CS124	**Lecture 11**	**Spring 2015**

This lecture was on even more dynamic programming problems! Below we will see some more examples.

11.1 Binomial coefficients

Recall that the binomial coefficient $\binom{n}{k}$, "n choose k", is the number of binary strings of length n with exactly k 1's. We have that $\binom{n}{k} = 0$ if $k > n$ and $\binom{n}{0} = 1$ for any nonnegative integer n. For all other cases, we have the recurrence

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

This is because the first bit of the binary string can either be a 1 or 0, so we sum over both possibilities. If it is a 1, then the remaining $n-1$ characters of the string should have $k-1$ ones, and otherwise it should have k ones. Thus with memoization, we see that the space to compute $\binom{n}{k}$, ignoring bit complexity, is the total number of inputs which is $\Theta(nk)$. The time, assuming arbitrary precision addition in constant time for simplicity, is also $\Theta(nk)$.

With a bottom-up approach though, using dynamic programming, we note that $\binom{a}{\cdot}$ values only depend on $\binom{a-1}{\cdot}$ values. Thus when building up our table of values for $\binom{a}{\cdot}$, we can throw away all values stored for $\binom{a-2}{\cdot}$ and below. Thus the space in a bottom-up approach can be improved to $\Theta(k)$.

11.2 Matrix chain multiplication

In this problem we would like to multiply k matrices $A_1 \times A_2 \times \ldots \times A_k$. We assume the matrix A_i is of dimension $n_i \times n_{i+1}$. Since matrix multiplication is associative, we can choose the order of multiplication. For example, $A_1 \times A_2 \times A_3 = (A_1 \times A_2) \times A_3 = A_1 \times (A_2 \times A_3)$. The straightforward way to multiply a $p \times q$ matrix by a $q \times r$ matrix takes $\Theta(pqr)$ time (nested for loops), and the total time to multiply the k matrices could depend on the order of multplication. For example consider the case $k = 3$ and $n_1 = 2, n_2 = 3, n_3 = 4, n_4 = 5$. Then multiplying $A_1 \times A_2$ first takes time $2 \cdot 3 \cdot 4 = 24$. Then multiplying the result times A_3 takes time $2 \cdot 4 \cdot 5 = 40$, so the total time is $24 + 40 = 64$. Meanwhile, multiplying $A_2 \times A_3$ first takes time $3 \cdot 4 \cdot 5 = 60$. Then multiplying A_1 times the result takes time $2 \cdot 3 \cdot 5 = 30$. Thus the total time is 90. Thus, $(A_1 \times A_2) \times A_3$ is more efficient in this case.

How can we figure out, given k and $n_1, \ldots, n_{k+1}$ as input, the optimal parenthesization to minimize the total cost of performing all multiplications? You guessed it: recursion and memoization!

Define $f(i,j)$ as the minimum cost to multiply $A_i \times A_{i+1} \times \ldots \times A_j$ so that we actually care to compute $f(1,k+1)$. Then we have the recurrence

$$f(i,j) = \begin{cases} 0, & \text{if } i = j \\ \min_{i \le r < j} \left[f(i,r) + f(r+1,j) + n_i n_{r+1} n_{j+1} \right], & \text{otherwise} \end{cases}.$$

The base case is when the product consists of a single matrix ($i = j$), so no work need to be done. Otherwise, if $j > i$, then in the optimal solution one of the $j - i$ products, call it the rth product, is the last product to be computed. We don't know which r is optimal, so we simply try all possible choices of r. That is, we try performing our multiplication as $(A_i \times A_{i+1} \times \ldots \times A_r) \times (A_{r+1} \times A_{r+2} \times \ldots \times A_j)$. Then we need to figure out the optimal way to multiply the matrices between A_i and A_r, add that to the optimal way to multiply the matrices between A_{r+1} and A_j, then sum these with the cost $n_i n_{r+1} n_{j+1}$ of performing the final product.

With memoization, the space is $\Theta(k^2)$ and the time is $\Theta(k^3)$.

11.3 Optimal Binary Search Tree (BST)

Suppose we have n elements in a database with key values $k_1 < k_2 < \ldots < k_n$. We also imagine we have dummy nodes $d_0 < k_1 < d_1 < k_1 < \ldots < k_n < d_n$ whose values interlace the key values of the n items in the database (when someone searches for a key value v not in the database, we imagine they arrive at the dummy node that lies between the two database key values sandwiching v, or at d_0 if $v < k_1$ or d_n if $v > k_n$). We would like to build a binary search tree on the $2n+1$ items here; the set of all keys k_i as well as dummy nodes d_i. Note the dummy nodes are the leaves, and the internal nodes are the keys k_i of the actual database items. If we built a perfect binary tree on these $2n+1$ items, we know that searching for an item would never take more than $O(\log n)$ time. But what if we don't care about *worst case* query time, but rather know just how many times each of these $2n+1$ items are queried?

Suppose we have some query logs in which we find that K_i is the number of times the key k_i was queried, and D_i is the number of times d_i was queried. We would like to build a binary search tree that would minimize the total cost of servicing all the queries in that log. If we measure the "cost" of querying a node u as the number of nodes touched on the root-to-leaf path in the binary search tree when searching for u, and we measure total cost $cost(T)$ of a binary search tree T as a sum over all u of the cost of querying u in that tree, then we have

$$cost(T) = \left(\sum_{i=1}^{n} (1 + depth_T(k_i)) \cdot K_i \right) + \left(\sum_{i=0}^{n} (1 + depth_T(d_i)) \cdot D_i \right)$$

where $depth_T(k_i)$ is the depth of the node containing key value k_i in the tree T (and likewise for d_i).

Now let us come up with a recursive solution for finding the binary search tree T of optimal cost. Then, as usual, we will obtain fast running time using memoization.

Define $f(i,j)$ to be the minimum cost of a tree built on the nodes $d_{i-1} < k_i < d_i < k_{i+1} < d_{i+1} < \ldots < k_j < d_j$. Thus the value we actually want is $f(1,n)$. Furthermore, we have the recurrence

$$f(i,j) = \begin{cases} D_{i-1}, & \text{if } j = i-1 \\ \left(\sum_{t=i}^{j} K_t\right) + \left(\sum_{t=i-1}^{j} D_t\right) + \min_{i \le r \le j}\left[f(i,r-1) + f(r+1,j)\right], & \text{else} \end{cases}.$$

The base case of the recurrence is when there is a single node, namely the node d_{i-1}. In the case $j \ge i$, we have at least one database key value (namely k_i) in our set of nodes. In this case, to find the best binary search tree, we note that the best tree must have *some* database key value k_r as its root for $i \le r \le j$. Thus we simply *try all possibilities* and pick the best one. Then we have to recursively build the left subtree on $d_{i-1} < k_i < \ldots < k_{r-1} < d_{r-1}$ and the right subtree on $d_r < k_{r+1} < \ldots < k_j < d_j$.

Now, if we memoize the computation of f then the total space will be $\Theta(n^2)$ (the number of (i,j) pairs). The running time would be $\Theta(n^3)$. To see that it is $O(n^3)$, note that for each (i,j) pair we do $O(j-i+1) = O(n)$ work. To see that it is $\Omega(n^3)$, note that for $\Omega(n^2)$ pairs (i,j) we have that $j - i \ge n/2$.

11.4 Longest Common Subsequence

Given a string $s = s_0 s_1 \cdots s_{n-1}$, a *subsequence* of s is a string that can be formed as $s_{i_0} s_{i_1} \cdots s_{i_t}$ for $0 \le i_0 < i_1 < \ldots < i_t < n$. For example, "pal" is a subsequence of "pineapple". Given two strings $s = s_0 s_1 \cdots s_{n-1}$ and $t = t_0 t_1 \cdots t_{m-1}$, a *common subsequence* of s,t is a string which appears as a subsequence of both. In the *longest common subsequence* (LCS) problem we would like to find a common subsequence of s,t which is as long as possible. Again, this problem can be solved by (surprise surprise), memoization/dynamic programming!

Define $f(i,j)$ as the length of the longest common subsequence of the two strings $s[i:] = s_i s_{i+1} \cdots s_{n-1}$ and $t[j:] = t_j t_{j+1} \cdots t_{m-1}$, so that we ultimately want $f(0,0)$. Then we have the recurrence relation

$$f(i,j) = \begin{cases} 0, & \text{if } i = n \text{ or } j = m \\ \max\{f(i+1,j), f(i,j+1)\}, & \text{if } s_i \ne t_j \\ 1 + f(i+1,j+1) & \text{if } s_i = t_j \end{cases}.$$

The idea is that in the base case, one of the two strings is the empty string and thus the empty sequence is the LCS. In the case when $s_i \ne t_j$, we know that the LCS either does not contain s_i (the first term in the max) or does not contain t_j (the second term), so we take the best of those two options. In the case $s_i = t_j$, then the first character of the two strings is in the LCS. Using memoization, we can compute $f(0,0)$ in $\Theta(nm)$ time and space.

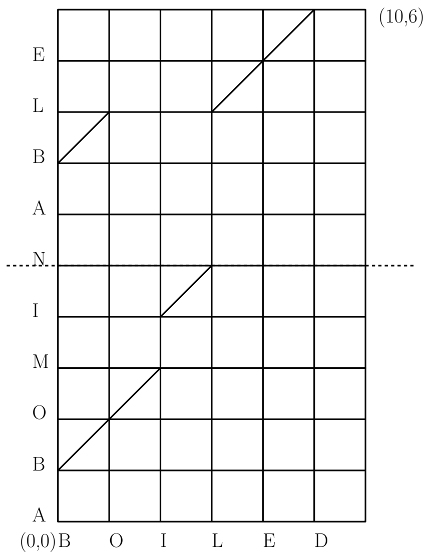

Figure 11.1: Want to find longest path from $(0,0)$ to $(11,6)$ in graph comparing BOILED to ABOMINABLE.

Now, just as in previous problems we can augment the lookup table in the memoization so that we actually remember what choice we made in each recursive step to achieve the max. In this way we could not only obtain the length of the LCS, but also achieve the actual subsequence itself in the same space and time.

We could also build the lookup table bottom-up using dynamic programming. Furthermore note that to compute $f(i,\cdot)$ values, we only need $f(i+1,\cdot)$ values and other $f(i,\cdot)$ values. Thus if we imagine the lookup table as being drawn in a 2-dimensional grid, where i labels a row and j labels a column, what we find is that to fill in the values in one row, we only need other values in that row plus the row immediately above. Thus we can make the space be $O(m)$ (in fact the roles of s,t can be reversed so that we can achieve space $O(\min\{n,m\})$). There is a problem though: this space-saving trick and the observation in the last paragraph of remembering choices we made don't go well together! If we throw away all information from all previous rows except the last one, when we finally come to computing $f(0,0)$, we will not be able to backtrack through the solution table to figure out what the actual LCS is. Though, there is a way around this ...

11.4.1 Hirschberg's algorithm

Hirschberg discovered an algorithm for simultaneously achieving $O(nm)$ time and $O(\min\{n,m\})$ space for the LCS problem (we will describe the $O(m)$ space solution, but $\min\{n,m\}$ space can be achieved by reversing the roles of s and t). The idea is to combine bottom-up dynamic programming with a technique we've seen before: *divide and conquer*.

The key is to realize that what we actually care about is finding the longest path between two vertices in a two-dimensional grid graph (see Figure 11.1). Namely, we would like to find the longest path from $(0,0)$ to (n,m) where all edges go either to the right or upward (or diagonal right-upward in the case of diagonal edges). Diagonal edges have weight 1, and all other edges have weight 0.

Note that $f(i,j)$ is the length of the longest path from (i,j) to (n,m), which we showed in the last section how to compute in $O((n-i+1)(m-j+1))$ time and $O(m-j+1)$ space. Also let $g(i,j)$ be the length of the LCS between $s[:i] = s_0 s_1 \cdots s_i$ and $t[:j] = t_0 t_1 \cdots t_j$. Then similarly as in the last section, we can compute $g(i,j)$ in $O((i+1)(j+1))$ time and $O(j+1)$ space. Now we use divide and conquer!

The optimal path from $(0,0)$ to (n,m) must pass through $(n/2,q)$ for *some* value of q. Thus we simply try all possibilities for q and take the best one! That is, we compute $f(n/2,t), g(n/2,t)$ for all values $0 \le t \le m$ and we choose q so that $f(n/2,q) + g(n/2,q)$ is maximized. Then we know that the sequence of nodes traversed in the optimal path from $(0,0)$ to (n,m) is the sequence of nodes traversed in the optimal path from $(0,0)$ to $(n/2,q)$ (recursion!), followed by the node $(n/2,q)$, followed by the nodes traversed in the optimal path from $(n/2,q)$ to (n,m) (recursion again!). The space in this divide and conquer is $\Theta(m)$ (since when computing the various values of f,g, we can reuse the space used to compute previous values). The running time $T(n,m)$ is $\Theta(m)$ for $n \le 2$ and $\Theta(n)$ for $m \le 2$, and otherwise satisfies the recurrence

$$T(n,m) \le cmn + T(n/2,q) + T(n/2,m-q)$$

for some constant $c > 0$. The cmn term comes from computing all the $f(n/2,t), g(n/2,t)$ values. As an exercise, you should show (by induction!) that the solution to this recurrence satisfies $T(n,m) = O(nm)$.