Optimal Binary Search Tree

Given a sorted array key *[0.. n-1]* of search keys and an array *freq[0.. n-1]* of frequency counts, where *freq[i]* is the number of searches for *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 the level of that node multiplied by its frequency. The level of the root is 1.

Examples:

```
Input:  keys[] = {10, 12}, freq[] = {34, 50}
There can be following two possible BSTs
        10                       12
          \                     /
           12                 10
          I                     II
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


Input:  keys[] = {10, 12, 20}, freq[] = {34, 8, 50}
There can be following possible BSTs
    10                12                  20         10              20
      \             /    \              /              \            /
      12          10     20           12               20         10
        \                            /                 /             \
         20                        10                12               12
     I               II             III             IV             V
Among all possible BSTs, cost of the fifth BST is minimum.
Cost of the fifth BST is 1*50 + 2*34 + 3*8 = 142
```

Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.

1) Optimal Substructure:
The optimal cost for freq[i..j] can be recursively calculated using the following formula.

$$optcost\,(i,\ j) = \sum_{k=i}^{j} freq\,[k] + min_{r=i}^{j}\,[optcost(i, r-1) + optcost(r+1, j)]$$

We need to calculate ***optCost(0, n-1)*** to find the result.
The idea of above formula is simple, we one by one try all nodes as root (r varies from i to j in second

term). When we make *rth* node as root, we recursively calculate optimal cost from i to r-1 and r+1 to j. We add sum of frequencies from i to j (see first term in the above formula)

The reason for adding the sum of frequencies from i to j:

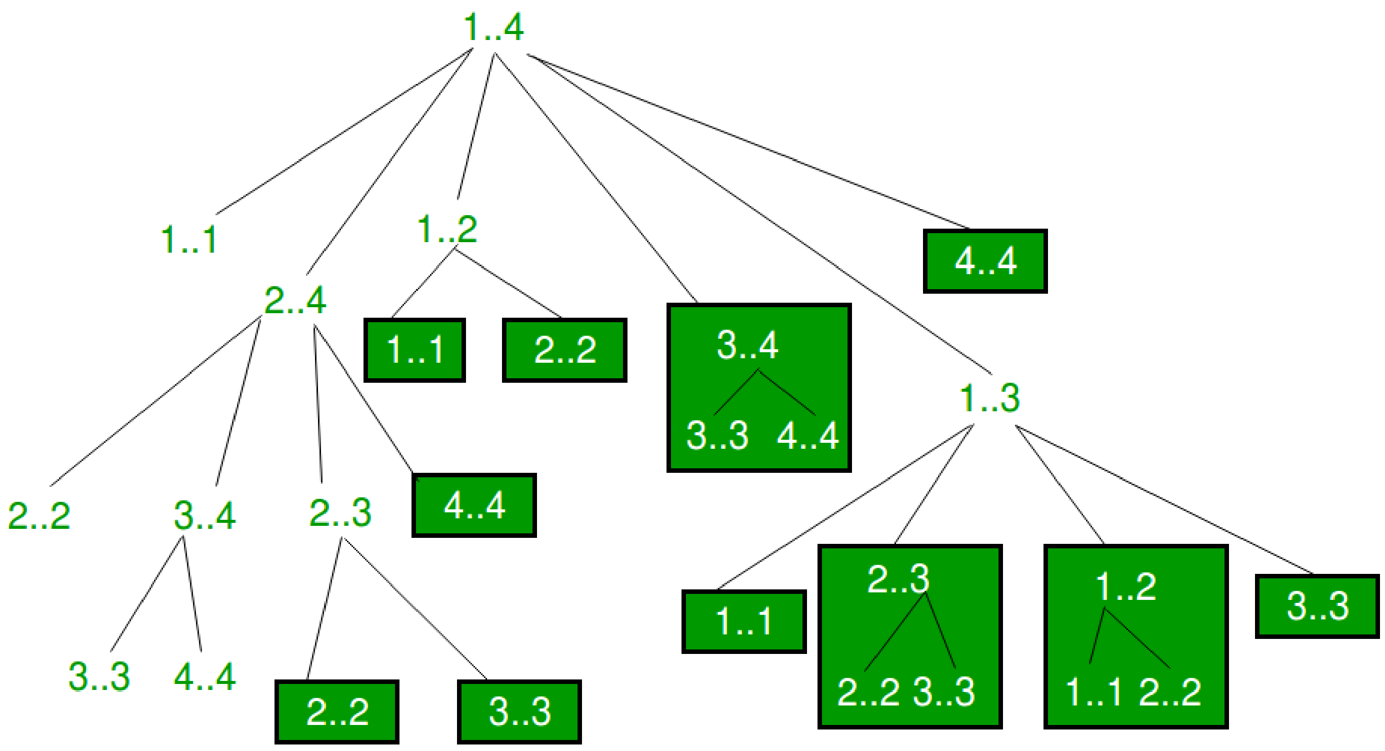
This can be divided into 2 parts one is the freq[r]+sum of frequencies of all elements from i to j except r. The term freq[r] is added because it is going to be root and that means level of 1, so freq[r]*1=freq[r]. Now the actual part comes, we are adding the frequencies of remaining elements because as we take r as root then all the elements other than that are going 1 level down than that is calculated in the subproblem. Let me put it in a more clear way, for calculating optcost(i,j) we assume that the r is taken as root and calculate min of opt(i,r-1)+opt(r+1,j) for all i<=r<=j. Here for every subproblem we are choosing one node as a root. But in reality the level of subproblem root and all its descendant nodes will be 1 greater than the level of the parent problem root. Therefore the frequency of all the nodes except r should be added which accounts to the descend in their level compared to level assumed in subproblem.

2) Overlapping Subproblems

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

Dynamic Programming Solution

Following is C/C++ implementation for optimal BST problem using Dynamic Programming. 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 how 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’.



```python
import sys


def optimalBinaryStructure(arr, cost):
    dp = [[0] * len(arr) for _ in range(len(arr))]
    prefixSum = [0] * (len(cost))
    for i in range(len(arr)):
        if i == 0:
            prefixSum[i] = cost[i]
        else:
            prefixSum[i] = prefixSum[i - 1] + cost[i]
    for gap in range(len(arr)):
        i = 0
        j = gap
        while j < len(arr):
            if gap == 0:
                dp[i][j] = cost[i]
            elif gap == 1:
                f1 = cost[i - 1] + 2 * cost[i]
                f2 = 2 * cost[i - 1] + cost[i]
                dp[i][j] = min(f1, f2)
            else:
                minCost = sys.maxsize
                addedCost = prefixSum[j] - (0 if i == 0 else prefixSum[i -
1])
                for k in range(i, j + 1):
                    leftTree = 0 if k == 0 else dp[i][k - 1]
                    rightTree = 0 if k == j else dp[k + 1][j]
                    minCost = min(minCost, leftTree + rightTree)

                dp[i][j] = addedCost + minCost
            i = i + 1
            j = j + 1
    return dp[0][-1]


keys = [10, 12, 20]
freq = [34, 8, 50]
print(optimalBinaryStructure(keys, freq))
```