Problem 8 Solution: Optimal Search Trees

By Nalin Ranjan and Ruijie Fang

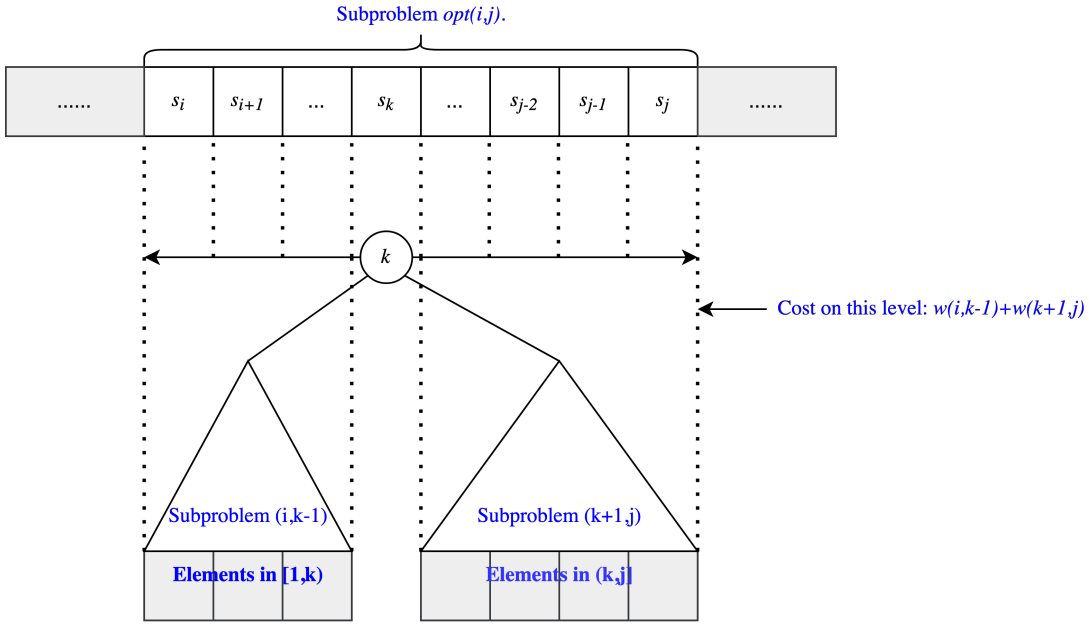
There were two parts and an extra credit problem to this problem. The first part asked you to come up with an $O(n^2)$-time dynamic programming scheme to solve the optimal binary search trees problem, and the second part asked you to solve the analogously defined "optimal ternary search trees" problem by generalizing your dynamic programming scheme from Part 1.

1 Solution to First Part

Let $\texttt{opt}(i, j)$ denote the optimal OBST cost for keys in range $[i, j]$. Then we can notice that

$$\texttt{opt}(i,j) = \min_{i \le k \le j} \Big[\texttt{opt}(i, k-1) + w(i, k-1) + \texttt{opt}(k+1, j) + w(k+1, j) \Big]$$

where the weight function $w(i, j) = \sum_{k=i}^{j} F_i$ is the total access frequency of the keys from i to j. The reason for this relation's correctness can be summed up in this diagram:



We're essentially just looping through all possible roots of the OBST on $[i, j]$ and asking what the best possible cost would be. If we select k as the root, then the best we can do is to make the left subtree the OBST on keys $[i, k-1]$, and the right subtree the OBST on keys $[k+1, j]$. In the process, we've "demoted" each of the nodes in the child subtrees, and thus we have increased our cost by the total cost of nodes in either of the two subtrees (which is where the $w(i, j)$ terms come from).

To turn this relation into an actual program that calculates the answer, we would first note that we know the base cases whenever $i \geq j$. Now, we have all the pre-requisite values for calculating $\texttt{opt}(i, j)$ if $j - i \leq 2$. After that, we will have all the pre-requisite values for whenever $j - i \leq 4$, and so on.

Unfortunately, even if we use prefix sums of $w(\cdot)$ to be able to retrieve $w(i, j)$ efficiently, the above algorithm still still takes $\Theta(n^3)$-time (memoized) in the worst case. However, observe the weight function satisfies the quadrangle inequality

$$w(i, j) + w(i', j') \leq w(i, j') + w(i', j), \text{ whenever } i \leq i' \leq j \leq j'$$

How does this help? There is a sequence of two lemmas in the original Yao paper and on pages 76-78 of this reference (which we would have encouraged you to Google during the competition) that ultimately prove that the calculation of all the elements on any 'diagonal' of the DP table (i.e. where $j - i$ is held constant) can be calculated in $O(n)$ steps, given that every diagonal below it has been calculated. Central to this is storing of an auxiliary *index table* K_B, where

$$K_B(i, j) = \max\{t \mid w(i, t) + \texttt{opt}(i, t-1) + \texttt{opt}(t, j) = \texttt{opt}(i, j)\}$$

That is, $K_B(i, j)$ is the *maximum* index that we can choose as the root of the OBST on keys $[i, j]$. It turns out that

Lemma 1: Suppose $w(i, j)$ satisfies the quadrangle inequality and is monotone (i.e. $w(i, j) \leq w(i', j')$ whenever $[i, j] \subseteq [i', j']$). Then if `opt` satisfies the recurrence at the beginning of this section, it follows that `opt` also satisfies the quadrangle inequality.

This in turn is used to show that

Lemma 2: If `opt` satisfies the quadrangle inequality, then $K_B(i, j-1) \leq K_B(i, j) \leq K_B(i+1, j)$.

And while lemma 2 may on its face seem unhelpful, it actually tells us two important things: First, the

elements on the diagonal of the DP table are increasing as we increase the row index. That is, if we fix $d = j - i = j' - i' \geq 0$, then

$$K_B(i, j) \leq K_B(i', j'), \text{ if } i \leq i'$$

Furthermore, our search for the optimal root for keys $[i, j]$ need only go from $K_B(i, j-1)$ to $K_B(i+1, j)$. Note that these two values are on the $(d-1)$th diagonal of the DP matrix, so this suggests that like before, we should first calculate the values along the zeroth (main) diagonal, then the values along the first, and so on. All in all, the cost of calculating all elements on the dth diagonal (assuming keys are 1-indexed) is

$$\sum_{i=1}^{n-d} \left[K_B(i+1, i+d) - K_B(i, i+d-1)\right] = K_B(n-d+1, n) - K_B(1, d) \leq n$$

And since there are only n total diagonals, the algorithm runs in $O(n^2)$ time.

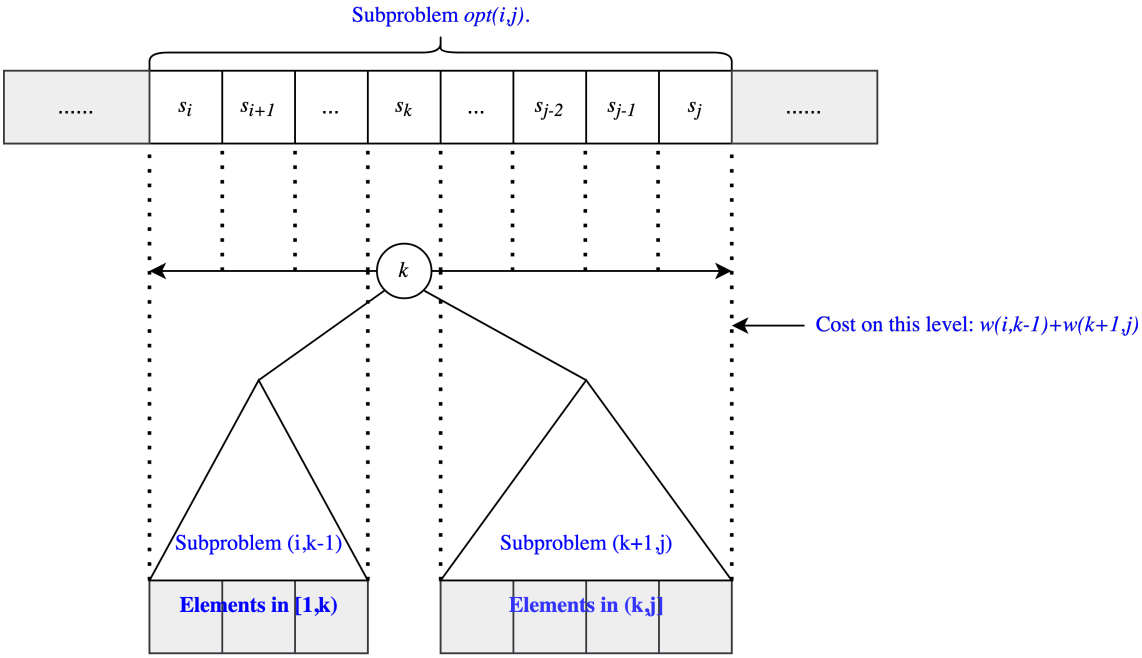
Proofs of the lemmas can be found in the original Yao paper. We provide two implementations of the optimal BST solution in C++: `OBST_Slow.cpp` which implements the $O(n^3)$-time dynamic programming scheme, and `OBST_Fast.cpp` which implements the quadrangle inequality-based speedup.

2 Solution to Second Part

Inspired by the naive dynamic programming recurrence from the first part, one might first think of a $O(n^4)$-time dynamic program $\texttt{opt}_3(i, j)$, which likewise stores the optimal cost for keys in range $i \leq k \leq j$, as follows:

$$\texttt{opt}(i, j) = \min_{i < k_1 \leq k_2 \leq j} \Big[\texttt{opt}_3(i, k_1 - 1) + \texttt{opt}_3(k_1 + 1, k_2 - 1) + \texttt{opt}_3(k_2 + 1, j) + w(i, k_1 - 1) + w(k_1 + 1, k_2 - 1) + w(k_2 + 1, j)\Big]$$

See the diagram on the next page for a visualization of this recurrence. Again, we're essentially just trying to brute-force guess the two points that are going to divide the range $[i, j]$. In an actual ternary search tree, these two points (the 'pivots') would be stored in the root so a search knows which subtree to recur on. Observe that the above recurrence works in $O(n^4)$ because of the double nested loop computation induced by two indices k_1, k_2 at each recursion level, so again it fails to meet the running time requirements.



However, there is room for improvement, and the answer lies in something you might remember from COS 226: 2-3 trees. Instead of trying to find two pivots k_1 and k_2, let's just look for one pivot, which we'll call k^*. We'll then force k^* to only have two children, like in the OBST problem. The crucial point: in this BST, one of the children of k^* will be a sibling of k^* in the actual TST. The other child who is not a sibling, in addition to the children of the sibling, will be the actual children of the node in the TST.

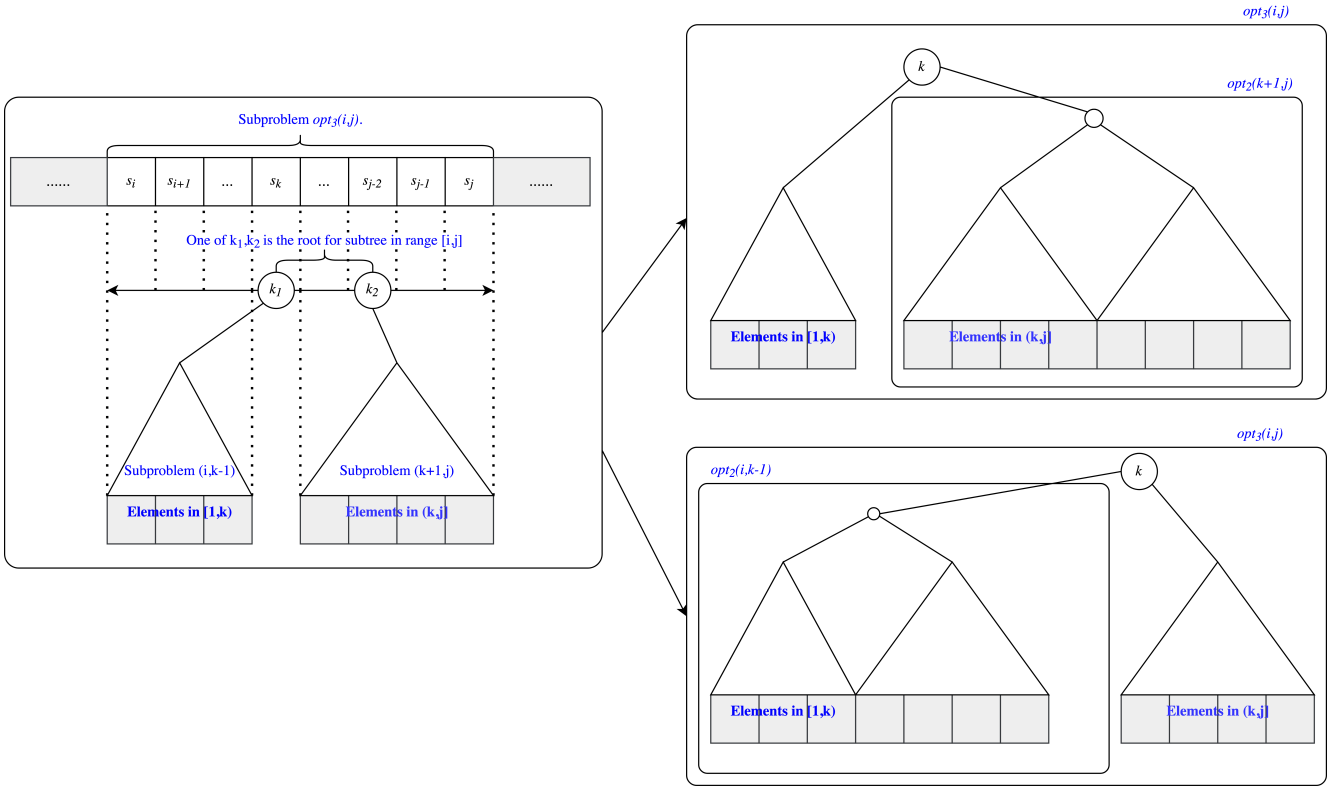
The only thing, then, is we must mandate is that the sibling of k^* has only two children. This leads us to define two quantities: $\texttt{opt}_3(i,j)$, which is the minimum weight of a TST we can construct on keys in $[i,j]$, and $\texttt{opt}_2(i,j)$, which is the minimum weight of a TST we can construct on keys in $[i,j]$ *if the root is forced to be a binary node (i.e. it only has two children).* Then we may write

$$\texttt{opt}_2(i,j) = \min_{i \le k \le j} \left[\texttt{opt}_3(i,k-1) + w(i,k-1) + \texttt{opt}_3(k+1,j) + w(k+1,j) \right]$$

and because of the previous discussion,

$$\texttt{opt}_3(i,j) = \min_{i \le k \le j} \Big[\min \Big(\texttt{opt}_3(i,k-1) + w(i,k-1) + \texttt{opt}_2(k+1,j), \\ \texttt{opt}_2(i,k-1) + \texttt{opt}_3(k+1,j) + w(k+1,j) \Big) \Big]$$

The two cases in the expression for $\texttt{opt}_3$ are essentially determining whether the root of the left subtree or the root of the right subtree should be the sibling of k in the OTST. See the diagram below for a visualization:



And again, we see that we should calculate both $\texttt{opt}_2(i,j)$ and $\texttt{opt}_3(i,j)$ in order of $j-i$ (i.e. we start from the "diagonal" and move up), because both $\texttt{opt}_2(i,j)$ and $\texttt{opt}_3(i,j)$ only depends on $\texttt{opt}_3(i',j')$ and $\texttt{opt}_2(i',j')$, where $j'-i' = j-i-1$.

All in all, we have decreased the runtime to $O(n^3)$ at the expense of storing another $n \times n$ matrix for $\texttt{opt}_2$. We provide two implementations of the optimal TST solution in C++: `OTST_Slow.cpp`, which implements the naive $O(n^4)$-time dynamic programming recurrence, and `OTST_Fast.cpp`, which implements the $O(n^3)$-time mutual recursive solution.

Plaudits:

- Congratulations to Antonio Molina-Lovett (grad), Yuping Luo (grad)/Xiaoqi Chen (grad)/Dingli Yu (grad), and Kaifeng Lyu (grad)/Shunyu Yao (grad)/Qinshi Wang (grad) for being the only teams to solve both parts of the problem!
- Congratulations to Yuping Luo (grad)/Xiaoqi Chen (grad)/Dingli Yu (grad) for being the first to submit a correct solution to the BST part of the problem, at a speedy 33 minutes.
- Congratulations to Antonio Molina-Lovett (grad) for being the first to submit a correct solution to the TST part of the problem, at one hour and thirteen minutes.