Optimal BST

Frank

October 21, 2021

1 Optimal BSTs

1.1 Introduction

n records with keys $1, 2, \ldots, n$ that we want to store in some dictionary and periodically search for them.

Suppose our dictionary is static (no insert or deletes), then let $P(i)$ be the probability of searching for the record with key i

Then we want to place those records such that we minimize the expected time to find an element.

1.2 Linked List Implementation

- Given linked list L, cost of the linked list

$$c(L) = \sum_{1 \leq i \leq n} P(i) \cdot pos(i, L)$$

  Note there are $n!$ ways we can order this list, and $pos(i, L)$ is the position of i in L

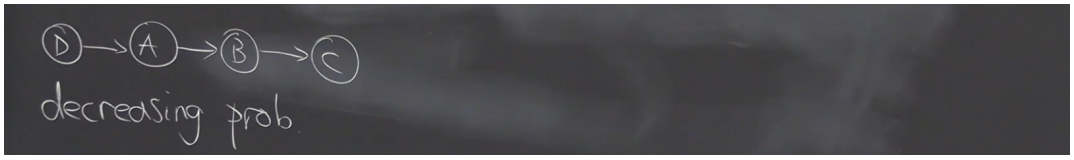
- Want to find L that minimizes $c(L)$

  By a greedy algorithm, simply arrange the records in non-increasing $P(i)$ and prove by any greedy algorithm proof technique.

- As an example, given following records with probabilies:

A	0.25
B	0.23
C	0.22
D	0.30

  We would have the following ordering



1.3 BST Implementation

- ***Input***: Given a BST T, then cost of the tree is

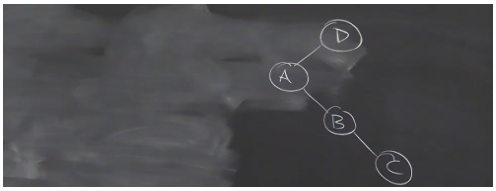
$$c(T) = \sum_{1 \le i \le n} P(i) \cdot (1 + depth(i, T))$$

  Where $depth(i, T)$ is the depth of node i in T (note that depth is the number of edges so we add 1)

- ***Output***: Find a BST T that minimizes $c(T)$
- Note that the number of trees with n nodes is

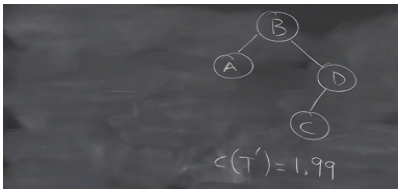
$$\frac{\binom{2n}{n}}{n+1}$$

- A greedy approach would be to place higher probability records closer to the root.



  but this is not the optimal tree. $c(T) = 2.37$

- This is the optimal tree, note that the BST property is maintained by nodes, not its probability.



  Cost of this tree is $c(T) = 1.99$

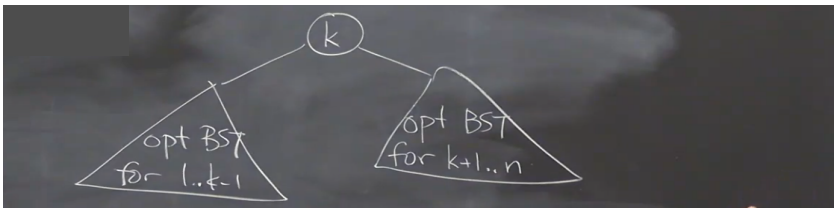
  This is a problem that cannot be solved by a greedy algorithm, so we will use DP

1.4 DP Algorithm

Want to find an optimal BST

- First step, start with an optimal T.

  Suppose T is optimal, with record k at its root, then we would have



However, suppose that the left and right trees are optimal for their respective set of nodes, how can we be sure that once we extend the solution, we still have an optimal cost?

Usually, the cost of a structure is the sum of its substructure plus whatever it extends, but this isn't the case here.

In this case, by cut-and-paste argument, if either the left or right subtree is not optimal, then we can get an optimal subtree that will decrease the cost of the entire tree, implying that T is not optimal. Thus the subtrees must be optimal. But how do we verify this? Consider the relationship between the cost of T and the cost of its subtrees:

$$c(T) = c(T_L) + c(T_R) + \sum_{u \in T} P(u)$$

Proof:

$$\begin{aligned}
c(T) &= \sum_{u \in T} P(u) \cdot (1 + depth(u, T)) \\
&= P(k)(1+0) + \sum_{u \in T_L} P(u) \cdot (1 + depth(u, T)) + \sum_{u \in T_R} P(u) \cdot (1 + depth(u, T)) \\
&= \sum_{u \in T} P(u) + \sum_{u \in T_L} P(u) \cdot depth(u, T) + \sum_{u \in T_R} P(u) \cdot depth(u, T) \\
&= \sum_{u \in T} P(u) + \sum_{u \in T_L} P(u) \cdot (1 + depth(u, T_L)) + \sum_{u \in T_R} P(u) \cdot (1 + depth(u, T_R)) \\
&= c(T_L) + c(T_R) + \sum_{u \in T} P(u)
\end{aligned}$$

By verifying this, we have discovered the optimal substructure.

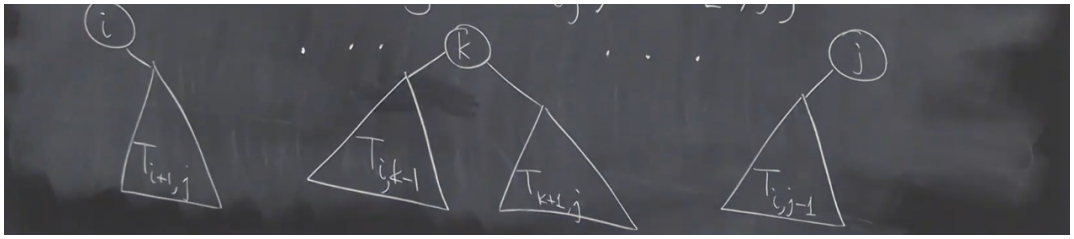
1.5 Subproblems

Find optimal BST T_{ij} for keys $i..j$, where $1 \leq i \leq j \leq n$, note we are looking through $O(n^2)$ key pairs for our interval and not n intervals. This is because when we recurse from our subtree, one of the ranges is not going to start from 1. So we need all $O(n^2)$ subproblems, and we need to find the optimal BST for all $O(n^2)$ problems.

For now, focus on finding the **cost**, $c(T_{ij})$ for all sub problems

$$c(T_{ij}) = C[i,j] \quad (\star)$$

Note that our T_{ij} looks like



Where $k \in [i,j]$, so we need to try all cases to see which one gives us the minimum cost.

Now we need to find (†) to compute ($\star$), which is $C[i,j]$, for $1 \leq i \leq j \leq n$. Then we can use ($\star$) to find the solution of $c(T)$ by computing $C[1,n]$.

$$C[i,j] = min_{i \leq k \leq j}(C[i,k-1] + C[k+1,j]) + \sum_{i \leq u \leq j} P(u) \quad (\dagger)$$

- Extreme Cases
  - $k = i$, then we have $C[i,k-1] = C[i,i-1]$, which is the empty left subtree.
  - $k = j$, then we have $C[k+1,j] = C[j+1,j]$, which is the empty right subtree

Note that the above two cases is outside of the bounds in which we call subproblems (we defined $i \leq j$ for $C[i,j]$).

But that is okay, we will treat these cases as subproblems also:

$$C[i,i-1] = C[j+1,j] = 0$$

Thus, we will redefine our subproblem:

$$\text{Compute } C[i,j] \text{ for } 1 \leq i \leq (n+1), 0 \leq j \leq n, i \leq (j+1) \quad (\star)$$

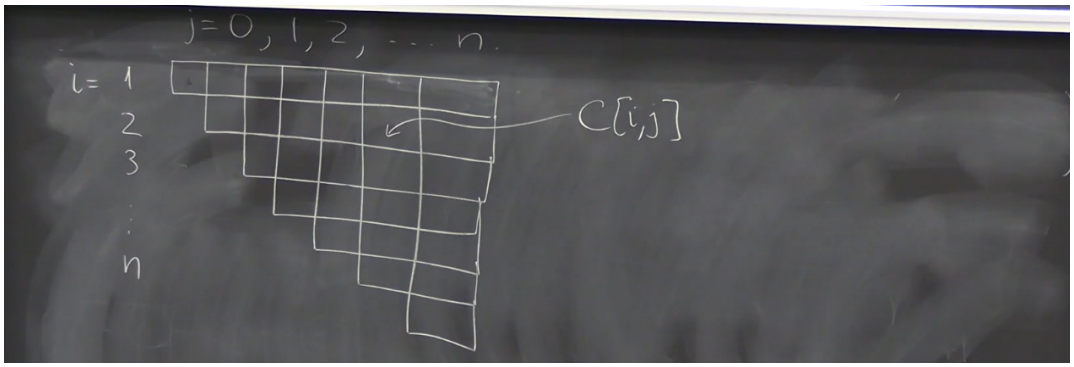
to accommodate the extreme cases.

Then

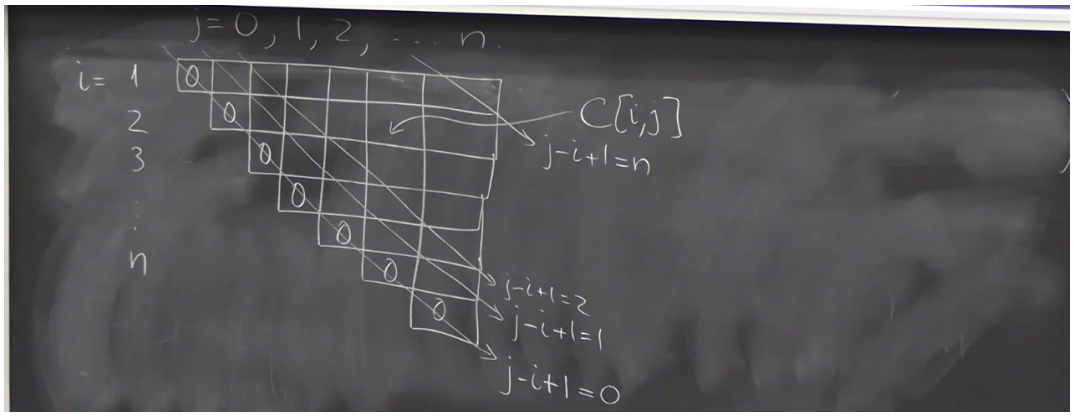
$$C[i,j] = \begin{cases} 0 & j - i + 1 = 0 \text{ (no nodes)} \\ min_{i \leq k \leq j}(C[i,k-1] + C[k+1,j]) & \\ + \sum_{i \leq u \leq j} P(u) & j - i + 1 > 0 \text{ (nodes in tree)} \end{cases} \quad (\dagger)$$

1.6 Visualization for subproblem

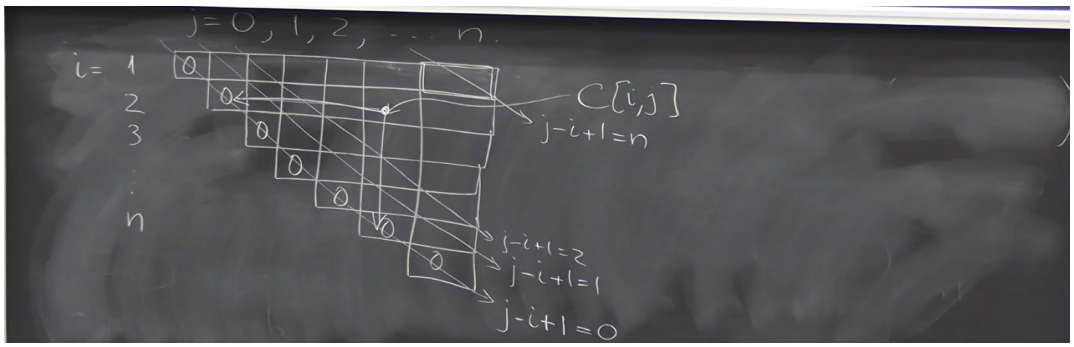
Visualization of all the subproblems we want to solve



Then note that all of these subproblems have a pattern regarding the number of nodes in these trees. And we start from the 0 nodes tree, and compute towards the tree that has n nodes



Note then that for any generic problem for $C[i, j]$, we look to its left for the left subtree, and down for its right subtree.



1.7 Algorithm

```
1:  procedure OPTBST(P[1..n])
2:      S[0] ← 0
3:      for (i = 1..n) do
4:          S[i] ← S[i − 1] + P[i]
5:      for (i = 1..n) do
6:          C[i, i − 1] ← 0
7:      for (d = 0..(n − 1)) do
8:          for (i = 1..(n − d)) do
9:              j ← i + d
10:             C[i, j] ← ∞
11:             for (k = i..j) do
12:                 if (C[i, j] > C[i, k − 1] + C[k + 1, j] + S[j] − S[i − 1]) then
13:                     C[i, j] ← C[i, k − 1] + C[k + 1, j] + S[j] − S[i − 1]
14:     return C[1, n]
```

- Note that lines 2-4 allows for us to use

$$\sum_{i \leq u \leq j} P(u) = S[j] - S[i-1]$$

  Which allows us to quickly compute $\sum_{i \leq u \leq j} P(u)$

- Note that line 5-6 solves the smallest subproblems $(i + j - 1 = 0)$

1.8 Running Time

$\Theta(n^3)$, since we are solving for $O(n^2)$ subproblems, and for each sub problem, we have to do a linear amount of work (trying out all values of k from i to j)

1.9 Reflection

How to retrofit the algorithm to solve for the optimal BST instead of the cost of the optimal BST

Hint: As you're finding a better and better k, you know a better and better root for that subtree, keep track of that.