Unit - 3

1. Dynamic Programming

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

Let's understand this approach through an example.

Consider an example of the Fibonacci series. The following series is the Fibonacci series:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ,...

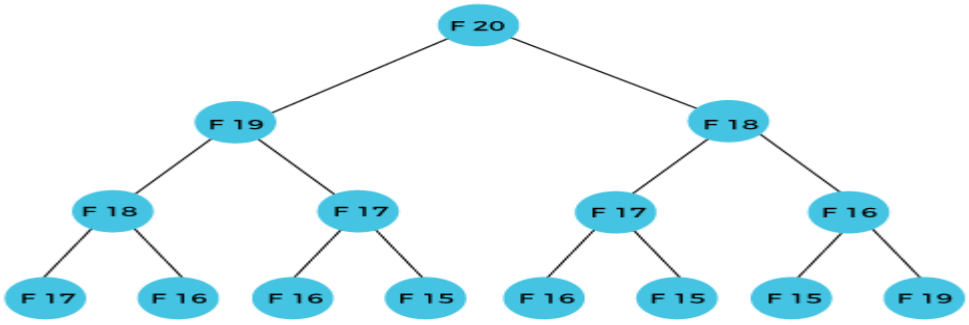
The numbers in the above series are not randomly calculated. Mathematically, we could write each of the terms using the below formula:

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

With the base values F(0) = 0, and F(1) = 1. To calculate the other numbers, we follow the above relationship. For example, F(2) is the sum **f(0)** and **f(1),** which is equal to 1.

How can we calculate F(20)?

The F(20) term will be calculated using the nth formula of the Fibonacci series. The below figure shows that how F(20) is calculated.



As we can observe in the above figure that F(20) is calculated as the sum of F(19) and F(18). In the dynamic programming approach, we try to divide the problem into the similar subproblems. We are following this approach in the above case where F(20) into the similar subproblems, i.e., F(19) and F(18). If we recap the definition of dynamic programming that it says the similar subproblem should not be computed more than once. Still, in the above case, the subproblem is calculated twice. In the above example, F(18) is calculated two times; similarly, F(17) is also calculated twice. However, this technique is quite useful as it solves the similar subproblems, but we need to be cautious while storing the results because we are not particular about storing the result that we have computed once, then it can lead to a wastage of resources.

In the above example, if we calculate the F(18) in the right subtree, then it leads to the tremendous usage of resources and decreases the overall performance.

The solution to the above problem is to save the computed results in an array. First, we calculate F(16) and F(17) and save their values in an array. The F(18) is calculated by summing the values of F(17) and F(16), which are already saved in an array. The computed value of F(18) is saved in an array. The value of F(19) is calculated using the sum of F(18), and F(17), and their values are already saved in an array. The computed value of F(19) is stored in an array. The value of F(20) can be calculated by adding the values of F(19) and F(18), and the values of both F(19) and F(18) are stored in an array. The final computed value of F(20) is stored in an array.

How does the dynamic programming approach work?

The following are the steps that the dynamic programming follows:

- It breaks down the complex problem into simpler subproblems.
- It finds the optimal solution to these sub-problems.
- It stores the results of subproblems (memoization). The process of storing the results of subproblems is known as memorization.
- It reuses them so that same sub-problem is calculated more than once.
- Finally, calculate the result of the complex problem.

The above five steps are the basic steps for dynamic programming. The dynamic programming is applicable that are having properties such as:

Those problems that are having overlapping subproblems and optimal substructures. Here, optimal substructure means that the solution of optimization problems can be obtained by simply combining the optimal solution of all the subproblems.

In the case of dynamic programming, the space complexity would be increased as we are storing the intermediate results, but the time complexity would be decreased.

Approaches of dynamic programming

There are two approaches to dynamic programming:

- Top-down approach
- Bottom-up approach

Top-down approach

The top-down approach follows the memorization technique, while bottom-up approach follows the tabulation method. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results.

Advantages

- It is very easy to understand and implement.
- It solves the subproblems only when it is required.
- It is easy to debug.

Disadvantages

It uses the recursion technique that occupies more memory in the call stack. Sometimes when the recursion is too deep, the stack overflow condition will occur.

It occupies more memory that degrades the overall performance.

Bottom-Up approach

The bottom-up approach is also one of the techniques which can be used to implement the dynamic programming. It uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.

There are two ways of applying dynamic programming:

- **Top-Down**
- **Bottom-Up**

The bottom-up is the approach used to avoid the recursion, thus saving the memory space. The bottom-up is an algorithm that starts from the beginning, whereas the recursive algorithm starts from the end and works backward. In the bottom-up approach, we start from the base case to find the answer for the end. As we know, the base cases in the Fibonacci series are 0 and 1. Since the bottom approach starts from the base cases, so we will start from 0 and 1.

Key points

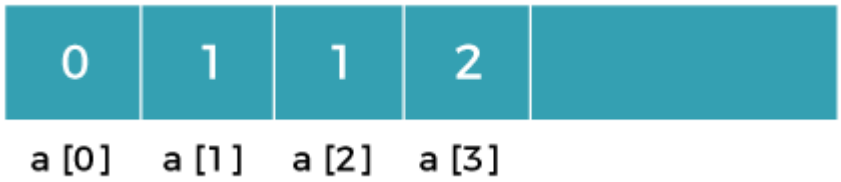
- We solve all the smaller sub-problems that will be needed to solve the larger sub-problems then move to the larger problems using smaller sub-problems.
- We use for loop to iterate over the sub-problems.
- The bottom-up approach is also known as the tabulation or table filling method.

Let's understand through an example.

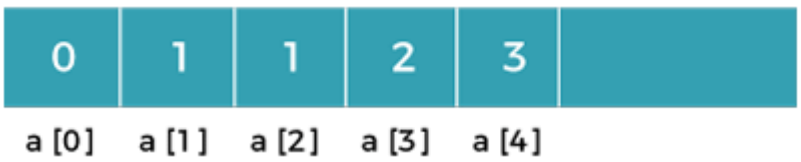
Suppose we have an array that has 0 and 1 values at a[0] and a[1] positions, respectively shown as below:

Since the bottom-up approach starts from the lower values, so the values at a[0] and a[1] are added to find the value of a[2] shown as below:

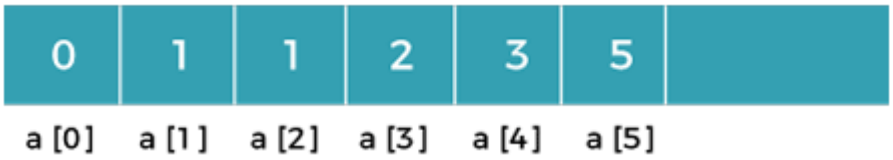
The value of a[3] will be calculated by adding a[1] and a[2], and it becomes 2 shown as below:



The value of a[4] will be calculated by adding a[2] and a[3], and it becomes 3 shown as below:



The value of a[5] will be calculated by adding the values of a[4] and a[3], and it becomes 5 shown as below:



The code for implementing the Fibonacci series using the bottom-up approach is given below:

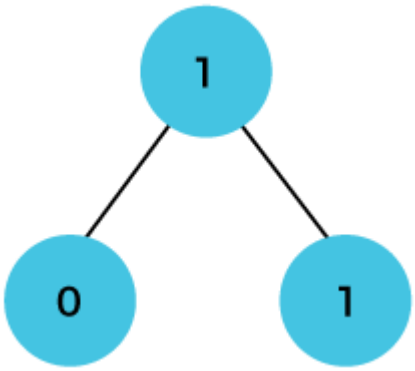
```
1.      int fib(int n)
2.      {
3.         int A[];
4.         A[0] = 0, A[1] = 1;
5.         for( i=2; i<=n; i++)
6.         {
7.             A[i] = A[i-1] + A[i-2]
8.         }
9.         return A[n];
10.     }
```

In the above code, base cases are 0 and 1 and then we have used for loop to find other values of Fibonacci series.

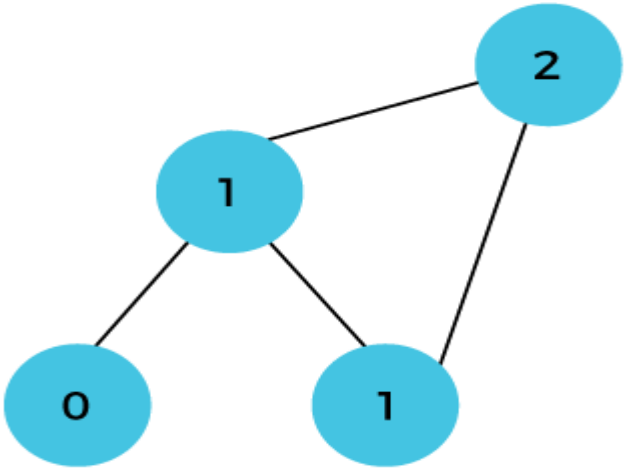
Let's understand through the diagrammatic representation.

Initially, the first two values, i.e., 0 and 1 can be represented as:

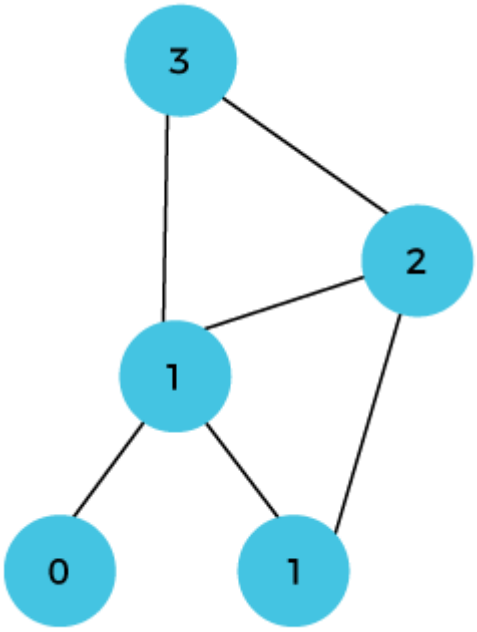
When i=2 then the values 0 and 1 are added shown as below:



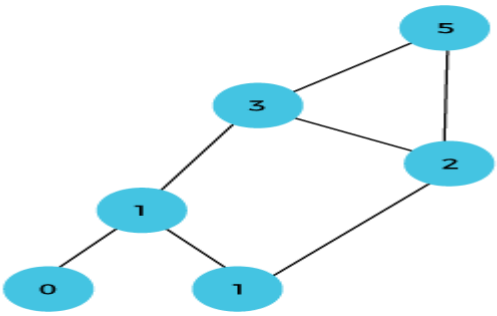
When i=3 then the values 1and 1 are added shown as below:



When i=4 then the values 2 and 1 are added shown as below:



When i=5, then the values 3 and 2 are added shown as below:



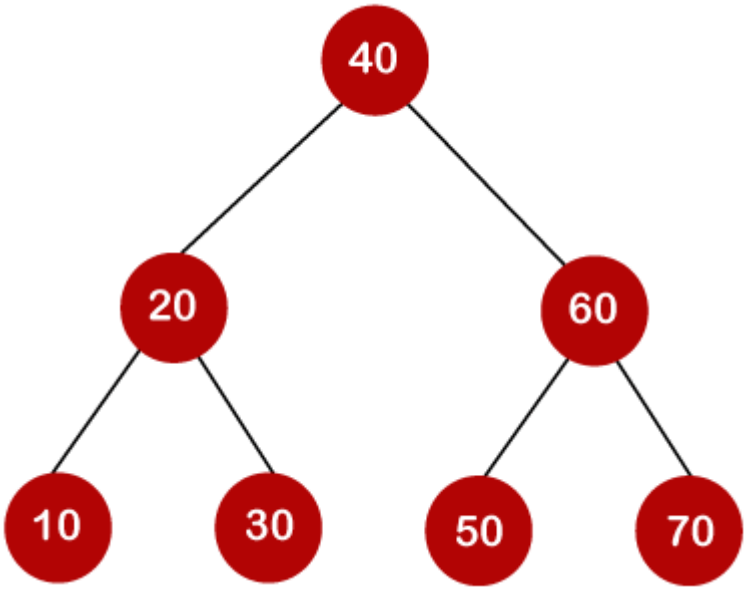
2. Optimal Binary Search Tree

As we know that in binary search tree, the nodes in the left subtree have lesser value than the root node and the nodes in the right subtree have greater value than the root node. It is also called as ordered binary tree.

We know the key values of each node in the tree, and we also know the frequencies of each node in terms of searching means how much time is required to search a node. The frequency and key-value determine the overall cost of searching a node. The cost of searching is a very important factor in various applications. The overall cost of searching a node should be less. The time required to search a node in BST is more than the balanced binary search tree as a balanced binary search tree contains a lesser number of levels than the BST. There is one way that can reduce the cost of a binary search tree is known as an **optimal binary search tree**.

Let's understand through an example.

If the keys are 10, 20, 30, 40, 50, 60, 70



In the above tree, all the nodes on the left subtree are smaller than the value of the root node, and all the nodes on the right subtree are larger than the value of the root node. The maximum time required to search a node is equal to the minimum height of the tree, equal to logn.

Now we will see how many binary search trees can be made from the given number of keys.

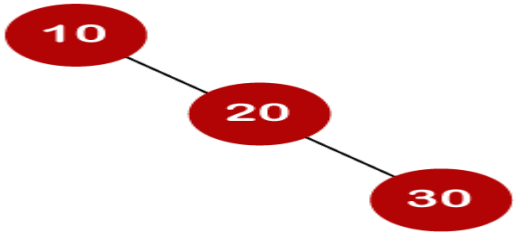
For example: 10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.

The Formula for calculating the number of trees:

$$\frac{{}^{2n}C_n}{n+1}$$

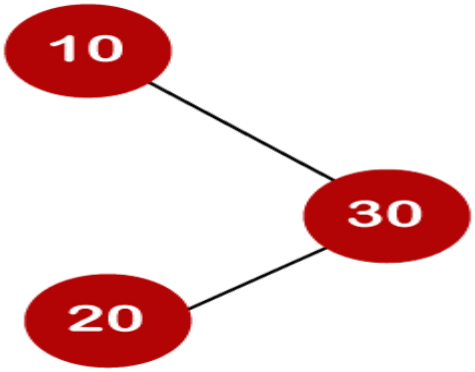
When we use the above formula, then it is found that total 5 number of trees can be created.

The cost required for searching an element depends on the comparisons to be made to search an element. Now, we will calculate the average cost of time of the above binary search trees.



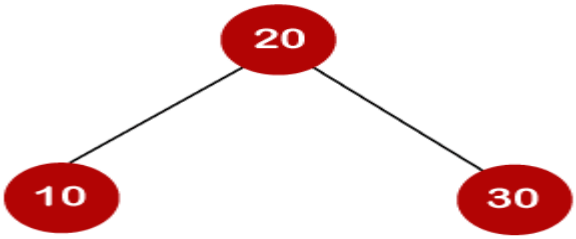
In the above tree, total number of 3 comparisons can be made. The average number of comparisons can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



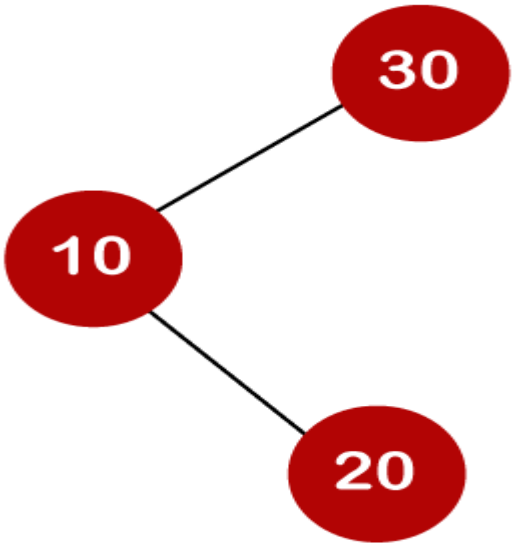
In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



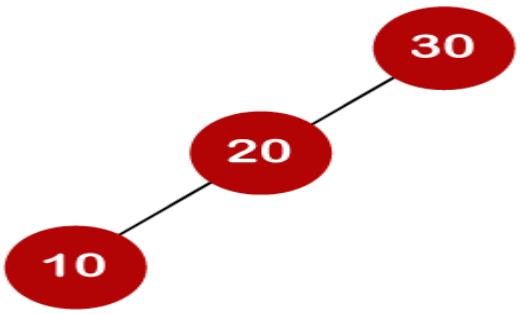
In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+2}{3} = 5/3$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$

In the third case, the number of comparisons is less because the height of the tree is less, so it's a balanced binary search tree.

Till now, we read about the height-balanced binary search tree. To find the optimal binary search tree, we will determine the frequency of searching a key.

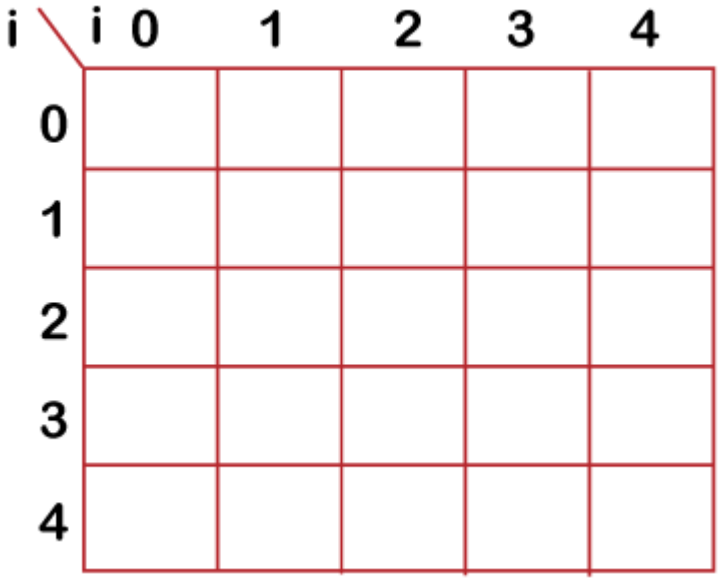
Let's assume that frequencies associated with the keys 10, 20, 30 are 3, 2, 5.

The above trees have different frequencies. The tree with the lowest frequency would be considered the optimal binary search tree. The tree with the frequency 17 is the lowest, so it would be considered as the optimal binary search tree.

Dynamic Approach

Consider the below table, which contains the keys and frequencies.

	1	2	3	4
Keys →	10	20	30	40
Frequency →	4	2	6	3



First, we will calculate the values where j-i is equal to zero.

When i=0, j=0, then j-i = 0

When i = 1, j=1, then j-i = 0

When i = 2, j=2, then j-i = 0

When i = 3, j=3, then j-i = 0

When i = 4, j=4, then j-i = 0

Therefore, c[0, 0] = 0, c[1 , 1] = 0, c[2,2] = 0, c[3,3] = 0, c[4,4] = 0

Now we will calculate the values where j-i equal to 1.

When j=1, i=0 then j-i = 1

When j=2, i=1 then j-i = 1

When j=3, i=2 then j-i = 1

When j=4, i=3 then j-i = 1

Now to calculate the cost, we will consider only the jth value.

The cost of c[0,1] is 4 (The key is 10, and the cost corresponding to key 10 is 4).

The cost of c[1,2] is 2 (The key is 20, and the cost corresponding to key 20 is 2).

The cost of c[2,3] is 6 (The key is 30, and the cost corresponding to key 30 is 6)

The cost of c[3,4] is 3 (The key is 40, and the cost corresponding to key 40 is 3)

	0	1	2	3	4
0	0	4			
1		0	2		
2			0	6	
3				0	3
4					0

Now we will calculate the values where j-i = 2

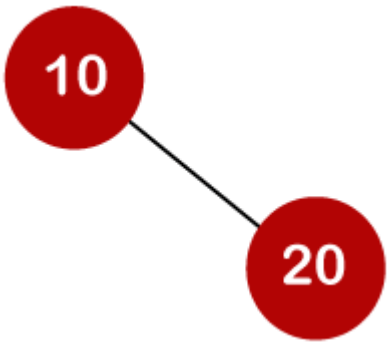
When j=2, i=0 then j-i = 2

When j=3, i=1 then j-i = 2

When j=4, i=2 then j-i = 2

In this case, we will consider two keys.

- When i=0 and j=2, then keys 10 and 20. There are two possible trees that can be made out from these two keys shown below:



In the first binary tree, cost would be: 4*1 + 2*2 = 8

In the second binary tree, cost would be: 4*2 + 2*1 = 10

The minimum cost is 8; therefore, c[0,2] = 8

	0	1	2	3	4
0	0	4	8		
1		0	2		
2			0	6	
3				0	3
4					0

- When i=1 and j=3, then keys 20 and 30. There are two possible trees that can be made out from these two keys shown below:

In the first binary tree, cost would be: 1*2 + 2*6 = 14

In the second binary tree, cost would be: 1*6 + 2*2 = 10

The minimum cost is 10; therefore, c[1,3] = 10

- When i=2 and j=4, we will consider the keys at 3 and 4, i.e., 30 and 40. There are two possible trees that can be made out from these two keys shown as below:

In the first binary tree, cost would be: 1*6 + 2*3 = 12

In the second binary tree, cost would be: 1*3 + 2*6 = 15

The minimum cost is 12, therefore, c[2,4] = 12

i \ 1	0	1	2	3	4
0	0	4	8^{1}		
1		0	2	10^{3}	
2			0	6	12^{3}
3				0	3
4					0

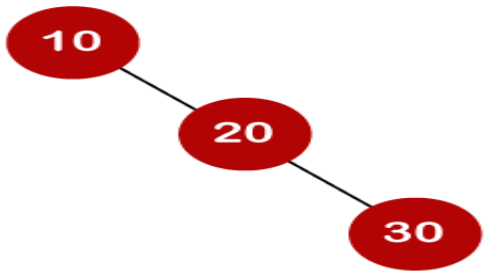
Now we will calculate the values when j-i = 3

When j=3, i=0 then j-i = 3

When j=4, i=1 then j-i = 3

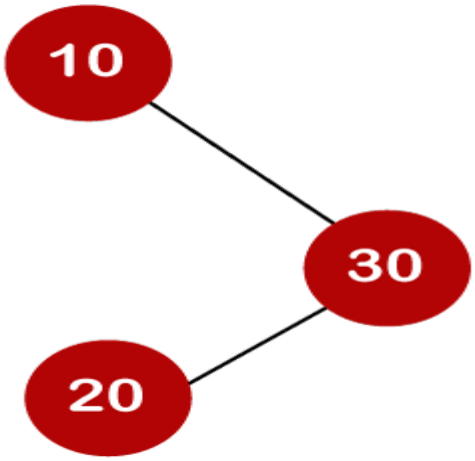
- When i=0, j=3 then we will consider three keys, i.e., 10, 20, and 30.

The following are the trees that can be made if 10 is considered as a root node.



In the above tree, 10 is the root node, 20 is the right child of node 10, and 30 is the right child of node 20.

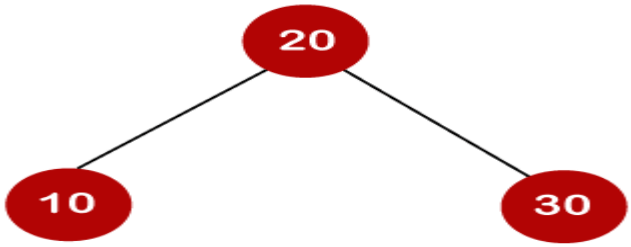
Cost would be: 1*4 + 2*2 + 3*6 = 26



In the above tree, 10 is the root node, 30 is the right child of node 10, and 20 is the left child of node 20.

Cost would be: 1*4 + 2*6 + 3*2 = 22

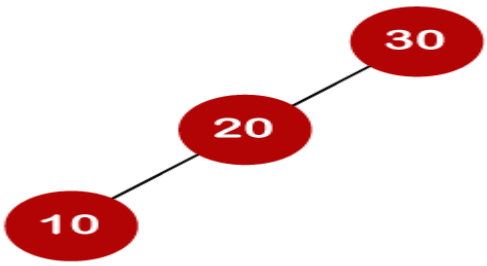
The following tree can be created if 20 is considered as the root node.



In the above tree, 20 is the root node, 30 is the right child of node 20, and 10 is the left child of node 20.

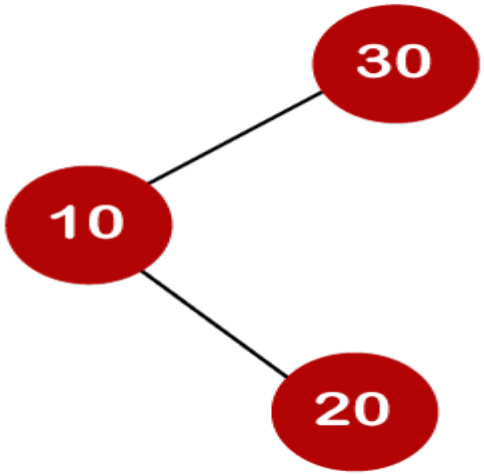
Cost would be: 1*2 + 4*2 + 6*2 = 22

The following are the trees that can be created if 30 is considered as the root node.



In the above tree, 30 is the root node, 20 is the left child of node 30, and 10 is the left child of node 20.

Cost would be: 1*6 + 2*2 + 3*4 = 22



In the above tree, 30 is the root node, 10 is the left child of node 30 and 20 is the right child of node 10.

Cost would be: 1*6 + 2*4 + 3*2 = 20

Therefore, the minimum cost is 20 which is the 3rd root. So, c[0,3] is equal to 20.

- When i=1 and j=4 then we will consider the keys 20, 30, 40

c[1,4] = min{ c[1,1] + c[2,4], c[1,2] + c[3,4], c[1,3] + c[4,4] } + 11

= min{0+12, 2+3, 10+0}+ 11

= min{12, 5, 10} + 11

The minimum value is 5; therefore, c[1,4] = 5+11 = 16

i \ j	0	1	2	3	4
0	0	4	8^1	20^3	
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

- **Now we will calculate the values when j-i = 4**

When j=4 and i=0 then j-i = 4

In this case, we will consider four keys, i.e., 10, 20, 30 and 40. The frequencies of 10, 20, 30 and 40 are 4, 2, 6 and 3 respectively.

w[0, 4] = 4 + 2 + 6 + 3 = 15

If we consider 10 as the root node then

C[0, 4] = min {c[0,0] + c[1,4]}+ w[0,4]

= min {0 + 16} + 15= 31

If we consider 20 as the root node then

C[0,4] = min{c[0,1] + c[2,4]} + w[0,4]

= min{4 + 12} + 15

= 16 + 15 = 31

If we consider 30 as the root node then,

C[0,4] = min{c[0,2] + c[3,4]} +w[0,4]

= min {8 + 3} + 15

= 26

If we consider 40 as the root node then,

C[0,4] = min{c[0,3] + c[4,4]} + w[0,4]

= min{20 + 0} + 15

= 35

In the above cases, we have observed that 26 is the minimum cost; therefore, c[0,4] is equal to 26.

i \ j	0	1	2	3	4
0	0	4	8^1	20^3	26^3
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

General formula for calculating the minimum cost is:

C[i,j] = min{c[i, k-1] + c[k,j]} + w(i,j).