Algorithms

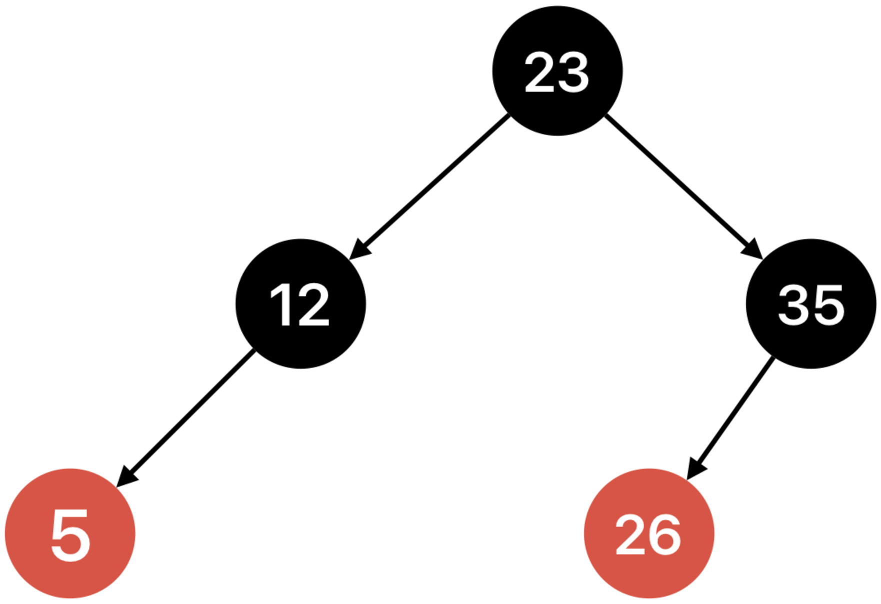
Assignment 7

Ashwin Sathiyanarayanan
002227609

Question 1:

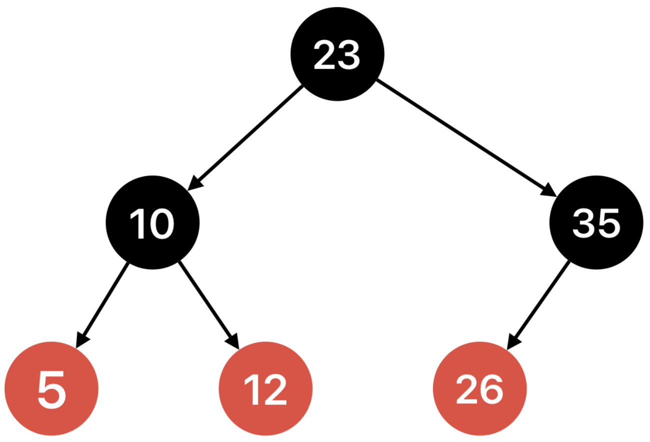
If we first insert an element and then immediately delete that element from the Red-Black tree, the tree may or may not be the same as before. This is because, both the insert and delete operations have some conditions in them, which when satisfied, call the methods that transplant a part of the tree, right rotates a subtree, or left rotates a subtree or at times all three occur. These steps are in place so the tree can become more balanced after each insertion and deletion. But, when certain conditions are not met, none of the above such methods are called and we are left with the same tree structure that we started with. So that's why when first inserting an element and immediately deleting that element, the tree may or may not be the same. Let's see this with an example:

Let's consider a RBT as below:

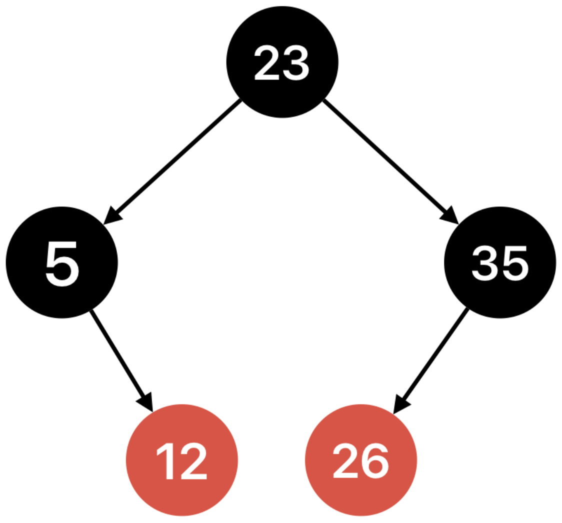


Let's first insert and delete 10:

After inserting 10:



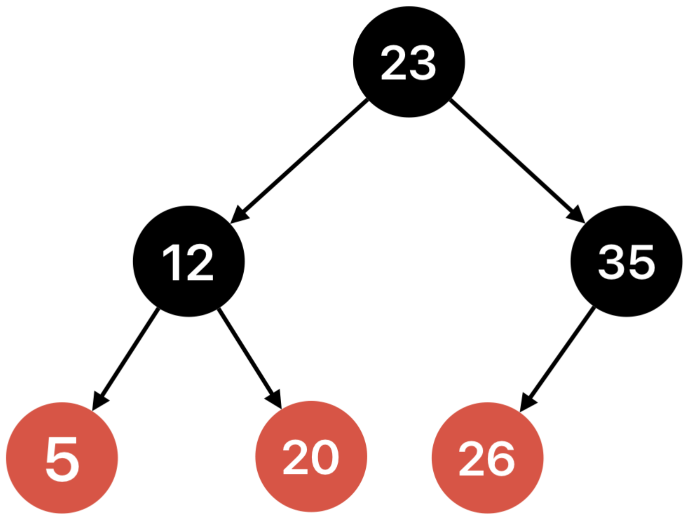
After Deleting 10:



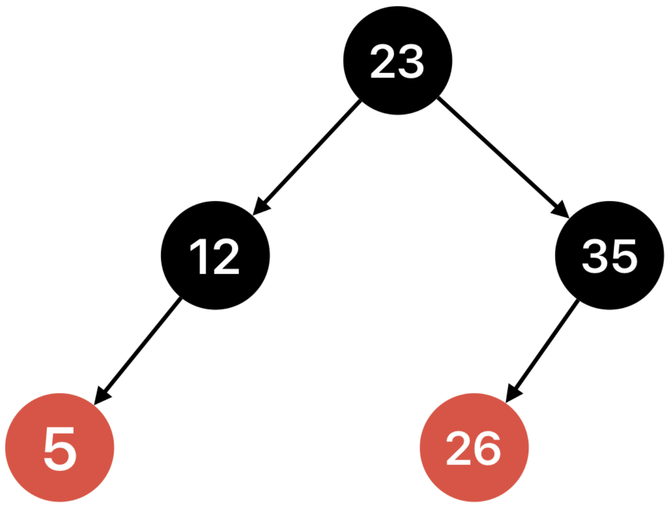
As we can see, the tree changes after inserting and deleting 10.

Now, let's insert and delete 20:

After inserting 20:



After deleting 20:



In this case, the tree remains the same as that of the original tree.
So, the tree may or may not be the same after inserting an element and immediately deleting that element.

Question 2:

This greedy strategy of finding the price density of the rod, its price per inch, does not always provide the optimal solution. This is because the algorithm may select a cut with maximum density and add up the remaining length's price to get the maximum price. Still, there may be other options where the density might not be the highest but the combination of the prices may be higher than that of the former. Let's see an example of this scenario,

	1	2	3	4
P_i	1	6	15	18
P_i / i	1	3	5	4.5

In this example, this greedy algorithm would cut a piece of length 3 because it would have the maximum density. So we will be left with a piece of length 1. So the total price for this combination of pieces will be 16. But, that is not the maximum price that we can get. If we do not make any cuts, then we can get a price of 18 which is higher than the price of 16. So in this case this greedy algorithm does not provide an optimal solution.

Question 3:

```
rod_cutting_with_cost(p, n, r, c):
    if r[n] >= 0:
        return r[n]
    if n == 0:
        q = 0
    else:
        q = -∞
        for i = 1 to n:
            q = max(q, p[i] + rod_cutting_with_cost(p, n - i, r, c) - c)
    r[n] = q
    return q
```

Explanation:

1. In order to modify the existing dynamic algorithm for rod cutting to also include the cost of cutting the rod, we can subtract the cost from the step where we compute the final cost.
2. This is the step where we recursively calculate the maximum of either the existing q value or the new rod-cutting price.
3. We can modify that step to compute the maximum value between the existing q value or the new cost of cutting the rod, which basically is the price of the rod cut subtracted by the cost of cutting the rod, c.
4. This whole process takes the same amount of time as the previous dynamic algorithm because we don't actually change anything from the existing algorithm other than just subtracting an extra value.

Question 4:

```
def FibonacciSeriesMemorized(n):
    let r[0 .... n] be a new array
    for i = 0 to n:
        r[i] = -∞
    return FibonacciSeriesMemorized_Aux(n, r)

def FibonacciSeriesMemorized_Aux(n, r):
    if r[n] > -1:
        return r[n]
    if n == 0:
        q = 0
    else if n == 1:
        q = 1
    else:
        q = FibonacciSeriesMemorized_Aux(n - 1, r)
                        + FibonacciSeriesMemorized_Aux(n - 2, r)
    r[n] = q
    return q
```

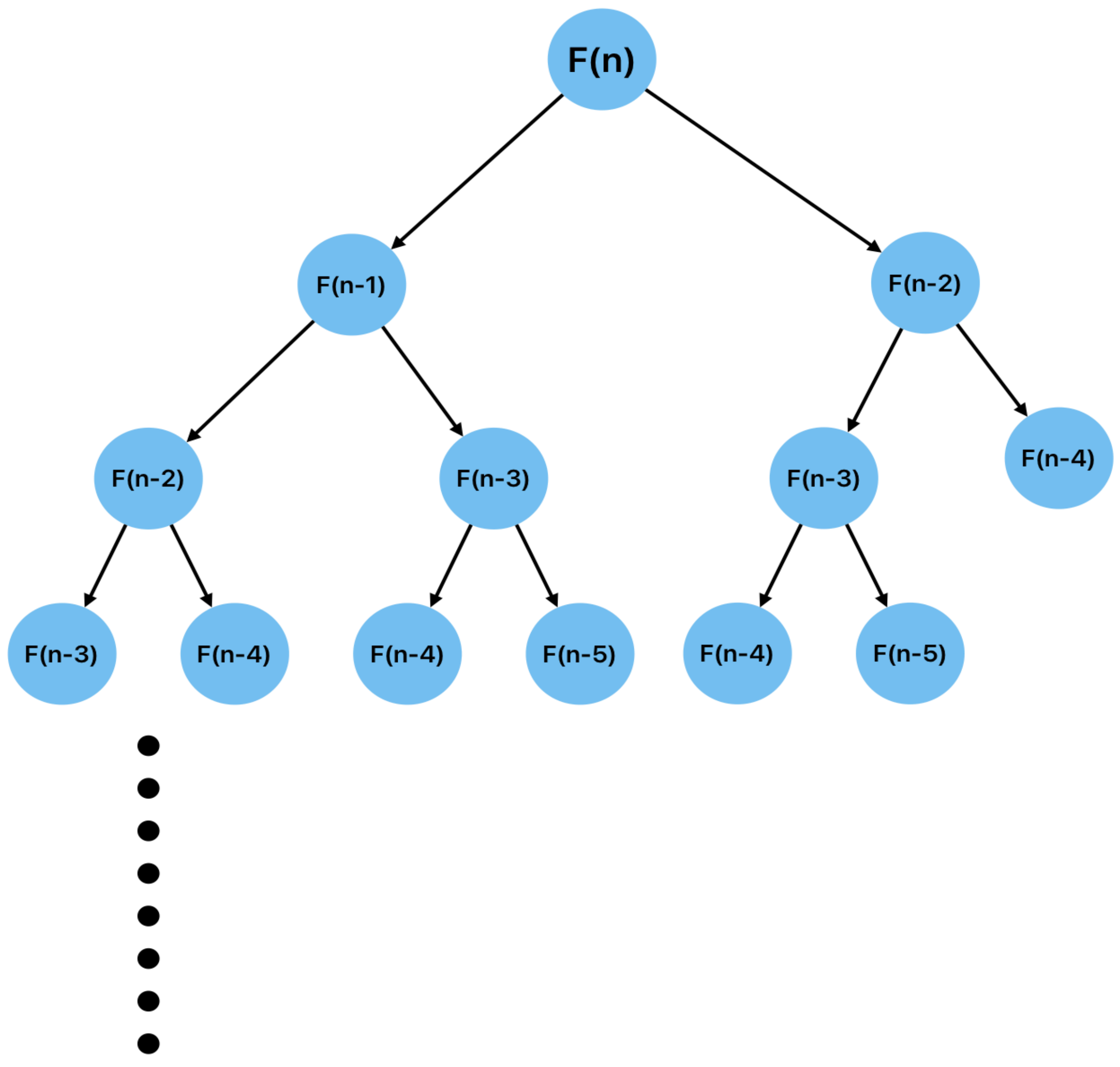
Explanation:

1. In order to compute the nth Fibonacci number using a dynamic programming algorithm, we can use two functions namely FibonacciSeriesMemorized and FibonacciSeriesMemorized_Aux.
2. First, we start from the FibonacciSeriesMemorized function which takes in the number n as the parameter. In this function, we create a new auxiliary array r to store the values of redundant subproblems. The array r ranges from 0 to n. Then we initialize each value inside the r array as negative infinity. Finally, we call the recursive FibonacciSeriesMemorized_Aux function with the number n and the memory array r as its parameter.
3. In the FibonacciSeriesMemorized_Aux, first, we check if we have already computed the value of this subproblem from the memory array r. If yes, then we just return that memorized value.
4. If not, then we check if n == 0 and if yes we set the value of q as 0, or

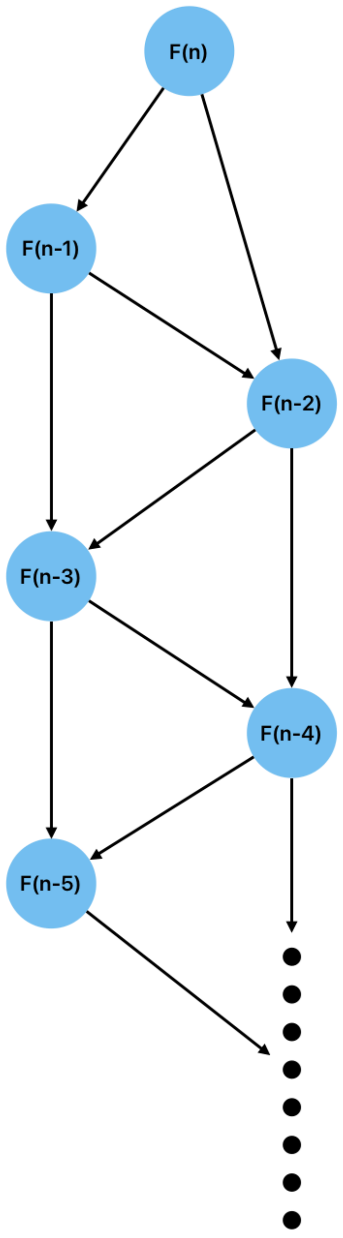
if n == 1, we set the value of q as 1. This is because, in a Fibonacci series, F0 and F1 are 0 and 1 respectively.

5. Then if n is a number greater than 1 and if we have not yet memorized the value for it, we recursively compute the value of q as the sum of the Fibonacci number of n - 1 and Fibonacci number of n - 2.
6. Finally, we set the value of r[n] as q so that in the next recursive call we would have memorized the value of Fn. Then we return q at the end.

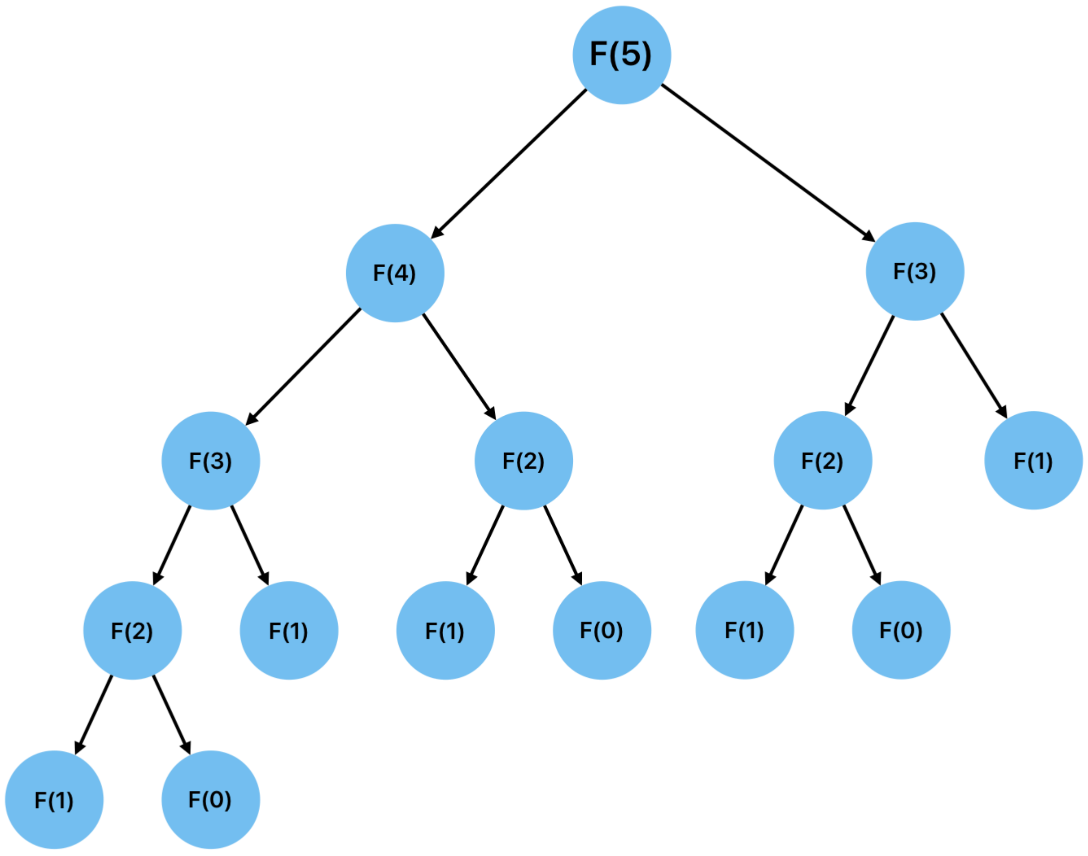
General Graph:



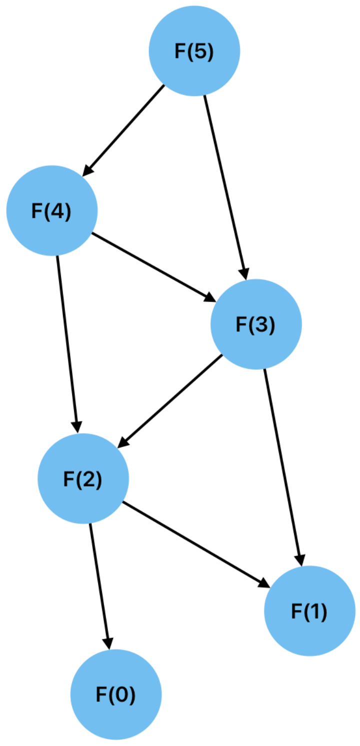
Optimal subproblem graph:



Graph for example: F(5)



Optimal sub-problem graph:



As we can see from the example optimal sub problem graph for a fibonacci of 5, the number of vertices is 6 and the number of edges is 8, because F(1) and F(0) has no outgoing edges. Therefore, we can create a general formula for the number of vertices and number of edges.

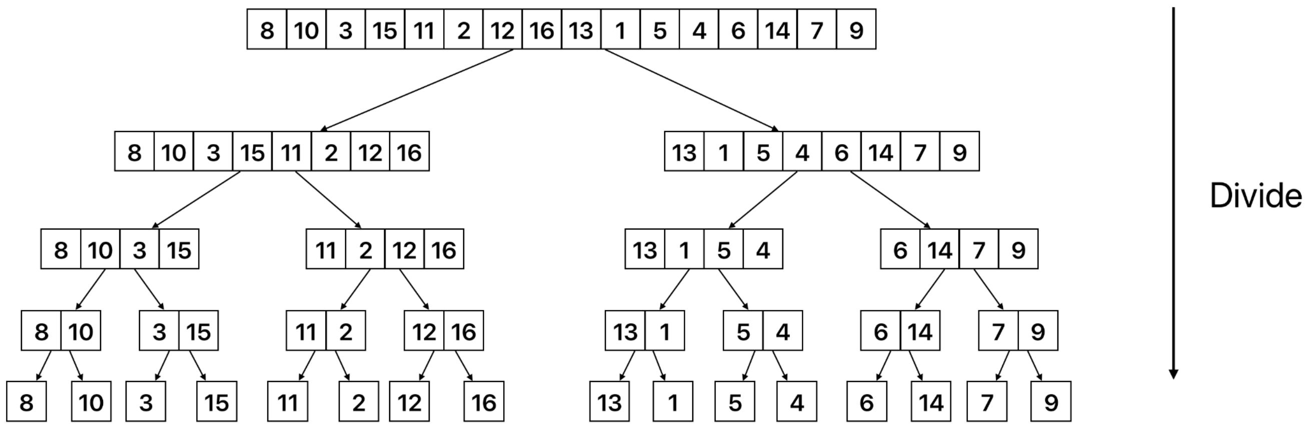
Number of vertices = n - 1
Number of edges = 2*(n - 1)

Question 5:

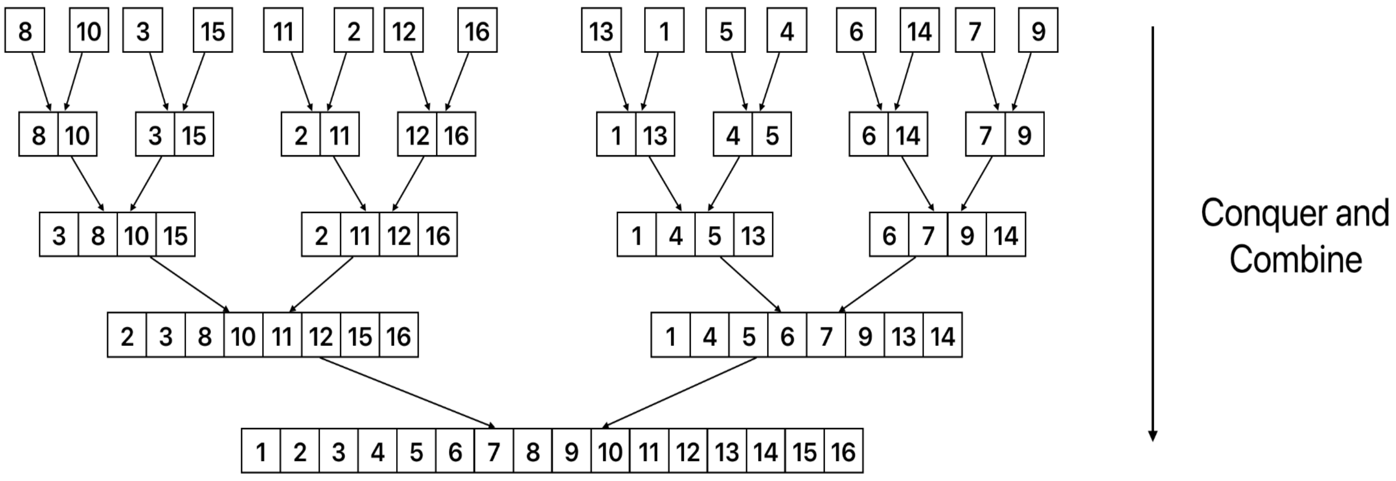
Let us consider an array of 16 elements like below:

8	10	3	15	11	2	12	16	13	1	5	4	6	14	7	9

Now, let us consider the recursive divide and conquer tree for the merge sort of this array:



Let's also visualize the conquer and combine tree for this merge sort:



Memorization fails to speed up a good divide-and-conquer algorithm such as merge-sort because, as we can see above, there are no redundant subproblems to be solved. In other words, memorization works well when we need to compute a problem in which we have to repeatedly calculate the same subproblem. But in the case of merge-sort, each subproblem is unique. We have to sort the elements in each of the subproblems separately and then combine the solutions of each of the subproblems in the combining step to get the final sorted array. Since each subproblem is unique, there is no need for memorization, and even if we did memorize the solutions for the subproblems, they would be of no use to us since we would not come across the same subproblem again. This is the reason why memorization fails to speed up a good divide-and-conquer algorithm such as merge-sort.

Question 6:

```
def LCSReconstruction(c, x, y, i, j):
    if i == 0 or j == 0:
        return

    if c[i, j] == c[i - 1, j - 1] + 1:
        LCSReconstruction(c, x, y, i - 1, j - 1)
        print $x_i$

    else if c[i, j] == c[i - 1, j]:
        LCSReconstruction(c, x, y, i - 1, j)

    else:
        LCSReconstruction(c, x, y, i, j - 1)
```

Explanation:

1. In order to write an algorithm that reconstructs an LCS without using the B table, we can use the C table values that we set during the LCS-Length method to find the direction in which we can move.
2. First, we check if we have reached the 0, 0 position in the table. This will be the terminating condition for our recursive statement.
3. Then we check if the current element is one greater than the element to its left diagonal. If this is true, we recursively move to the left diagonal element and print that element.
4. Then we check if the current element is equal to the element to its top. If this is true, we recursively move to the top element.
5. Then finally, if all these conditions are not true, it means that the current element is equal to the element to the left of it. So now we recursively move to the left element.
6. In this way, we only print the diagonal elements during our LCS-Reconstruction process without using the B table.

Question 7:

```
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          $e[i, j] = \infty$
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 $root$
```

This is the algorithm to find out the e and root tables for the given probabilities of keys. Let us trace the execution of this pseudocode for our question.

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

1. Let us first calculate the values for e and w arrays using the q probabilities. The calculation goes as follows:
    a. e[1][0] = q[0] = 0.06, w[1][0] = q[0] = 0.06
    b. e[2][1] = q[1] = 0.06, w[2][1] = q[1] = 0.06

   c. e[3][2] = q[2] = 0.06, w[3][2] = q[2] = 0.06
   d. And so on….
2. Next, let us calculate the root index value and the e value for each iteration of l starting from 1 to n. In each iteration of l, we minimize the e table values and set the root table values accordingly.
   a. For example, the when l = 1:
      i. For i = 1:
         1. j = i + l - 1 = 1
         2. w[1][1] = 0.06 + 0.04 + 0.06 = 0.16
         3. t = 0.06 + 0.06 + 0.16 = 0.28
         4. e[1][1] = 0.28
         5. root[1][1] = 1
      ii. For i = 2:
         1. j = i + l - 1 = 2
         2. w[2][2] = 0.06 + 0.06 + 0.06 = 0.18
         3. t = 0.06 + 0.06 + 0.18 = 0.30
         4. e[2][2] = 0.30
         5. root[2][2] = 2
      iii. This goes on….
   b. For l = 2:
      i. For i = 1:
         1. j = i + l - 1 = 2
         2. t = e[1][0] + e[3][2] + w[1][2]

            = 0.62

         3. e[1][2] = 0.62
         4. root[1][2] = 2
      ii. For i = 2:
         1. j = i + l - 1 = 3
         2. Minimize t = e[2][1] + e[4][3] + w[2][3]

            = 0.68

         3. e[2][3] = 0.68
         4. root[2][3] = 3
      iii. This goes on…..

3. Let us skip to the final values of the e table and the root table:

e table:

	0	1	2	3	4	5	6	7
1	0.06	0.28	0.62	1.02	1.34	1.83	2.44	3.12
2	0.00	0.06	0.30	0.68	0.93	1.41	1.96	2.61
3	0.00	0.00	0.06	0.32	0.57	1.04	1.48	2.13
4	0.00	0.00	0.00	0.06	0.24	0.57	1.01	1.55
5	0.00	0.00	0.00	0.00	0.05	0.30	0.72	1.20
6	0.00	0.00	0.00	0.00	0.00	0.05	0.32	0.78
7	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.34
8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.05

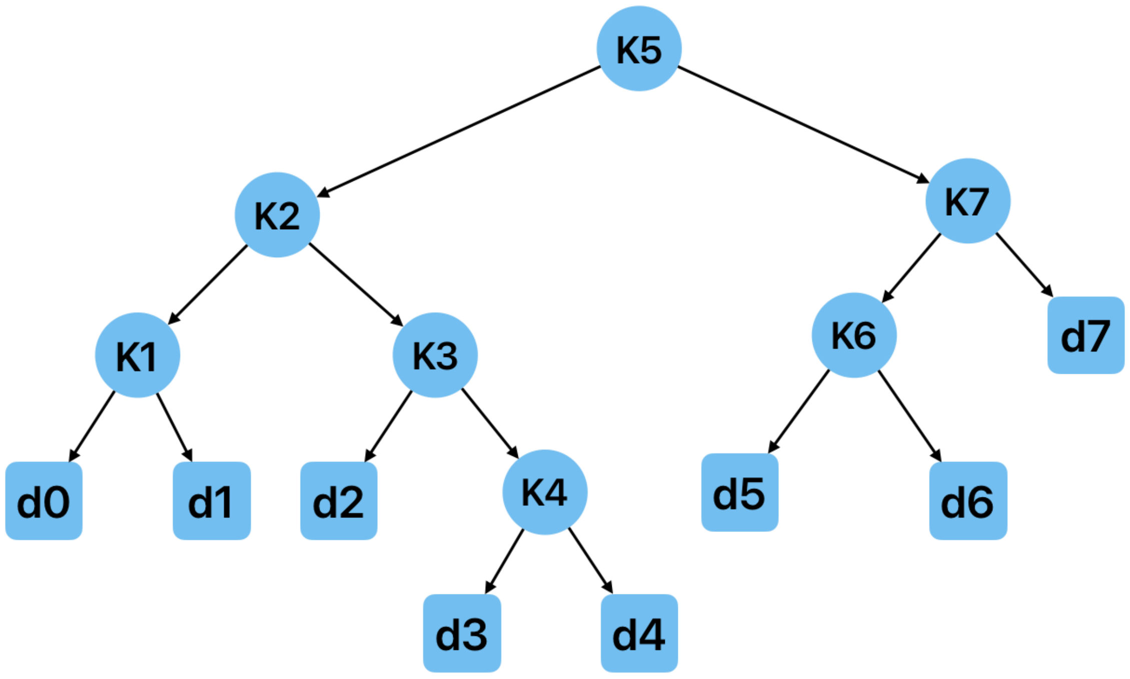
0.06	0.28	0.62	1.02	1.34	1.83	2.44	3.12
0	0.06	0.3	0.68	0.93	1.41	1.96	2.61
0	0	0.06	0.32	0.57	1.04	1.48	2.13
0	0	0	0.06	0.24	0.57	1.01	1.55
0	0	0	0	0.05	0.3	0.72	1.2
0	0	0	0	0	0.05	0.32	0.78
0	0	0	0	0	0	0.05	0.34
0	0	0	0	0	0	0	0.05

root table:

```
     1    2    3    4    5    6    7
1  1.0  2.0  2.0  2.0  3.0  3.0  5.0
2  0.0  2.0  3.0  3.0  3.0  5.0  5.0
3  0.0  0.0  3.0  3.0  4.0  5.0  5.0
4  0.0  0.0  0.0  4.0  5.0  5.0  6.0
5  0.0  0.0  0.0  0.0  5.0  6.0  6.0
6  0.0  0.0  0.0  0.0  0.0  6.0  7.0
7  0.0  0.0  0.0  0.0  0.0  0.0  7.0
```

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

Structure of Optimal BST:



We can infer that the cost of this optimal binary search tree is 3.12.