from row number i and
        // chain length L
        int j = i+L-1;
        cost[i][j] = INT_MAX;

        // Try making all keys in interval keys[i..j] as root
        for (int r=i; r<=j; r++)                    O(n)
        {
  // c = cost when keys[r] becomes root of this subtree
            int c = ((r > i)? cost[i][r-1]:0) +
                    ((r < j)? cost[r+1][j]:0) +
                    sum(freq, i, j);                O(n)
            if (c < cost[i][j])
              cost[i][j] = c;
        }
    }
    return cost[0][n-1];
}
// A utility function to get sum of array elements
// freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)        O(n)
      s += freq[k];
    return s;
}
 // Driver program to test above functions
int main()
{   int keys[] = {10, 12, 20};    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ",
                optimalSearchTree(keys, freq, n));
    return 0;
}
```

1) The time complexity of the above solution is $O(n^4) = O(n) + O(n^4)$

The time complexity can be easily reduced to $O(n^3)$ by pre-calculating sum of frequencies instead of calling sum() again and again.

2) In the above solutions, we have computed optimal cost only.

The solutions can be easily modified to store the structure of BSTs also.

We can create another auxiliary array of size n to store the structure of tree.

All we need to do is, store the chosen 'r' in the innermost loop.

End of Module 4

Thank You.