Lecture 5: Dynamic programming II

Longest common subsequence (LCS)

In biological applications we may want to compare two DNA strings, X and Y, to see how similar they are, as a measure of how closely related the organisms are. This could be indicated by a common substring, the longer it is the more similar X and Y are. The characters in the substring are not necessarily consecutive in X or Y.

Step 1: Characterizing an LCS

Given sequences $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$, find their LCS.

$$\begin{array}{lccccccc} X: & A & B & C & B & D & A & B \\ Y: & B & D & C & A & B & A & \end{array} \quad \Rightarrow \text{ LCS} = B\,C\,B\,A$$

A brute-force algorithm checks for each subsequence of the shorter sequence, Y, if it is a subsequence of X. This takes $\Theta(m2^n)$ time, since there are 2^n subsequences of Y, and each check takes $\Theta(m)$ time.

Optimal substructure of an LCS

The optimal solution to the problem contains optimal solutions to subproblems.

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

Clearly, if $x_m = y_n$ this implies $z_k = x_m = y_n$ and that Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}. Since if $z_k \neq x_m$ then by appending x_m to Z we would get a common subsequence of X and Y of length $k+1$. Contradiction!

If $x_m \neq y_n$ then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y, and $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Step 2: A recursive solution

Keep track of lengths, not the actual sequences.

Let $c[i,j]$ be the length of an LCS of $X_i = \langle x_1, \ldots, x_i \rangle$ and $Y_j = \langle y_1, \ldots, y_j \rangle$.

The optimal substructure of the problem gives

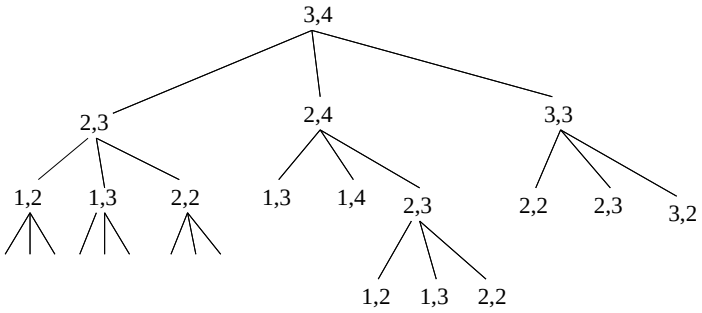
$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

The desired answer is $c[m,n]$.

Step 3: Computing the length of an LCS

The definition of $c[i, j]$ easily gives an exponential-time recursive algorithm, but more efficient is to use dynamic programming.

Recursion tree for $c[3, 4]$:



Overlapping subproblems Circle repeated subproblems in the recursion tree above. There are few different subproblems (contrast this with divide-and-conquer where subproblems are independent). For LCS there are $m \cdot n$ subproblems (since there are m prefixes of X, and n prefixes of Y).

Start with c[1,1] and store computed subsolutions in a table that we look into. We demonstrate the algorithm by filling the LCS table in Figure 15.8, page 395, with X and Y as before giving a $(7 + 1) \times (6 + 1)$ table:

- Initialize first row and first column to 0.
- Fill squares row by row from left to right, based on the values of three neighbors. If $x_i = y_j$ take the diagonal value and add 1; and draw a diagonal arrow. Otherwise take maximum of value above and value to the left.

Time is proportional to the size of the table, $\Theta(mn)$, each subproblem is computed once and looked up twice. Length of an LCS is the number in the lower right-hand corner of the table.

Step 4: Constructing an LCS

Can construct an LCS (in reverse order) by tracing the table backwards from entry $[m, n]$: diagonally if there is an arrow, otherwise to a neighbor with the same value. This takes $\Theta(m + n)$ time.

Space above is $\Theta(mn)$. In tutorial 2 we will show how this can be reduced to $\Theta(\min(m, n))$ if we just compute lengths of LCS.

However, if we want to give the characters in an LCS, and not just its length, more than linear space is required. There is an algorithm which uses $O(mn/\log n)$ space, where $n \leq m$ and the characters are from a bounded alphabet.

Memoization takes care of overlapping subproblems by filling the table top-down instead. When a subproblem is first encountered during the execution of the recursive algorithm, its solution is computed and stored in the table.

A memoized algorithm for LCS would start with $c[m, n]$ instead of $c[1, 1]$. Each subproblem is computed once and looked up twice.

In practice, a bottom-up dynamic programming algorithm usually outperforms a top-down memoized algorithm by a constant factor, because there is no overhead for recursion and less overhead for maintaining the table (page 389).

Optimal binary search trees (OBST)

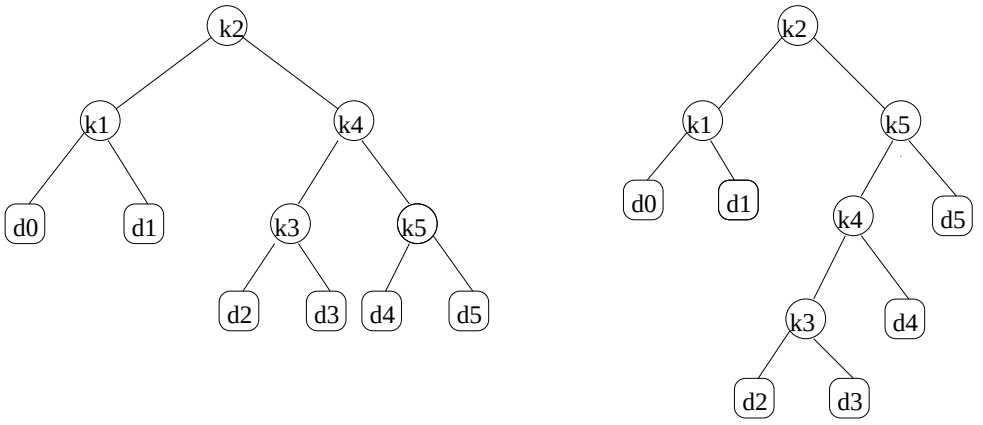
Consider the problem of a compiler identifying keywords (like *begin*, *end*, etc.) in a program. We could build a red-black tree with n keywords, which each can be found in $O(\log n)$ time. But if we want to minimize the time to find all keys in the text we better place frequent keys close to the root of the tree.

In general: Given is a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct, sorted keys, where k_i has probability p_i to be queried. An unsuccessful search will end up in one of the $n+1$ intervals $d_0, d_1, \ldots, d_n$ between keys (where d_0 represents values $< k_1$, d_n values $> k_n$). These d_i have search probabilities q_i.

(Compare with Huffman trees which have no search order between stored characters.)

Consider two binary search trees with the following probabilities:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10



The first tree has expected search cost 2.80, the second has expected search cost 2.75 (which is optimal). It holds that

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

and since the root has depth 0, the number of tests to find k_i is its depth $+1$.

$$E[\text{search cost}] = \sum_{i=1}^{n} (\text{depth}(k_i)+1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}(d_i)+1) \cdot q_i = 1 + \sum_{i=1}^{n} \text{depth}(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}(d_i) \cdot q_i$$

This gives the following expected search cost for the first tree in the figure above:

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

While preserving the *search-tree property*, such that all keys in the left subtree $<$ root, and all keys in the right subtree $>$ root, we want to minimize the total search cost by testing all keys as possible root. In addition, a subtree with $k_i, \ldots, k_j$ also contains $d_{i-1}, \ldots, d_j$ as leaves.

Step 1: Structure of an OBST

Optimal substructure: If a tree is optimal then so are its subtrees (since otherwise we would get a better tree by substituting a subtree for an optimal one). This gives the algorithm idea to systematically build bigger optimal subtrees.

Step 2: A recursive solution

Let $\boxed{e[i,j]}$ denote the minimum expected search cost for the subtree with $k_i, \ldots, k_j$ and $d_{i-1}, \ldots, d_j$.

When $j = i - 1$ we only have d_{i-1} in the subtree, so $e[i, i-1] = q_{i-1}$. (i.e. no keys)

When $j \geq i$ we shall choose a root k_r among $k_i, \ldots, k_j$, and then form the optimal binary search tree with $k_i, \ldots, k_{r-1}$ in left subtree, and $k_{r+1}, \ldots, k_j$ in right subtree.

For a subtree with keys $k_i, \ldots, k_j$ let $w(i,j)$ be the sum of its probabilities:

$$w(i,j) = \sum_{\ell=i}^{j} p_\ell + \sum_{\ell=i-1}^{j} q_\ell$$

So if k_r is the root in an optimal subtree with $k_i, \ldots, k_j$ we get

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

Terms

$$\begin{aligned} w(i, r-1) &= p_i + \cdots + p_{r-1} + q_{i-1} + \cdots + q_{r-1} \\ w(r+1, j) &= p_{r+1} + \cdots + p_j + q_r + \cdots + q_j \end{aligned}$$

are added since the depths of nodes in the two subtrees of k_r *increase by* 1.

As $w(i,j) = w(i, r-1) + p_r + w(r+1, j)$, this results in:

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

which gives a recursive formula similar to the one for matrix-chain multiplication:

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

Step 3: Computing the expected search cost of an OBST

Successively build larger optimal subtrees (where subtrees with one key are trivially optimal):

- Compute optimal trees with m keys by choosing as root the one that gives the cheapest partition into two smaller subtrees, whose optimal structures have already been computed.
- Repeat until $m = n$.

Total time is $\Theta(n^3)$. This can be reduced to $\Theta(n^2)$, which we show on tutorial 2.

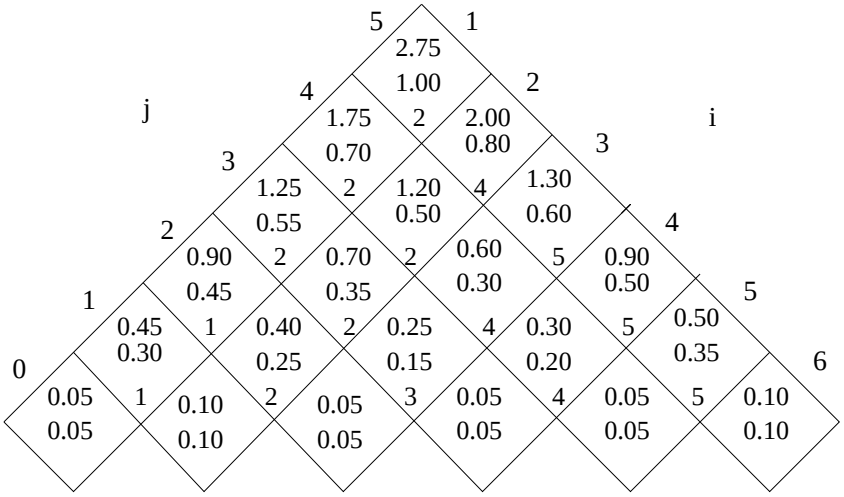
To avoid computing the $w(i,j)$ from scratch each time, we store them in a table $w[1..n+1, 0..n]$, where $w[1,0] = q_0$ and $w[n+1,n] = q_n$.

We can then update the values $w[i,j]$ successively given the base case $w[i,i-1] = q_{i-1}$ for $1 \le i \le n+1$:

$$w[i,j] = w[i,j-1] + p_j + q_j, \;\; i \le j$$

This is constant work each time and hence $\Theta(n^2)$ in total.

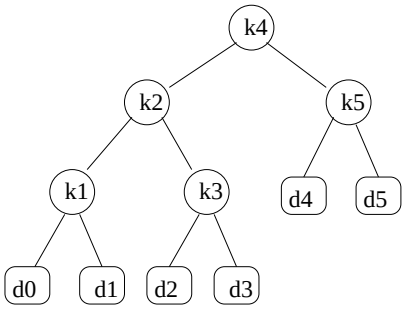
The following figure gives $e[i,j], w[i,j]$ and the root in an optimal binary search tree with probabilities as above.



For example, the values in the root are given by

$$\begin{aligned} e[1,5] &= w[1,5] + \min(e[1,0]+e[2,5], e[1,1]+e[3,5], e[1,2]+e[4,5], e[1,3]+e[5,5], e[1,4]+e[6,5]) \\ &= 1.00 + \min(0.05+2.00,\ 0.45+1.30,\ 0.90+0.90,\ 1.25+0.50,\ 1.75+0.10) \\ &= 1.00 + \min(2.05,\ 1.75,\ 1.80,\ 1.75,\ 1.85) = 2.75 \end{aligned}$$

Note that two trees have $e[1,5] = 2.75$. In addition to the one to the right in the figure on page 3 also the following tree:



Check:

$$\begin{aligned} E[\text{search cost}] &= 4(d_0+d_1+d_2+d_3) + 3(d_4+d_5) + 3(k_1+k_3) + 2(k_2+k_5) + k_4 \\ &= 4 \cdot 0.25 + 3 \cdot 0.15 + 3 \cdot 0.20 + 2 \cdot 0.30 + 0.10 = 2.75 \end{aligned}$$

Special case: If there are no unsuccessful searches, i.e. $q_i = 0$, the problem is to minimize

$$1 + \sum_{i=1}^{n} \text{depth}(k_i) \cdot p_i$$