Module 29: Optimal Binary Search Tree

This module 28 focuses on introducing dynamic programming design strategy and applying it to problems to construct optimal binary search trees. The objectives of this module are

- **To Explain Dynamic Programming**
- **To explain Binary Search Tree**
- **To explain Optimal Binary Search Tree**

Dynamic Programming

Dynamic programming is useful for solving multistage optimization problems, especially sequential decision problems [1,2,3]. Richard Bellman is widely considered as the father of dynamic programming. He was an American mathematician. Richard Bellman is also credited with the coining of the word "Dynamic programming". Here, the word "dynamic" refers to some sort of time reference and "programming" is interpreted as planning or tabulation rather than programming that is encountered in computer programs.

Dynamic programming is used in variety of applications. *Dynamic programming* (DP) is used to solve discrete optimization problems such as scheduling, string-editing, packaging, and inventory management [1,2].

Dynamic programming employs the following steps as shown below:

Step 1: The given problem is divided into a number of subproblems as in "divide and conquer" strategy. But in divide and conquer, the subproblems are independent of each other but in dynamic programming case, there are all overlapping subproblems. A recursive formulation is formed of the given problem.

Step 2: The problem, usually solved in the bottom-up manner. To avoid, repeated computation of multiple overlapping subproblems, a table is created. Whenever a subproblem is solved, then its solution is stored in the table so that in future its solutions can be reused. Then the solutions are combined to solve the overall problem.

Binary Search Tree

A binary search tree is a special kind of binary tree. In binary search tree, the elements in the left and right sub-trees of each node are respectively lesser and greater than the element of that node. Fig. 1 shows a binary search tree.

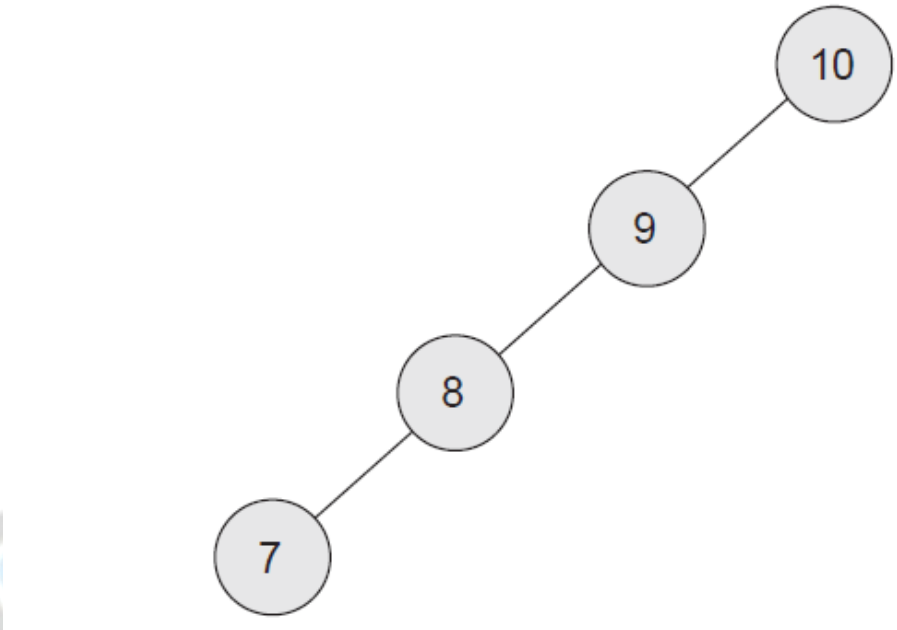


Fig. 1: Skewed Binary Search Tree

Fig. 1 is a binary search tree but is not balanced tree. On the other hand, this is a skewed tree where all the branches on one side.

The advantage of binary search tree is that it facilitates search of a key easily. It takes O(n) to search for a key in a list. Whereas, search tree helps to find an element in logarithmic time.

How an element is searched in binary search tree?

Let us assume the given element is x. Compare x with the root element of the binary tree, if the binary tree is non-empty. If it matches, the element is in the root and the algorithm terminates successfully by returning the address of the root node. If the binary tree is empty, it returns a NULL value. If x is less than the element in the root, the search continues in

the left sub-tree

If x is greater than the element in the root, the search continues

in the right sub-tree.

This is by exploiting the binary search property. There are many applications of binary search trees. One application is construction of dictionary.

There are many ways of constructing the binary search tree. Brute force algorithm is to construct many binary search trees and finding the cost of the tree. How to find cost of the tree? The cost of the tree is obtained by multiplying the probability of the item and the level of the tree. The following example illustrates the way of find this cost of the tree.

Example 1: Find the cost of the tree shown in Fig. 2 where the items probability is given as follows:

a1 = 0.4 , a2= 0.3 , a3 = 0.3

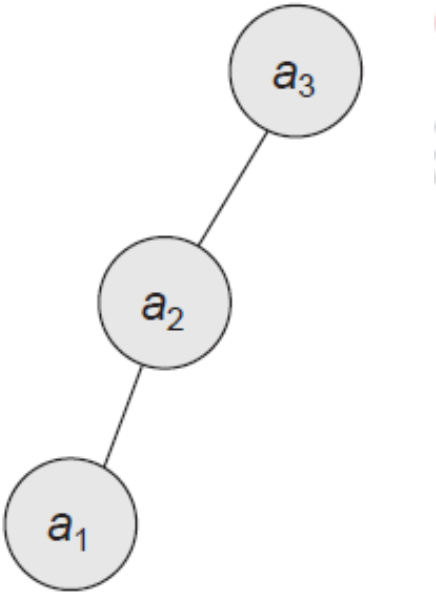


Fig. 2: Sample Binary Search Tree

Solution

As discussed earlier, the cost of the tree is obtained by multiplying the item probability and the level of the tree. The cost of the tree is computed as follows;

$$\text{Cost of BST} = 3(0.4) + 2(0.3) + 1(0.3) = 2.1$$

It can be observed that the cost of the tree is 2.1.

Optimal Binary search Tree

What is an optimal binary search tree? An optimal binary search tree is a tree of optimal cost. This is illustrated in the following example.

Example 2: Construct optimal binary search tree for the three items a1 = 0.4 , a2= 0.3 , a3 = 0.3 ?

Solution

There are many ways one can construct binary search trees. Some of the constructed binary search trees and its cost are shown in Fig. 3.

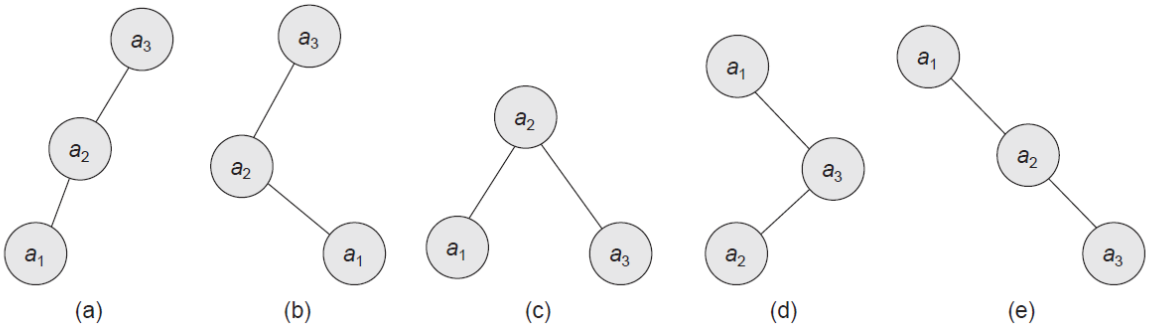


Fig. 3. : Some of the binary search trees

It can be seen the cost of the trees are respectively, 2.1, 1.3, 1.6, 1.9 and 1.9. So the minimum cost is 1.3. Hence, the optimal binary search tree is (b) Fig. 3.

How to construct optimal binary search tree? The problem of optimal binary search tree is given as follows:

Given sequence $K = k_1 < k_2 < \cdots < k_n$ of n sorted keys, with a search probability p_i for each key k_i. The aim is to build a binary search tree with minimum expected cost.

One way is to use brute force method, by exploring all possible ways and finding the expected cost. But the method is not practical as the number of trees possible is Catalan sequence. The Catalan number is given as follows:

$$c(n) = \begin{pmatrix} 2n \\ n \end{pmatrix} \frac{1}{n+1} \text{ for } n > 0, \quad c(0) = 1.$$

If the nodes are 3, then the Catalan number is

$$C_3 = \frac{1}{3+1}\begin{pmatrix} 6 \\ 3 \end{pmatrix} = 5 .$$

Hence, five search trees are possible. In general, $\Omega(4^n/n^{3/2})$ different BSTs are possible with n nodes. Hence, alternative way is to explore dynamic programming approach.

Dynamic programming Approach

The idea is to One of the keys in $a_1, \ldots, a_n$, say a_k, where $1 \leq k \leq n$, must be the root. Then , as per binary search rule, Left subtree of a_k contains $a_1, \ldots, a_{k-1}$ and right subtree of a_k contains $a_{k+1}, \ldots, a_n$.

So, the idea is to examine all candidate roots a_k , for $1 \leq k \leq n$ and determining all optimal BSTs containing $a_1, \ldots, a_{k-1}$ and containing $a_{k+1}, \ldots, a_n$

The informal algorithm for constructing optimal BST based on [1,3] is given as follows:

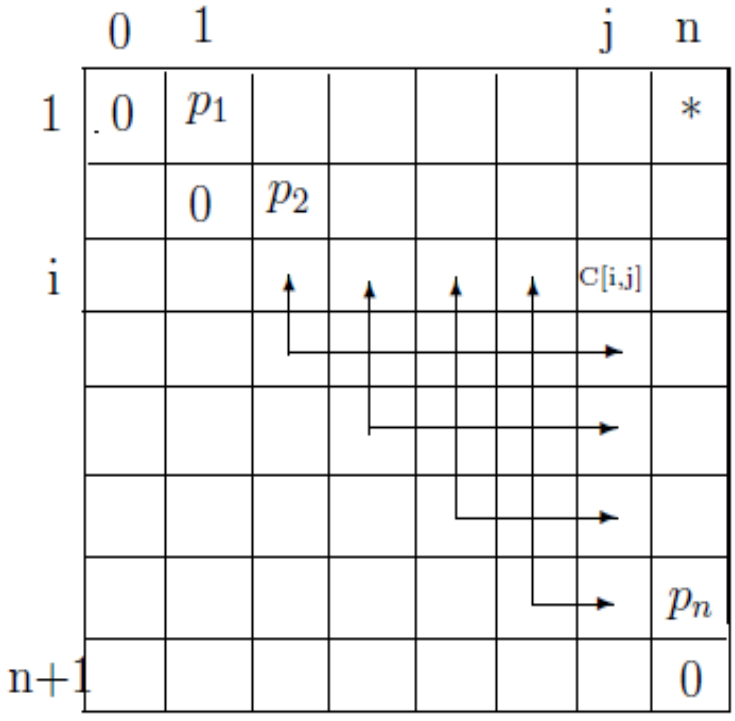
Step 1: Read n symbols with probability p_i.
Step 2: Create the table $C[i, j]$, $1 \leq i \leq j + 1 \leq n$.
Step 3: Set $C[i, i] = p_i$ and $C[i - 1, j] = 0$ for all $i \in [n]$.
Step 4: Recursively compute the following relation:

$$C[i,j] = C[1 \ldots k+1] + C[k+1 \ldots j] + \sum_{m=1}^{n} p_m, \quad \text{for all i and j}$$

Step 5: Return $C[1 \ldots n]$ as the maximum cost of constructing a BST.
Step 6: End.

The idea is to create a table as shown in below [2]

Table 2: Constructed Table for building Optimal BST



The aim of the dynamic programming approach is to fill this table for constructing optimal BST. What should be entry of this table? For example, to compute C[2,3] of two items, say key 2 and key 3, two possible trees are constructed as shown below in Fig. 4 and filling the table with minimum cost.

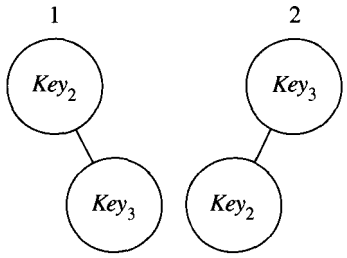


Fig. 4: Two possible ways of BST for key 2 and key 3.

Example 3: Let there be four items *A (Danny), B(Ian), C (Radha) , and D (zee)* with probability 2/7 1/7 3/7 1/7 . Apply dynamic programming approach and construct optimal binary search trees?

Solution

The initial Table is given below in Table 1,

Table 1: Initial table

i \ j	0	1	2	3	4
1	0	2/7			
2		0	1/7		
3			0	3/7	
4				0	1/7
5					0

It can be observed that the table entries are initial probabilities given. Then, using the recursive formula, the remaining entries are calculated.

$$C[1,2] = \min \begin{cases} C[1,0] + C[2,2] + p_1 + p_2 & \text{when } k = 1 \\ C[1,1] + C[3,2] + p_1 + p_2 & \text{when } k = 2 \end{cases}$$

$$C[2,3] = \min \begin{cases} C[2,1] + C[3,3] + p_1 + p_3 & \text{when } k = 2 \\ C[2,2] + C[4,3] + p_1 + p_3 & \text{when } k = 3 \end{cases}$$

$$C[3,4] = \min \begin{cases} C[3,2] + C[4,4] + p_3 + p_4 & \text{when } k = 3 \\ C[3,3] + C[5,4] + p_3 + p_4 & \text{when } k = 4 \end{cases}$$

The updated entries are shown below in Table 2.

Table 2: Updated table

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

Similarly, the other entries are obtained as follows:

$$C[1,3] = \min \begin{cases} C[1,0] + C[2,3] + p_1 + p_2 + p_3, & \text{when } i = 1, j = 3, k = 1 \\ C[1,1] + C[3,3] + p_1 + p_2 + p_3, & \text{when } i = 1, j = 3, k = 2 \\ C[1,2] + C[4,3] + p_1 + p_2 + p_3, & \text{when } i = 1, j = 3, k = 3 \end{cases}$$

$$= \min\{{}^{12}/_{7}, {}^{11}/_{7}, {}^{10}/_{7}\} = {}^{10}/_{7}$$

$$C[2,4] = \min \begin{cases} C[2,3] + C[5,4] + p_2 + p_3 + p_4, \text{when } k = 2 \\ C[2,2] + C[4,4] + p_2 + p_3 + p_4, \text{when } k = 3 \\ C[2,3] + C[5,4] + p_2 + p_3 + p_4 \text{ when } k = 4 \end{cases}$$

$$= \min\{{}^{11}/_{7}, {}^{7}/_{7}, {}^{11}/_{7}\} = {}^{7}/_{7}$$

The updated table is given in Table 3.

Table 3: Updated table

	0	1	2	3	4
1	0	2/7	4/7	10/7	
2		0	1/7	6/7	7/7
3			0	3/7	5/7
4				0	1/7
5					0

The procedure is continued as

$$C[1,4] = \min \begin{cases} C[1,0] + C[2,4] + P_1 + P_2 + P_3 + P_4, \text{ when } k = 1 \\ C[1,1] + C[3,4] + P_1 + P_2 + P_3 + P_4, \text{ when } k = 2 \\ C[1,2] + C[4,4] + P1 + P_2 + P_3 + P_4, \text{ when } k = 3 \\ C[1,3] + C[4,4] + P_1 + P_2 + P_3 + P_4 \} \text{when } k = 4 \end{cases}$$

$$= \min\{{}^{14}/_{7}, {}^{14}/_{7}, {}^{12}/_{7}, {}^{18}/_{7}\} = {}^{12}/_{7}$$

The updated final table is given as shown in Table 4.

Table 4: Final table

	0	1	2	3	4
1	0	2/7	4/7	10/7	12/7
2		0	1/7	6/7	7/7
3			0	3/7	5/7
4				0	1/7
5					0

It can be observed that minimum cost is 12/7. What about the tree structure? This can be reconstructed by noting the minimum k in another table as shown in Table 5.

Table 5: Minimum k

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

It ca be seen from the table 5 that C(1,4) is 3. So the item 3 is root of the tree. Continuing this fashion, one can find the binary search tree as shown in Fig. 5.

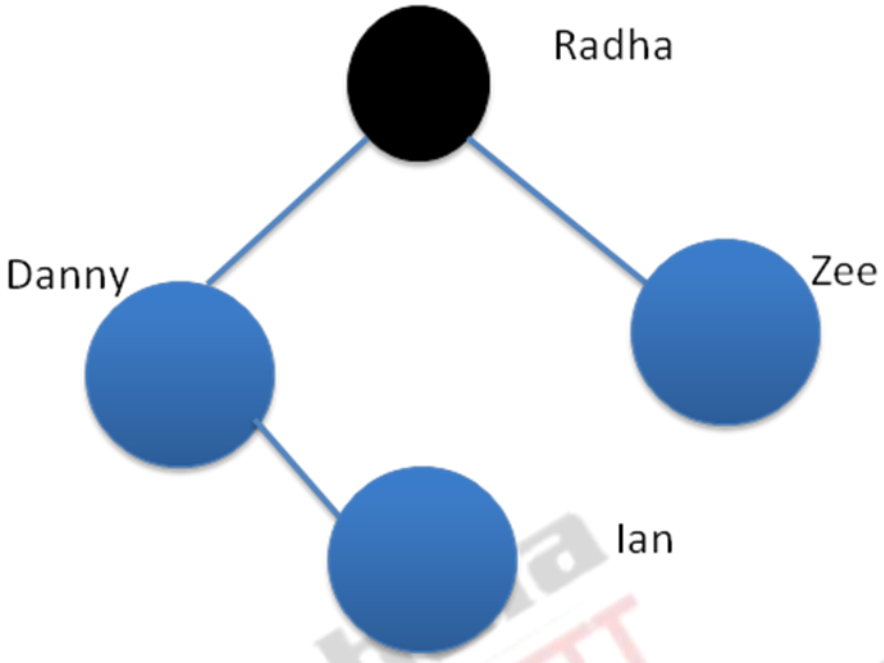


Fig. 5: Constructed Optimal BST

It can be seen that the constructed binary tree is optimal and balanced.

The formal algorithm for constructing optimal BST is given as follows:

```
%% Initialize the table for C
for i = 1 to n do
    C[i, i - 1] = 0
    C[i, i] = pi
End for
C[n + 1,n]= 0
```

```
%% Initialize the table for R
for i = 1 to n do
    R[i, i - 1] = 0
    R[i, i] = i
End for
C[n + 1, n] = 0
```

```
for diag = 1 to n - 1 do
    for i = 1 to n - diag do
        j = i + diag
```

$$C[i,j] = \min_{i<k\leq j} C[1\ldots k-1] + C[k+1\ldots j] + \sum_{m=1}^{n} p_m$$

```
        R[i,j] = k
    End for
End for
Return C[1,n]
```

Complexity Analysis

The time efficiency is Θ (n^3) but can be reduced to Θ(n^2) by taking advantage of monotonic property of the entries. The monotonic property is that the entry $R[i,j]$ is always in the range between $R[i,j-1]$ and $R[i+1,j]$. The space complexity is Θ(n^2) as the algorithm is reduced to filling the table.

Summary

In short, one can conclude as part of this module 28 that

- Dynamic Programming is an effective technique.
- DP can solve LCS problem effectively.
- Edit Distance Problem can be solved effectively by DP approach.

References:

1. *S.Sridhar , Design and Analysis of Algorithms , Oxford University Press, 2014.*
2. *A.Levitin, Introduction to the Design and Analysis of Algorithms, Pearson Education, New Delhi, 2012.*
3. T.H.Cormen, C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA 1992.