CSci 4041: Algorithms and Data Structures
Optimal Binary Search Trees (OBST's)
Part 3

Last revision March 30, 2020

Here's the agenda for this lecture.

Computing the expected cost of a BST.
Computing the cost of a single node.
Computing the cost of many nodes.
How to choose the root.
An efficient algorithm for making OBST's.

In the previous lecture, we saw a recursive procedure for constructing an optimal binary search tree. However, it couldn't decide what the root of the tree should be, so it either had to guess the root somehow, or else it had to choose the root in all possible ways. In this lecture we'll show how to choose the root so as to minimize the expected cost of the tree. To do that, we'll need to compute the cost in a different way than we saw before. We'll have all we need for an efficient procedure that makes OBST's. This is (finally!) the last OBST lecture for the course.

The cost of a single node. We'll start small. What's the expected cost of exactly one node? Suppose we have a tree T with n nodes. We saw in an earlier lecture that $E[S(T)]$, the expected cost of searching T, is this:

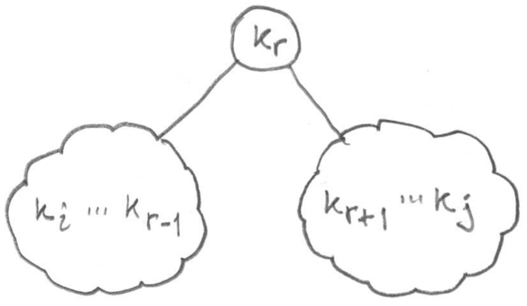
$$E[S(T)] \;=\; \sum_{i=1}^{n}(\mathit{depth}_T(k_i)+1)\;p_i + \sum_{i=0}^{n}(\mathit{depth}_T(d_i)+1)\;q_i$$

This suggests that the expected cost of a single node—whether it's a success node or a failure node—should be just its depth plus 1, times its probability.

Cormen computes depths by counting links, not nodes, so $\mathit{depth}(x)+1$ is the number of nodes on the path to the node with key x. In a BST, we find a node by following a path to it, starting at the root, and making key comparisons as we go. As a result, $\mathit{depth}(x)+1$ is the number of key comparisons needed to find the node with key x. It makes sense that the cost of a node should be the number of comparisons needed to find it in a tree.

Now suppose we have a node whose key is x, and whose probability is u. Its expected cost must be $(\mathit{depth}(x)+1)\;u$. But since we have only one node, its depth must be 0. So the expected cost of the node is $(0+1)\;u = u$. In other words, the expected cost of a single node is just its probability.

The cost of many nodes. What if we have many nodes? Suppose we have a series of keys $k_i \cdots, k_j$ in ascending order, where $i \geq 1$ and $i-1 \leq j \leq n$. Also suppose that we want to construct a BST by choosing one of those keys k_r to be at the root. Then the keys in the root's left subtree will be $k_i \cdots, k_{r-1}$, and the keys in the root's right subtree will be $k_{r+1} \cdots, k_j$, as shown in the sketch below.



If we can turn the keys (shown as mysterious clouds) into BST's, then we'll get a BST containing all the keys. Unfortunately, we don't know how to choose k_r yet. We'll work on fixing that problem now.

We'll deal with a couple of edge cases first. Suppose that we choose $r = i$, so that the first key k_i is the root. Then the left subtree has keys $k_i \cdots, k_{i-1}$. Since there are no keys between k_i and k_{i-1}, we'll assume that the failure key d_0 is in the left subtree instead.

Similarly, suppose that we choose $r = j$, so that the last key k_j is the root. Then the right subtree has keys $k_{j+1} \cdots, k_j$. There are no keys between k_{j+1} and k_j either, so we'll assume that the failure key d_j is in the right subtree.

Eventually we'll want to choose k_r in a way that minimizes the cost of the entire tree. So what's the cost of the BST shown above? It has three pieces: a root node, a left subtree, and a right subtree. To determine its cost, we'll add up the costs of all three pieces. The cost of the root node, with key k_r, is just its probability p_r, because it's a single node.

What are the cost of its left and right subtrees? According to the definition of $E[S(T)]$, the cost of a node is its depth plus one, times the probability of its key. But we've increased the depths of each node in the left and right subtrees by one again, because we've placed the node containing k_r above them. That means we've increased the cost of each subtree by the sum of the probabilities of all its keys.

We'll write an expression for that. Let $w(i, j)$ be the sum of all the probabilities in a tree containing the success nodes $k_i \cdots, k_j$, and the failure nodes $d_{i-1} \cdots, d_j$. (I guess Cormen called it w to suggest the word *weight*.)

$$w(i, j) \ = \ \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

Now let $e[i, j]$ be the cost of a tree containing those same nodes. (Cormen is inconsistent with brackets and parentheses—sometimes he uses one, and sometimes the other, without apparent reason.)

$$e[i, j] \ = \ (e[i, r-1] + w(i, r-1)) + p_r + (e[r+1, j] + w(r+1, j))$$

This says that $e[i, j]$ is the sum of all these things: $e[i, r-1]$, the cost of the left subtree by itself, plus $w(i, r-1)$, the increased cost of the left subtree, plus p_r, the cost of the root, plus $e[r+1, j]$, the cost of the right subtree by itself, plus $w(r+1, j)$, the increased cost of the right subtree.

Let's simplify this. We'll move the terms around so the e's are together, and the w's are together with p_r. Of course addition associates and commutes.

$$e[i, j] \ = \ (e[i, r-1] + e[r+1, j]) + (w(i, r-1) + p_r + w(r+1, j))$$

But the term $(w(i, r-1) + p_r + w(r+1, j))$ is the sum of the probabilities in the entire BST, $w(i, j)$. So we can rewrite $e[i, j]$ like so.

$$e[i, j] \ = \ e[i, r-1] + e[r+1, j] + w(i, j)$$

This tells us the cost of a BST with success keys $k_i \cdots, k_j$ is the cost of its left subtree $e[i, r-1]$, plus the cost of its right subtree, $e[r+1, j]$, plus the sum of all the probabilities in the entire BST, $w(i, j)$.

How to choose the root. We now know how to compute the cost of a BST. We can use this cost to help choose the root node k_r. We'll choose r, the subscript of the root key k_r, so that it minimizes the cost of the resulting BST. This gives us the following definition for $e[i, j]$.

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j \ = \ i - 1 \\ \min\limits_{1 \le r \le j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \le j \end{cases}$$

Previous definitions of $e[i, j]$ just assumed we knew r, the subscript of the root node. This definition is better because it computes r for itself. It's a recursion that uses some ideas from CURSY-OPTIMAL-BST in a previous lecture.

The base case of the recursion occurs when we try to compute $e[i, i-1]$. As we saw earlier, this asks for the cost of a subtree with no success keys in it—it has only the failure key d_{i-1}, so its cost is the failure key's probability, q_{i-1}. In the recursive case, we try all possible subscripts r until we find one that minimizes the cost of the entire BST. If we modified this recursion so that it recorded all its choices of r, then we could use those choices to construct an OBST.

An efficient algorithm. We can do this efficiently by working "bottom up," starting with the external nodes on the fringe of the BST, and assembling them into little OBST's as we ascend the tree toward the root. At all times, we compute parts of the BST from other, smaller parts that we've already computed. When we reach the root node, we will have computed the entire OBST.

The procedure OPTIMAL-BST (Cormen 402) does this without recursion. It takes three parameters: the probabilities of the success keys p, the probabilities of the failure keys q, and the number of success keys n. Cormen treats p and q like arrays whose elements are accessed by subscripts. OPTIMAL-BST also uses three two-dimensional arrays (page 401–402), called e, w, and *root*.

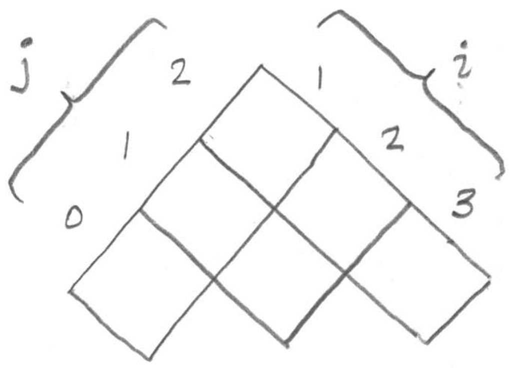
The array $e[1 \mathinner{..} n+1,\ 0 \mathinner{..} n]$ holds the costs of subtrees, what we called $e[i, j]$ above. The first index has an upper bound of $n + 1$ instead of n, because to have a subtree containing only the failure key d_n, we need $e[n + 1, n]$. The second index has a lower bound of 0 instead of 1, because to have a subtree containing only the failure key d_0, we need $e[1, 0]$. We use only the elements of $e[i, j]$ for which $i \geq j - 1$.

The array $w[1 \mathinner{..} n,\ 0 \mathinner{..} n]$ stores sums of probabilities, what we called $w(i, j)$ above. It's used only for efficiency, so we don't have to compute the sums over and over again. Its indexes are used the same way as those of the array e.

The array *root* records the OBST that is built by OPTIMAL-BST. We saw how it worked in an earlier lecture: the element $root[i, j]$ holds the index of the key at the root of the tree with success keys $k_i \cdots, k_j$. We use only the elements of $root[i, j]$ where $1 \leq i \leq j \leq n$.

I won't discuss OPTIMAL-BST in detail; I'll just walk through it and try to explain how each of its parts work. To do that, I'll use a small example where $n = 2$. There will therefore be two success nodes k_1, k_2, and three failure nodes d_0, d_1, d_2.

Cormen writes the arrays e, w and *root* tilted on their sides, so that e and w will look like this when $n = 2$. The array *root* will look much the same, but it's missing the bottom row of elements.

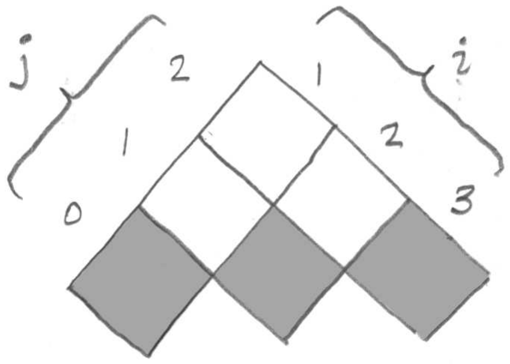


The arrays are tilted because OPTIMAL-BST will work "bottom up," filling in values for the bottom row of elements, then the middle row, and finally the single element at the top. Cormen thinks tilting the array on its side makes this more apparent. Maybe it does.

Here's the procedure OPTIMAL-BST.

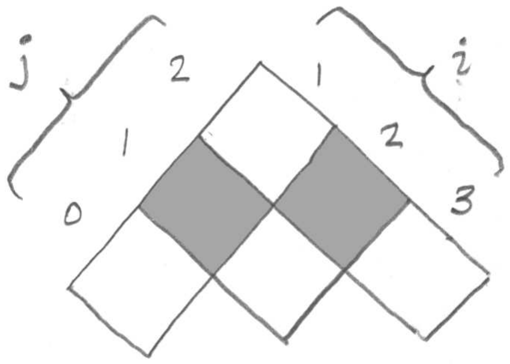
```
01  OPTIMAL-BST($p$, $q$, $n$)
02      for $i = 1$ to $n + 1$
03          $e[i, i-1] = q_{i-1}$
04          $w[i, i-1] = q_{i-1}$
05      for $l = 1$ to $n$
06          for $i = 1$ to $n - l + 1$
07              $j = i + l - 1$
08              $e[i, j] = \infty$
09              $w[i, j] = w[i, j-1] + p_j + q_j$
10              for $r = 1$ 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 $e, root$
```

The first loop in lines 02–04 is for initialization. Recall that all the external nodes of an OBST have failure keys, so for i between 1 and $n + 1$, we set $e[i, i-1]$ and $w[i, i-1]$ to the probabilities of those failure keys. This corresponds to the base case in the recursive definition of $e[i, j]$ that we saw earlier. The loop fills in values for the bottom row of tilted arrays e and w, which are shown here shaded.

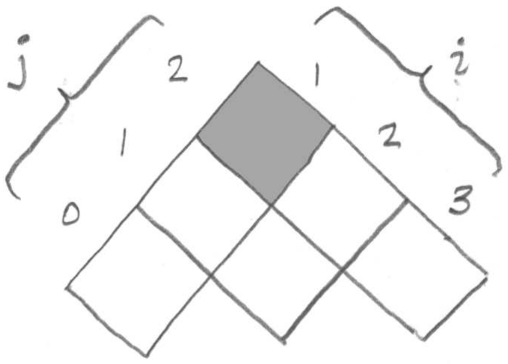


With initialization complete, the procedure executes two nested loops with an assignment statement inside, at lines 05–07. Together, these three lines generate values for i and j that are used to schedule the rest of the computation. The variable l appears nowhere else; it's just a temporary that helps compute i and j.

Computation starts at the bottom of the arrays e and w, then works its way up; lines 05–07 set i and j to indexes that reflect this. They first set $i = 1$ and $j = 1$. Then they set $i = 2$ and $j = 2$. These indexes select the shaded elements shown below in e and w.



Finally, lines 05–07 set $i = 1$ and $j = 2$. These indexes select the shaded element at the top of e and j.



Now let's look at the rest of the code inside the two nested loops. Line 08 sets $e[i, j]$ to ∞. This suggests that remaining lines will compute several possible values for $e[i, j]$ and keep the minimum value. (We saw a similar use of ∞ in a pre-viral lecture when we discussed a procedure to find the minimum element of an array.)

Line 09 sets $w[i, j]$ to the sum of the probabilities in the tree containing success keys k_i through k_j. To save time, this is computed by simply adding two new probabilities p_j and q_j to a sum of probabilities $w[i, j-1]$ that was computed earlier. Remember, we're computing everything in terms of things we've already computed.

Line 10 introduces a loop that generates possible choices for the root subscript r, between i and j. For each choice of r, line 11 computes the cost t of a BST with that root. Note that the cost is computed using $e[i, r-1]$, $e[r+1, j]$, and $w[i, j]$, all of which were computed previously.

At line 12, we test if the cost t is less than the previous cost $e[i, j]$. If it is, then we update $e[i, j]$ at line 13, and record the choice of root at line 14. If it isn't, then we do nothing. This part of the procedure actually finds a minimum cost subtree. (Again, it's similar to the procedure that finds the minimum element of an array that I mentioned a couple of paragraphs ago.)

Finally, when all the loops have finished running, line 16 returns two values: the arrays e and *root*. The array *root* is really all we need, because it contains all the information needed to reconstruct the tree. I guess Cormen decided to return e because it contains enough information to show that the tree in *root* is actually an OBST. It might be used to convince skeptics.

How fast does OPTIMAL-BST run? We can measure its run time in terms of the number of times its loops are executed. The procedure has three nested loops, each of which run at most n times, so it's easy to see that its run time must have an upper bound $O(n^3)$. On page 403, Cormen appeals to an analysis of another algorithm (which we don't care about for this course) to show that OPTIMAL-BST's run time has a lower bound $\Omega(n^3)$. Since we have both a lower and an upper bound, we conclude that the procedure has an exact run time $\Theta(n^3)$.

All we've done in this lecture is skim OPTIMAL-BST. If I were giving this lecture in person, I'd do a very small example on the whiteboard, in which I'd walk through the algorithm step by step. That doesn't work well in writing, so enthusiastic students may want to try it themselves. As I said earlier, my example has $n = 2$. Its success probabilities are $p_1 = 0.5$ and $p_2 = 0.2$. Its failure probabilities are $q_0 = 0.1$, $q_1 = 0.1$, and $q_2 = 0.1$. (These probabilities should sum to 1.0.) I'll leave it up to you whether node k_1 or k_2 ends up at the root of the OBST.