0810976

Optimal binary search trees

Abstract/interpretation:

A binary search tree or ordered or sorted binary tree, is a rooted binary tree whose internal nodes each store a key greater than all the keys in the node's left subtree and less than those in its right subtree. A binary tree is a type of data structure for storing data such as numbers in an organized way. Binary search trees allow binary search for fast lookup, addition and removal of data items, and can be used to implement dynamic sets and lookup tables. The order of nodes in a BST means that each comparison skips about half of the remaining tree, so the whole lookup takes time proportional to the binary logarithm of the number of items stored in the tree. This algorithm is implemented using dynamic programming.

The basic idea of dynamic programming is to solve a problem by breaking it up into smaller subproblems and reusing solutions to subproblems.
An important step in designing a dynamic programming algorithm is to figure out the recursive structure of the underlying problem. Typically, this involves three steps:

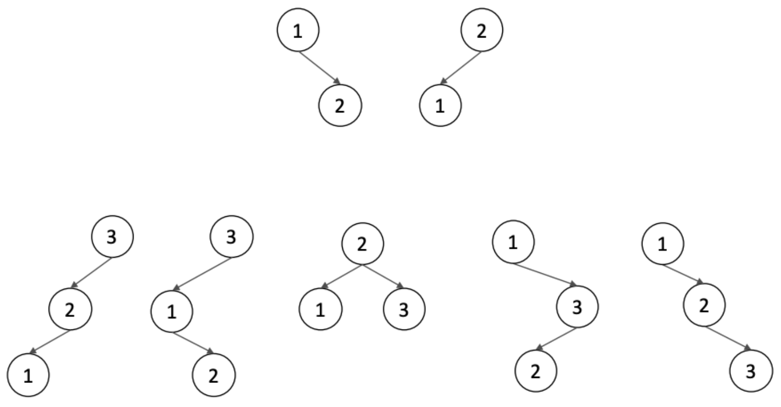
1. Identify all the possible options for the “first” choice;
2. Conditioned on the first choice, find the optimal solution;
3. Take the first choice that leads to the overall best solution. Next,I will explain how I go about to implement the optimal BST problem.

Properties:

1. Solution to the original problem can be computed from solutions to (independent) subproblems. 2. There are polynomial number of subproblems.
3. There is an ordering of the subproblems such that the solution to a subproblem depends only on solutions to subproblems that precede it in this order.

Defining the sub-problems:

As can be seen in the picture, the orientation of the nodes goes from higher values to lower ones from the parent to the left child, and lower values of the node to higher ones when we go from the parent to the right child



It can also be noticed that there’s a recurrence in this particular pattern.

$$f(n) = \sum_{i=0}^{n-1} f(i) \cdot f(n-i-1)$$

The function above defined this particular behavior,

Since the function is defined in recursive form, it is necessary to give base case values:

The base cases given here are also the values of B(0), B(1), B(2).
Where B(i) represent each node and its associated value.

Naive analysis.

In the naive implementation of the algorithm, we can analyze the runtime with the recurrence tree written below.

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} T(i) + n \\ &\geq T(n-1) + T(n-2) \\ &\geq F(n) \\ &= 2^{\Theta(n)} \end{aligned}$$

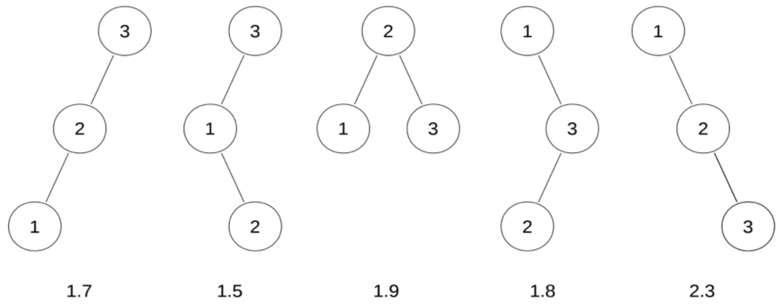
Where F(n) represents the Fibonacci sequence.

Optimal Binary Search Trees

Suppose we are given a list of keys k1 < k2 < . . . < kn, and a list of probabilities pi that each key will be looked up. An optimal binary search tree is a BST T that minimizes the expected search time

$$\sum_{i=1}^{n} p_i(\mathrm{depth}_T(k_i) + 1).$$

For n = 3, there are 5 possible BSTs. The following figure enumerates them and their corresponding expected search times. The optimal BST for the given input is the second tree.



In general, there are Θ(4^n*n^ −3/2) binary trees, so we cannot enumerate them all. By using dynamic programming, however, we can solve the problem efficiently.

Runtime

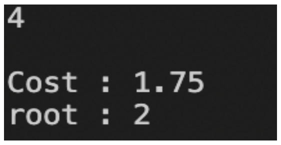
There are a total of n 2 subproblems, and each subproblem takes O(n) time to compute, assuming all its subproblems are already solved. Thus, the total running time is O(n 3).

By the theorem, for each subproblem we can restrict the possible values of k to r(i + 1, j) − r(i, j − 1) + 1 choices instead of trying all i ≤ k ≤ j. For this case, the total runtime is

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=1}^{n} r(i+1,j) - r(i,j-1) + 1 &= n^2 + \sum_{i=1}^{n}\sum_{j=1}^{n} r(i+1,j) - \sum_{i=1}^{n}\sum_{j=1}^{n} r(i,j-1) \\
&= n^2 + \sum_{i=2}^{n+1}\sum_{j=1}^{n} r(i,j) - \sum_{i=1}^{n}\sum_{j=0}^{n-1} r(i,j) \\
&\le n^2 + \sum_{i=1}^{n+1}\sum_{j=1}^{n} r(i,j) - \sum_{i=1}^{n}\sum_{j=1}^{n-1} r(i,j) \\
&= n^2 + \left(\sum_{i=1}^{n}\sum_{j=1}^{n} r(i,j) + \sum_{j=1}^{n} r(n+1,j)\right) \\
&\qquad - \left(\sum_{i=1}^{n}\sum_{j=1}^{n} r(i,j) - \sum_{i=1}^{n} r(i,n)\right) \\
&= n^2 + \sum_{j=1}^{n} r(n+1,j) + \sum_{i=1}^{n} r(i,n) \\
&= O(n^2)
\end{aligned}$$

Simulations:

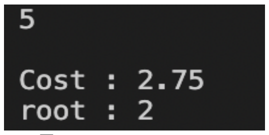
n=4:



```
4

Cost : 1.75
root : 2
```

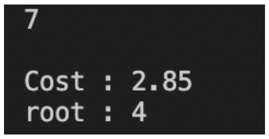
n=5:



```
5

Cost : 2.75
root : 2
```

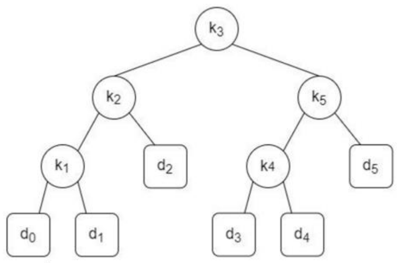
n=7

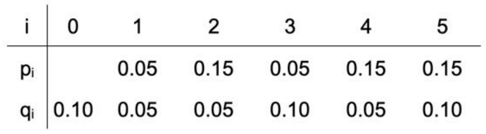


```
7

Cost : 2.85
root : 4
```

Discussion:





i	0	1	2	3	4	5
p_i		0.05	0.15	0.05	0.15	0.15
q_i	0.10	0.05	0.05	0.10	0.05	0.10

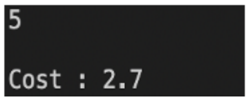
1. What are the Optimal Binary Search Tree and Smallest Expected Search Cost?

- Please calculate the Search Cost of this Binary Search Tree, and is it an Optimal Binary Search Tree? If it is not,
- please draw (or print out result by the program)
- What is the Smallest Expected Search Cost?

The binary search tree is an optimal binary search tree as it provide the smallest expected search time.

To calculate the search cost for this BST we can use the formula in the textbook

$$\begin{aligned} \text{E}\,[\text{search cost in } T] &= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i \,, \qquad (15.11) \end{aligned}$$



```
5

Cost : 2.7
```

The expected search cost for n=5 is 2.7

2. Why do you need to use a Table to store the value of w, please describe your thoughts.

BSTs do not reserve more memory than they need to. In order to work so it has to keep track of already computed elements.
As BST preserves the order information, it provides us with four additional dynamic set operations These operations are:

- Maximum
- Minimum
- Successor
- Predecessor

All these operations like every BST operation have time complexity of O(n). Additionally all the stored keys remain sorted in the BST thus enabling us to get the sorted sequence of keys just by traversing the tree in in-order.