This lecture note describes an application of the dynamic programming paradigm on computing the optimal static binary search tree of \( n \) keys to minimize the expected search time for a given search probability distribution.

Suppose we are given a collection of \( n \) keys \( K_1 < K_2 < \cdots < K_n \) which are to be stored in a binary search tree. After the tree has been constructed, only search operations will be performed — i.e. there will be no insertions or deletions. We are also given a probability density function \( P \) where \( P(i) \) is the probability of searching for key \( K_i \). There are many different binary search trees where the \( n \) given keys can be stored. For a particular tree \( T \) with these keys, the average number of comparisons to find a key, for the given probability density is

\[
\sum_{i=1}^{n} P(i) \cdot (\text{depth}_T(K_i) + 1),
\]

where \( \text{depth}_T(K_i) \) denotes the depth of the node where \( K_i \) is stored in \( T \). The problem we would like to solve is to find, among all the possible binary search trees that contain the \( n \) keys, one which minimizes this quantity. Such a tree is called an optimal (static) binary search tree. Note that there may be several optimal binary trees for the given density function. This is why we speak of an optimal, rather than the optimum binary search tree.

A simple way to accomplish this is to try out all possible binary trees with \( n \) nodes, computing the average number of comparisons to find a key in each tree considered, and selecting a tree with the minimum average. Unfortunately, this simple strategy is ridiculously inefficient because there are too many trees to try out. In particular, there are \( \binom{2n}{n}/(n+1) \) different binary trees with \( n \) nodes (if interested in the derivation of this formula, see Knuth, vol. I, pp. 388-389). Thus, if there are 20 keys, we have to try out 131,282,408,400 different trees. Computing the average number of comparisons in each at the rather astonishing speed of \( 1\mu \text{sec} \) per tree, will still take 2188 hours or approximately 91 days and nights of computing to find an optimal binary search tree (for just 20 keys)! Fortunately, there is a much more efficient, if less straightforward, way to find an optimal binary search tree. Let \( T \) be a binary search tree that contains \( K_i, K_{i+1}, \ldots, K_j \) for some \( 1 \leq i \leq j \leq n \). We shall see shortly why it is useful to consider trees that contain subsets of successive keys. We define the cost of \( T \) as,

\[
c(T) = \sum_{l=i}^{j} P(l) \cdot (\text{depth}_T(K_l) + 1)
\]

Hence, if \( T \) contains all \( n \) keys (i.e. \( i = 1 \) and \( j = n \)), the cost of \( T \) is precisely the expected number of comparisons to find a key for the given density function.† Thus, we can rephrase our problem as follows: Given a density function for the \( n \) keys, find a minimum cost tree with \( n \) nodes.

Before giving the algorithm to find an optimal binary search tree, we prove two key facts.

\begin{lemma}
Let \( T \) be a binary search tree containing keys \( K_i, K_{i+1}, \ldots, K_j \), \( T_L \) and \( T_R \) be the left and right subtrees of \( T \) respectively. Then
\end{lemma}

† This is not so if \( T \) is missing some of the keys, because in that case the probabilities of the keys that are in \( T \) do not sum to 1 — that is, \( P \) is not a proper density function relative to the set of keys in the tree.
\[ c(T) = c(T_L) + c(T_R) + \sum_{i=1}^{j} P(l). \]

**Proof:** This is an easy consequence of the definition of cost of a tree. You should prove it on your own. □

**Lemma 2:** Let \( T \) be a binary search tree that has minimum cost among all trees containing keys \( K_i, K_{i+1}, \ldots, K_j \) and let \( K_m \) be the key at the root of \( T \) (so \( i \leq m \leq j \)). Then \( T_L \), the left subtree of \( T \), is a binary search tree that has minimum cost among all trees containing keys \( K_i, \ldots, K_{m-1} \); and \( T_R \), the right subtree of \( T \), is a binary search tree that has minimum cost among all trees containing keys \( K_{m+1}, \ldots, K_j \).

**Proof:** We prove the contrapositive. That is, if either the left or right subtree of \( T \) fails to satisfy the minimality property asserted in the lemma we show that \( T \) does not really have the minimum possible cost among all trees that contain \( K_i, \ldots, K_j \).

Let \( T'_L \) and \( T'_R \) be minimum cost binary search trees that contain \( K_i, \ldots, K_{m-1} \) and \( K_{m+1}, \ldots, K_j \) respectively. Thus, \( c(T'_L) \leq c(T_L) \) and \( c(T'_R) \leq c(T_R) \) (*).

Further, let \( T' \) be the tree with key \( K_m \) in the root, and left and right subtrees \( T'_L \) and \( T'_R \) respectively. Evidently, \( T' \) is a binary search tree that contains keys \( K_i, \ldots, K_j \). If \( T_L \) or \( T_R \) do not have the minimality property asserted by the lemma, it must be that either \( c(T_L) > c(T'_L) \), or \( c(T_R) > c(T'_R) \). This, together with (*) implies that \( c(T_L) + c(T_R) > c(T'_L) + c(T'_R) \). From this and Lemma 1 we have,

\[ c(T) = c(T_L) + c(T_R) + \sum_{i=1}^{j} P(l) > c(T'_L) + c(T'_R) + \sum_{i=1}^{j} P(l) = c(T'). \]

Thus, \( c(T) > c(T') \) and \( T \) is not a minimum cost binary search tree among all trees that contain keys \( K_i, \ldots, K_j \). □

**Computing an Optimal Binary Search Tree**

Lemma 2 is the basis of an efficient algorithm to find an optimal binary search tree. Let \( T_{ij} \) denote an optimal binary search tree containing keys \( K_i, K_{i+1}, \ldots, K_j \). Then \( T_{1n} \) is precisely the optimal binary search tree that we want to construct. Lemma 2 says that \( T_{ij} \) must be of the following form:

\[ \begin{array}{c}
K_m \\
T_{i,m-1} & & & T_{m+1,j} \\
\end{array} \]

That is, its root has key \( K_m \) for some \( m, i \leq m \leq j \), and its subtrees are \( T_{i,m-1} \) and \( T_{m+1,j} \), i.e. minimum cost subtrees containing keys \( K_i, \ldots, K_{m-1} \) and \( K_{m+1}, \ldots, K_j \) respectively. But \( T_{i,m-1} \) and \( T_{m+1,j} \) are "smaller" trees than \( T_{ij} \). This suggests proceeding inductively, starting with small minimum cost trees (each containing just one key) and progressively building larger and larger minimum cost trees, until we have a minimum cost tree with \( n \) nodes which is what we are looking for.
More specifically, we start the induction with minimum cost trees each of which contains exactly one key, and proceed by constructing minimum cost trees with 2, 3, \ldots, \ n successively.

Note that there are exactly \( n - d + 1 \) groups of \( d \) successively keys for each \( d = 1, \ \ldots, \ n \).

Thus, instead of considering all possible trees with \( n \) nodes we consider only \( n \) (minimum cost) trees with 1 node each, \( n - 1 \) (minimum cost) trees with 2 nodes each, \( \cdots \), 1 minimum cost tree with \( n \) nodes, i.e. a total of \( n(n + 1)/2 \) trees — much fewer than \( \binom{n+1}{2} \).

So now the question is how to compute these trees \( T_{ij} \) inductively. The basis of the induction, i.e. when \( j - i = 0 \) is trivial. In this case we have \( j = i \) and the minimum cost binary search tree that stores \( K_i \) (in fact the only such tree!) is a single node containing \( K_i \); its cost is \( c(T_{ii}) = P(i) \).

For the inductive step, take \( j - i > 0 \) and assume that we have already computed all the \( T_{uv} \)'s and their costs, for \( v - u < j - i \). Let \( T_{imj} \) be the tree with \( K_m \) in the root, and left and right subtrees \( T_{i,m-1} \) and \( T_{m+1,j} \) respectively. As we saw before, Lemma 2 implies that \( T_{ij} \) is the minimum cost tree among the \( T_{imj} \)'s. Thus we can find \( T_{ij} \) simply by trying out all the \( T_{imj} \)'s for \( m = i, i + 1, \cdots, j \). In fact Lemma 1 tells us how to compute \( c(T_{imj}) \) efficiently, so that "trying out" each possible \( m \) will not take too long: Since \( (m - 1) - i \) and \( j - (m + 1) \) are both \( < j - i \), we have already (inductively) computed \( T_{i,m-1} \) and \( T_{m+1,j} \) as well as their costs, \( c(T_{i,m-1}) \) and \( c(T_{m+1,j}) \). Lemma 1 then tells us how to get \( c(T_{imj}) \) in terms of these. Note that when \( m = i \) the left subtree of \( T_{imj} \) is \( T_{i,j-1} \) and when \( m = j \) the right subtree of \( T_{imj} \) is \( T_{i+1,j} \). We define \( T_{ij} \) to be empty if \( i > j \), and the cost of an empty tree to be 0.

Figure 1 shows this algorithm in pseudo-code. The algorithm takes as input an array \( \text{Prob}[1..n] \), which specifies the probability density (i.e. \( \text{Prob}[i] = P(i) \)). It computes two two-dimensional arrays, \( \text{Root} \) and \( \text{Cost} \), where \( \text{Root}[i, j] \) is the root of \( T_{ij} \), and \( \text{Cost}[i, j] = c(T_{ij}) \), for \( 1 \leq i \leq j \leq n \).

To help compute \( \text{Root} \) and \( \text{Cost} \) the algorithm maintains a third array, \( \text{SumOfProb} \), where \( \text{SumOfProb}[i] = \sum_{l=1}^{i} P(l) \) for \( 1 \leq i \leq n \), and \( \text{SumOfProb}[0] = 0 \). Note that \( \sum_{i=1}^{j} P(l) = \text{SumOfProb}[j] - \text{SumOfProb}[i-1] \).

The algorithm of Figure 1 does not explicitly construct an optimal binary search tree but such a tree is implicit in the information in array \( \text{Root} \). As an exercise you should write an algorithm which, given \( \text{Root} \) and an array \( \text{Key}[1..n] \), where \( \text{Key}[i] = K_i \), constructs an optimal binary search tree.

It is not hard to show that this algorithm has worst case time complexity \( \Theta(n^3) \). A slight modification of this algorithm leads to a \( \Theta(n^2) \) complexity (if interested, see D.E. Knuth, "Optimum binary search trees," \textit{Acta Informatica}, vol. 1 (1971), pp. 14-25.)

\textbf{Unsuccessful Searches}

In this discussion we have only considered successful searches. However, if we take into account unsuccessful searches, maybe the constructed tree is no longer optimal. Fortunately, this problem can be taken care of in a straightforward manner. To find an optimal binary search tree in the case where both successful and unsuccessful searches are taken into account, we must

\* For technical reasons that will become apparent when you look at the algorithm carefully we need to set 
\[ \text{Cost}[i, i-1] = 0 \text{ for } 1 \leq i \leq n + 1. \] 
Recall that \( T_{i,j-1} \) is empty and thus has cost 0.
algorithm *OptimalBST* (*Prob*[1..n]*); begin
(* initialization *)
for *i* ← 1 to *n* + 1 do *Cost*[i, *i* − 1] ← 0;
*SumOfProb*[0] ← 0;
for *i* ← 1 to *n* do begin
    *SumOfProb*[i] ← *Prob*[i] + *SumOfProb*[i − 1];
    *Root*[i, i] ← *i*;
    *Cost*[i, i] ← *Prob*[i]
end;

for *d* ← 1 to *n* − 1 do (*compute info about trees with *d* + 1 consecutive keys *)
    for *i* ← 1 to *n* − *d* do begin (*compute *Root*[i, *j*] and *Cost*[i, *j]*)
        *j* ← *i* + *d*;
        *MinCost* ← +∞;
        for *m* ← *i* to *j* do begin (*find *m* between *i* and *j* so that *c*(T<sub>imj</sub>) is minimum *)
            *c* ← *Cost*[i, *m* − 1] + *Cost*[m + 1, *j*] + *SumOfProb*[*j*] − *SumOfProb*[i − 1];
            if *c* < *MinCost* then begin
                *MinCost* ← *c*;
                *r* ← *m*
            end
        end;  
        *Root*[i, *j*] ← *r*;
        *Cost*[i, *j*] ← *MinCost*
end
end

*Figure 1*: Algorithm for optimal binary search tree.

know the probability density for both successful and unsuccessful searches. So, in addition to *P*(i) we are also given *Q*(i) for 0 ≤ *i* ≤ *n*, where

- *Q*(0) = prob. of searching for keys < *K*<sub>1</sub>;  
- *Q*(i) = prob. of searching for keys *x*, *K*<sub>i</sub> < *x* < *K*<sub>i+1</sub>, for 1 ≤ *i* < *n*;  
- *Q*(n) = prob. of searching for keys > *K*<sub>n</sub>.  

In each binary search tree containing keys *K*<sub>1</sub>, *K*<sub>2</sub>, ..., *K*<sub>*n*</sub> we add *n* + 1 external nodes *E*<sub>0</sub>, *E*<sub>1</sub>, ..., *E*<sub>*n*</sub>. This is illustrated below; the external nodes are drawn in boxes, as usual.

![Diagram](https://via.placeholder.com/150)

The average number of comparisons for a successful or unsuccessful search in such a tree *T* is
\[
\sum_{i=1}^{n} P(i) \cdot (\text{depth}_T(K_i) + 1) + \sum_{i=0}^{n} Q(i) \cdot \text{depth}_T(E_i).
\]

The left term is the contribution to the average number of comparisons by the successful searches and the right term is the contribution to the average number of comparisons by the unsuccessful searches. Now we want to find a tree that minimizes this quantity.

We can proceed exactly as before, except that the definition of the cost of a tree \(T\) with consecutive keys \(K_i, K_{i+1}, \ldots, K_j\) is slightly modified to account for the unsuccessful searches. Namely, it becomes,

\[
c'(T) = \sum_{l=i}^{j} P(l) \cdot (\text{depth}_T(K_l) + 1) + \sum_{l=i-1}^{j} Q(l) \cdot \text{depth}_T(E_l).
\]

With this cost function Lemma 1 is slightly different:

**Lemma 1’**: Let \(T\) be a binary search tree containing keys \(K_i, K_{i+1}, \ldots, K_j, T_L\) and \(T_R\) be the left and right subtrees of \(T\) respectively. Then

\[
c'(T) = c'(T_L) + c'(T_R) + Q(i-1) + \sum_{l=i}^{j} (P(l) + Q(l)).
\]

Everything else works out exactly as before. In particular, Lemma 2 is still valid (check this!). As an exercise show how to modify the algorithm in Figure 1 to account for these changes.

**Example**: Suppose we want to find an optimal binary search tree for the dictionary \{begin, end, goto, repeat, until\} for the following probabilities of searching for these keys \((P(i)’s)\) and keys alphabetically “in between” (the \(Q(i)’s)\):

<table>
<thead>
<tr>
<th></th>
<th>(K_1 = \text{begin})</th>
<th>(K_2 = \text{end})</th>
<th>(K_3 = \text{goto})</th>
<th>(K_4 = \text{repeat})</th>
<th>(K_5 = \text{until})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P(1))</td>
<td>.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P(2))</td>
<td></td>
<td>.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P(3))</td>
<td></td>
<td></td>
<td>.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P(4))</td>
<td></td>
<td></td>
<td></td>
<td>.05</td>
<td></td>
</tr>
<tr>
<td>(P(5))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.05</td>
</tr>
<tr>
<td>(Q(0))</td>
<td>.05</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Q(1))</td>
<td></td>
<td>.05</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Q(2))</td>
<td></td>
<td></td>
<td>.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Q(3))</td>
<td></td>
<td></td>
<td></td>
<td>.2</td>
<td></td>
</tr>
<tr>
<td>(Q(4))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.1</td>
</tr>
<tr>
<td>(Q(5))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.05</td>
</tr>
</tbody>
</table>

The trees \(T_{ij}\) as computed by the algorithm on this example are shown in the table below with their costs. The computation proceeds row by row (i.e. to compute the trees and costs in row \(d\), all the trees up to row \(d-1\) must have been previously computed). The optimal binary search tree \(T_{1,5}\) is shown on the last row.

You are strongly encouraged to trace this example carefully. Even if you think you understand the algorithm from the previous abstract discussion, you may be surprised at how much better you’ll understand it after working out an example.
$d = 1:$

\[
\begin{array}{cccccc}
K_2 & K_2 & K_3 & K_3 & K_4 & K_5 \\
E_2 & K_2 & E_3 & E_4 & E_3 & E_5 \\
E_0 & E_1 & E_2 & E_3 & E_4 & E_5 \\
\end{array}
\]

\[c(T_{12}) = .7 \quad c(T_{23}) = .95 \quad c(T_{34}) = .95 \quad c(T_{45}) = .65\]
\[ d = 2 : \]

\[
\begin{array}{c}
K_1 \\
K_0 & E_1 & E_2 & E_3 & E_4 \\
K_2 & K_3
\end{array}
\]

\[ c(T_{13}) = 1.4 \quad c(T_{24}) = 1.45 \quad c(T_{35}) = 1.35 \]

\[ d = 3 : \]

\[
\begin{array}{c}
K_1 \\
K_3 & K_2 & K_4 \\
K_0 & E_0 & E_1 & E_2 & E_3 & E_4 \\
K_0 & E_1 & E_2 & E_3 & E_4 & E_5
\end{array}
\]

\[ c(T_{14}) = 1.95 \quad c(T_{25}) = 1.85 \]

\[ d = 4 : \]

\[
\begin{array}{c}
K_1 \\
K_3 & K_2 & K_4 \\
K_0 & E_0 & E_1 & E_2 & E_3 & E_4 & E_5
\end{array}
\]

\[ c(T_{15}) = 2.35 \]