If we have no knowledge of the frequency with which of data keys are accessed, then we make the assumption of uniform access (frequency = 1/n for each of the n items), and derive our average case retrieval cost accordingly. In some applications, the data set is relatively static, and further, the access frequencies are not uniform - and can be well estimated. Then it is appropriate to build a binary search tree which gives a minimum average-case retrieval cost. Consider three items, with the keys 'a' (0.45), 'b' (0.40), and 'c' (0.15). There are several binary search trees that can be constructed with these keys, but if we want to minimize average search cost, it seems plain that the keys with high access frequencies should be close to the root of the tree. Suppose we try a greedy approach: put 'a' at the root, and 'b' in it's right subtree (of course), and 'c' in the right subtree of 'b'. The average cost of treesearch (in comparisons) is:

$$\text{average cost} = (0.45)(1) + (0.40)(3) + (0.15)(5) = 2.4$$

since 'a' is retrieved in one comparison, 'b' in three, and 'c' in five. But consider the BST with 'b' at the root, 'a' in the left subtree, and 'c' in the right subtree. Now the average search cost is:

$$\text{average cost} = (0.40)(1) + (0.45)(3) + (0.15)(3) = 2.2$$

so the "greedy" approach is not the best. We'll find the optimal binary search tree by using a method called dynamic programming.

Suppose we have n keys, $k_1 < k_2 < \ldots < k_n$ and their associated access frequencies $f_1, f_2, \ldots, f_n$ where $\sum f_i = 1$. Which key should be the root? And which the root of the left-subtree/right-subtree? And their respective subtrees? We could enumerate all BST's for a given set of keys and take the one(s) with the smallest average search cost, but - as you might imagine - the number of BST's is exponential in the number of keys. The dynamic programming method exploits the (fairly obvious) idea that the optimal tree has optimal subtrees.

Let $C_{ij}$ be the average cost of searching an optimal BST containing the keys $k_i$ through $k_j$, $i \leq j$, and take $w_{ij} = f_i + \ldots + f_j$. Then, as we have come to expect with binary trees, we can express $C_{ij}$ recursively as:
\[ C_{ij} = \min \{ f_k + 2w_{i,k-1} + C_{i,k-1} + 2w_{k+1,j} + C_{k+1,j} \}, \quad i \leq k \leq j \]
\[ = 2w_{ij} + \min \{ C_{i,j} + C_{k+1,j} - f_k \}, \quad i \leq k \leq j \]
\[ = f_i, \quad i = j \]
\[ = 0, \quad i > j \]

After all, if \( k_k \) is the root key, then it can be retrieved with just one comparison (and that happens \( f_k \) of the time). Further, we have to search in the left-subtree \( w_{i,k-1} \) of the time at a cost of two comparisons at the root plus the average cost of searching the optimal BST containing the keys \( k_i \) through \( k_{k-1} \), and a similar component must be added for the right subtree. And the smallest average search cost results from finding the minimum of the expression above over all possible choices for the root key. This formulation - of an optimal value (an extremal value) over a structure expressed in terms of optimal values over substructures - is characteristic of the dynamic programming method.

Of course we are only interested in \( C_{1,n} \)- but all the other \( C_{ij} \)'s must be computed in order to produce the final result. And, as the optimal costs are computed, we also discover the optimal subtrees that finally produce the complete optimal BST that contains all the keys. The recursive formulation above is elegant, and summarizes the situation very well - but is next to useless in terms of computation. (Recall the expensive computational cost associated with the recursive calculation of the Fibonacci numbers.) The dynamic programming method eliminates some of the recursive overhead by maintaining a table of intermediate results (we could have done this with Fibonacci, of course) so that repetitive calculations are avoided. In fact, there is a very elegant way to view how the computation takes place, and it motivates a non-recursive program for the calculation.

Consider the upper-triangle of an \( n \times n \) array, where the elements of the array are the \( C_{ij} \)'s. The diagonal elements - \( C_{ii} \) - are just the \( f_i \) as noted above, so we know these values. Next, we must calculate the \( C_{i,i+1} \)'s: the terms on the diagonal just above the major diagonal. If you examine the recursive formula above, these can all be calculated (\( i = 1, 2, \ldots, n-1 \)) from the values on the major diagonal. Then the \( C_{i,i+2} \) terms (\( i = 1, 2, \ldots, n-2 \)) must be computed from the previous diagonal. Thus, we compute the values of the terms on the \( k \)th diagonal (\( k = 1, 2, \ldots, n-1 \)) and ultimately learn the only value we really want - \( C_{1,n} \).

Notice that on the \( k \)th diagonal above the major diagonal there are \( n-k \) values that must be computed. To obtain the overall cost of the algorithm, we need to calculate for each \( k \) the cost of computing the values of all the elements on that diagonal, and then sum the costs over all the diagonals. Suppose we are trying to calculate \( C_{i,i+k} \), for some \( k = 1, 2, \ldots, n-1 \). That is we are, in effect, trying to locate the optimal root over the range of keys from \( k_i \) to \( k_{i+k} \) so there are exactly \( k+1 \) possibilities for the root, and each must be considered (in the presence of optimal left and right subtrees), and a minimum chosen. The total effort to calculate all the values on the \( k \)th diagonal is thus proportional to \( (k+1)(n-k) \), and since \( k \) ranges from 1 to \( n-1 \), we have that the computational effort required to “fill in” the cells of the array is proportional to \( \sum_{k=1}^{n-1} (k+1)(n-k) = n^3 \), or \( O(n^3) \). (The sum of the squares of successive integers from 1 to \( n \) has a simple relationship to the sum of the integers over the same range.)
This is a high complexity order, and only makes sense if the data elements of the tree are essentially static, so that we only have to invest this effort once to build the optimal search tree, and then subsequently enjoy the benefits of least-average-cost-search. In fact, the complexity can be reduced to $O(n^2)$ - which is sometimes possible in problems solved with a dynamic programming formulation: see the exercises relating to the optimal BST in the text.

Can you sketch the non-recursive code for computing the elements of the upper triangular array? And once the optimal cost is known, how do we know the optimal tree? Think of a clever way to use the lower triangle of the array so that the optimal tree can be reconstructed once the computation of the costs is completed.