ICS 161: Design and Analysis of Algorithms
Lecture notes for February 29, 1996

Longest Common Subsequences

In this lecture we examine another string matching problem, of finding the longest common subsequence of two strings.

This is a good example of the technique of *dynamic programming*, which is the following very simple idea: start with a recursive algorithm for the problem, which may be inefficient because it calls itself repeatedly on a small number of subproblems. Simply remember the solution to each subproblem the first time you compute it, then after that look it up instead of recomputing it. The overall time bound then becomes (typically) proportional to the number of distinct subproblems rather than the larger number of recursive calls. We already saw this idea briefly in the first lecture.

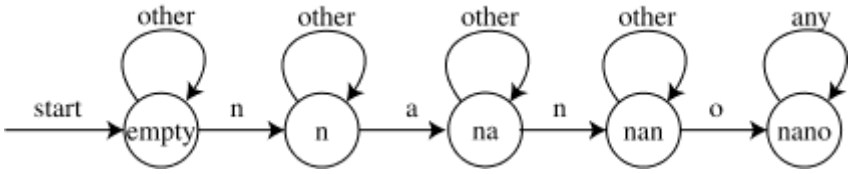
As we'll see, there are two ways of doing dynamic programming, top down and bottom-up. The top down (memoizing) method is closer to the original recursive algorithm, so easier to understand, but the bottom up method is usually a little more efficient.

Subsequence testing

Before we define the longest common subsequence problem, let's start with an easy warmup. Suppose you're given a short string (pattern) and long string (text), as in the string matching problem. But now you want to know if the letters of the pattern appear in order (but possibly separated) in the text. If they do, we say that the pattern is a *subsequence* of the text.

As an example, is "nano" a subsequence of "nematode knowledge"? Yes, and in only one way. The easiest way to see this example is to capitalize the subsequence: "NemAtode kNOwledge".

In general, we can test this as before using a finite state machine. Draw circles and arrows as before, corresponding to partial subsequences (prefixes of the pattern), but now there is no need for backtracking.



Equivalently, it is easy to write code or pseudo-code for this:

```
subseq(char * P, char * T)
{
    while (*T != '\0')
        if (*P == *T++ && *++P == '\0')
            return TRUE;
    return FALSE;
}
```

Longest common subsequence problem

What if the pattern does not occur in the text? It still makes sense to find the longest subsequence that occurs both in the pattern and in the text. This is the *longest common subsequence problem*. Since the pattern and text have symmetric roles, from now on we won't give them different names but just call them strings A and B. We'll use m to denote the length of A and n to denote the length of B.

Note that the automata-theoretic method above doesn't solve the problem -- instead it gives the longest prefix of A that's a subsequence of B. But the longest common subsequence of A and B is not always a prefix of A.

Why might we want to solve the longest common subsequence problem? There are several motivating applications.

- Molecular biology. DNA sequences (genes) can be represented as sequences of four letters ACGT, corresponding to the four submolecules forming DNA. When biologists find a new sequences, they typically want to know what other sequences it is most similar to. One way of computing how similar two sequences are is to find the length of their longest common subsequence.

- File comparison. The Unix program "diff" is used to compare two different versions of the same file, to determine what changes have been made to the file. It works by finding a longest common subsequence of the lines of the two files; any line in the subsequence has not been changed, so what it displays is the remaining set of lines that have changed. In this instance of the problem we should think of each line of a file as being a single complicated character in a string.

- Screen redisplay. Many text editors like "emacs" display part of a file on the screen, updating the screen image as the file is changed. For slow dial-in terminals, these programs want to send the terminal as few characters as possible to cause it to update its display correctly. It is possible to view the computation of the minimum length sequence of characters needed to update the terminal as being a sort of common subsequence problem (the common subsequence tells you the parts of the display that are already correct and don't need to be changed).

(As an aside, it is natural to define a similar *longest common substring* problem, asking for the longest substring that appears in two input strings. This problem can be solved in linear time using a data structure known as the *suffix tree* but the solution is extremely complicated.)

Recursive solution

So we want to solve the longest common subsequence problem by dynamic programming. To do this, we first need a recursive solution. The dynamic programming idea doesn't tell us how to find this, it just gives us a way of making the solution more efficient once we have.

Let's start with some simple observations about the LCS problem. If we have two strings, say "nematode knowledge" and "empty bottle", we can represent a subsequence as a way of writing the two so that certain letters line up:

```
n e m a t o d e  k n o  w  l e d g e
  | |   |        |      |     | |
  e m p t y        b o t t l e
```

If we draw lines connecting the letters in the first string to the corresponding letters in the second, no two lines cross (the top and bottom endpoints occur in the same order, the order of the letters in the subsequence). Conversely any set of lines drawn like this, without crossings, represents a subsequence.

From this we can observe the following simple fact: if the two strings start with the same letter, it's always safe to choose that starting letter as the first character of the subsequence. This is because, if you have some other subsequence, represented as a collection of lines as drawn above, you can "push" the leftmost line to the start of

the two strings, without causing any other crossings, and get a representation of an equally-long subsequence that does start this way.

On the other hand, suppose that, like the example above, the two first characters differ. Then it is not possible for both of them to be part of a common subsequence - one or the other (or maybe both) will have to be removed.

Finally, observe that once we've decided what to do with the first characters of the strings, the remaining subproblem is again a longest common subsequence problem, on two shorter strings. Therefore we can solve it recursively.

Rather than finding the subsequence itself, it turns out to be more efficient to find the length of the longest subsequence. Then in the case where the first characters differ, we can determine which subproblem gives the correct solution by solving both and taking the max of the resulting subsequence lengths. Once we turn this into a dynamic program we will see how to get the sequence itself.

These observations give us the following, very inefficient, recursive algorithm.

Recursive LCS:

```
int lcs_length(char * A, char * B)
{
    if (*A == '\0' || *B == '\0') return 0;
    else if (*A == *B) return 1 + lcs_length(A+1, B+1);
    else return max(lcs_length(A+1,B), lcs_length(A,B+1));
}
```

This is a correct solution but it's very time consuming. For example, if the two strings have no matching characters, so the last line always gets executed, the the time bounds are binomial coefficients, which (if m=n) are close to 2^n.

Memoization

The problem with the recursive solution is that the same subproblems get called many different times. A subproblem consists of a call to lcs_length, with the arguments being two suffixes of A and B, so there are exactly (m+1)(n+1) possible subproblems (a relatively small number). If there are nearly 2^n recursive calls, some of these subproblems must be being solved over and over.

The dynamic programming solution is to check whenever we want to solve a subproblem, whether we've already done it before. If so we look up the solution instead of recomputing it. Implemented in the most direct way, we just add some code to our recursive algorithm to do this look up -- this "top down", recursive version of dynamic programming is known as "memoization".

In the LCS problem, subproblems consist of a pair of suffixes of the two input strings. To make it easier to store and look up subproblem solutions, I'll represent these by the starting positions in the strings, rather than (as I wrote it above) character pointers.

Recursive LCS with indices:

```
char * A;
char * B;
int lcs_length(char * AA, char * BB)
{
    A = AA; B = BB;
    return subproblem(0, 0);
}
int subproblem(int i, int j)
```

```
{
    if (A[i] == '\0' || B[j] == '\0') return 0;
    else if (A[i] == B[j]) return 1 + subproblem(i+1, j+1);
    else return max(subproblem(i+1, j), subproblem(i, j+1));
}
```

Now to turn this into a dynamic programming algorithm we need only use an array to store the subproblem results. When we want the solution to a subproblem, we first look in the array, and check if there already is a solution there. If so we return it; otherwise we perform the computation and store the result. In the LCS problem, no result is negative, so we'll use -1 as a flag to tell the algorithm that nothing has been stored yet.

Memoizing LCS:

```
char * A;
char * B;
array L;

int lcs_length(char * AA, char * BB)
{
    A = AA; B = BB;
    allocate storage for L;
    for (i = 0; i <= m; i++)
        for (j = 0; j <= m; j++)
            L[i,j] = -1;

    return subproblem(0, 0);
}

int subproblem(int i, int j)
{
    if (L[i,j] < 0) {
        if (A[i] == '\0' || B[j] == '\0') L[i,j] = 0;
        else if (A[i] == B[j]) L[i,j] = 1 + subproblem(i+1, j+1);
        else L[i,j] = max(subproblem(i+1, j), subproblem(i, j+1));
    }
    return L[i,j];
}
```

Time analysis: each call to subproblem takes constant time. We call it once from the main routine, and at most twice every time we fill in an entry of array L. There are (m+1)(n+1) entries, so the total number of calls is at most 2(m+1)(n+1)+1 and the time is O(mn).

As usual, this is a worst case analysis. The time might sometimes better, if not all array entries get filled out. For instance if the two strings match exactly, we'll only fill in diagonal entries and the algorithm will be fast.

Bottom up dynamic programming

We can view the code above as just being a slightly smarter way of doing the original recursive algorithm, saving work by not repeating subproblem computations. But it can also be thought of as a way of computing the entries in the array L. The recursive algorithm controls what order we fill them in, but we'd get the same results if we filled them in in some other order. We might as well use something simpler, like a nested loop, that visits the array systematically. The only thing we have to worry about is that when we fill in a cell L[i,j], we need to already know the values it depends on, namely in this case L[i+1,j], L[i,j+1], and L[i+1,j+1]. For this reason we'll traverse the array backwards, from the last row working up to the first and from the last column working up to the first. This is *iterative* (because it uses nested loops instead of recursion) or *bottom up* (because the order we fill in the array is from smaller simpler subproblems to bigger more complicated ones).

Iterative LCS:

```
int lcs_length(char * A, char * B)
{
    allocate storage for array L;
    for (i = m; i >= 0; i--)
        for (j = n; j >= 0; j--)
        {
            if (A[i] == '\0' || B[j] == '\0') L[i,j] = 0;
            else if (A[i] == B[j]) L[i,j] = 1 + L[i+1, j+1];
            else L[i,j] = max(L[i+1, j], L[i, j+1]);
        }
    return L[0,0];
}
```

Advantages of this method include the fact that iteration is usually faster than recursion, we don't need to initialize the matrix to all -1's, and we save three if statements per iteration since we don't need to test whether L[i,j], L[i+1,j], and L[i,j+1] have already been computed (we know in advance that the answers will be no, yes, and yes). One disadvantage over memoizing is that this fills in the entire array even when it might be possible to solve the problem by looking at only a fraction of the array's cells.

The subsequence itself

What if you want the subsequence itself, and not just its length? This is important for some but not all of the applications we mentioned. Once we have filled in the array L described above, we can find the sequence by working forwards through the array.

```
sequence S = empty;
i = 0;
j = 0;
while (i < m && j < n)
{
    if (A[i]==B[j])
    {
        add A[i] to end of S;
        i++; j++;
    }
    else if (L[i+1,j] >= L[i,j+1]) i++;
    else j++;
}
```

Let's see an example of this. Here's the array for the earlier example:

	n	e	m	a	t	o	d	e	_	k	n	o	w	l	e	d	g	e	
e	7	7	6	5	5	5	5	5	4	3	3	3	2	2	2	1	1	1	0
m	6	6	6	5	5	4	4	4	4	3	3	3	2	2	1	1	1	1	0
p	5	5	5	5	5	4	4	4	4	3	3	3	2	2	1	1	1	1	0
t	5	5	5	5	5	4	4	4	4	3	3	3	2	2	1	1	1	1	0
y	4	4	4	4	4	4	4	4	4	3	3	3	2	2	1	1	1	1	0
_	4	4	4	4	4	4	4	4	4	3	3	3	2	2	1	1	1	1	0
b	3	3	3	3	3	3	3	3	3	3	3	3	2	2	1	1	1	1	0
o	3	3	3	3	3	3	3	3	3	3	3	3	2	2	1	1	1	1	0
t	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	0
t	3	3	3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	0
l	2	2	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	0
e	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(you can check that each entry is computed correctly from the entries below and to the right.)

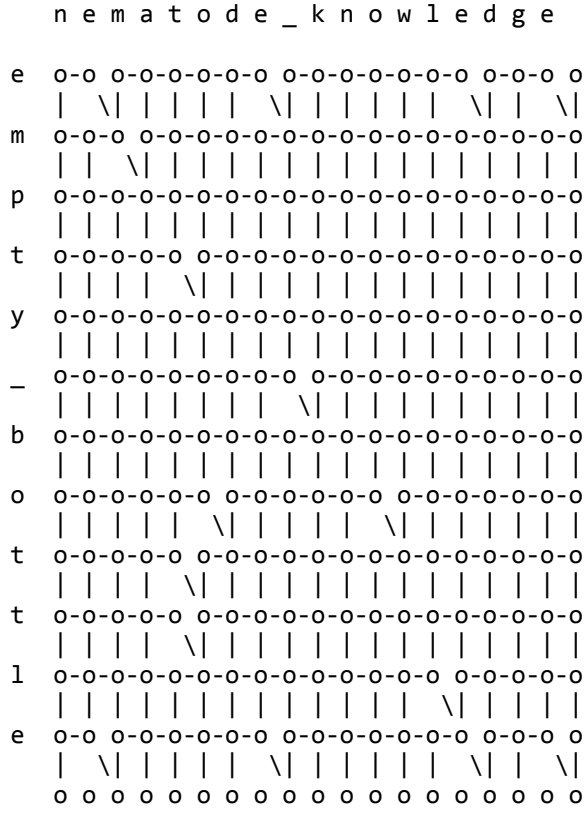
To find the longest common subsequence, look at the first entry L[0,0]. This is 7, telling us that the sequence has seven characters. L[0,0] was computed as max(L[0,1],L[1,0]), corresponding to the subproblems formed by

deleting either the "n" from the first string or the "e" from the second. Deleting the "n" gives a subsequence of length L[0,1]=7, but deleting the "e" only gives L[1,0]=6, so we can only delete the "n". Now let's look at the entry L[0,1] coming from this deletion. A[0]=B[1]="e" so we can safely include this "e" as part of the subsequence, and move to L[1,2]=6. Similarly this entry gives us an "m" in our sequence. Continuing in this way (and breaking ties as in the algorithm above, by moving down instead of across) gives the common subsequence "emt ole".

So we can find longest common subsequences in time O(mn). Actually, if you look at the matrix above, you can tell that it has a lot of structure -- the numbers in the matrix form large blocks in which the value is constant, with only a small number of "corners" at which the value changes. It turns out that one can take advantage of these corners to speed up the computation. The current (theoretically) fastest algorithm for longest common subsequences (due to myself and co-authors) runs in time O(n log s + c log log min(c,mn/c)) where c is the number of these corners, and s is the number of characters appearing in the two strings.

Relation to paths in graphs

Let's draw a directed graph, with vertices corresponding to entries in the array L, and an edge connecting an entry to one it depends on: either one edge to L[i+1,j+1] if A[i]=B[j], or two edges to L[i+1,j] and L[i,j+1] otherwise. Don't draw edges from the bottom right fringe of the array (since those entries don't depend on any others).



Then if you look at any path in the graph, the diagonal edges form a subsequence of the two strings. Conversely, if you define the horizontal and vertical edges to have length zero, and the diagonal edges to have length one, the longest common subsequence corresponds to the longest path from the top left corner to one of the bottom right vertices. This graph is acyclic, so we can compute longest paths in time linear in the size of the graph, here O(mn).

Where did these edge lengths come from? They're just the amount by which the LCS length increases compared to the length at the corresponding subproblem. If A[i]=B[j], then L[i,j]=L[i+1,j+1]+1, and we use that last "+1" as the edge length. Otherwise, L[i,j]=max(L[i+1,j],L[i,j+1])+0, so we use zero as the edge length.

This sort of phenomenon, in which a dynamic programming problem turns out to be equivalent to a shortest or longest path problem, does not always happen with other problems, but it is reasonably common. This idea doesn't really help compute the single longest common subsequence, but one of my papers uses similar graph-theoretic ideas to find multiple long common subsequences (and multiple solutions to many other problems).

Reduced space complexity

One disadvantage of the dynamic programming methods we've described, compared to the original recursion, is that they use a lot of space: O(mn) for the array L (the recursion only uses O(n+m)). But the iterative version can be easily modified to use less space -- the observation is that once we've computed row i of array L, we no longer need the values in row i+1.

Space-efficient LCS:

```
int lcs_length(char * A, char * B)
{
    allocate storage for one-dimensional arrays X and Y
    for (i = m; i >= 0; i--)
    {
        for (j = n; j >= 0; j--)
        {
            if (A[i] == '\0' || B[j] == '\0') X[j] = 0;
            else if (A[i] == B[j]) X[j] = 1 + Y[j+1];
            else X[j] = max(Y[j], X[j+1]);
        }
        Y = X;
    }
    return X[0];
}
```

This takes roughly the same amount of time as before, O(mn) -- it uses a little more time to copy X into Y but this only increases the time by a constant (and can be avoided with some more care). The space is either O(m) or O(n), whichever is smaller (switch the two strings if necessary so there are more rows than columns). Unfortunately, this solution does not leave you with enough information to find the subsequence itself, just its length.

In 1975, Dan Hirschberg showed how to find not just the length, but the longest common subsequence itself, in linear space and O(mn) time. The idea is as above to use one-dimensional arrays X and Y to store the rows of the larger two dimensional array L. But Hirschberg's method treats the middle row of array L specially: for all i<m/2, he stores along with the numbers in X and Y the place where some path (corresponding to a subsequence with that many characters) crosses the middle row. These crossing places can be updated along with the array values, by copying them from X[j+1], Y[j], or Y[j+1] as appropriate.

Then when the algorithm above has finished with the LCS length in X[0], Hirschberg finds the corresponding crossing place (m/2,k). He then solves recursively two LCS problems, one for A[0..m/2-1] and B[0..k-1] and one for A[m/2..m] and B[k..n]. The longest common subsequence is the concatenation of the sequences found by these two recursive calls.

It is not hard to see that this method uses linear space. What about time complexity? This is a recursive algorithm, with a time recurrence

```
T(m,n) = O(mn) + T(m/2,k) + T(m/2,n-k)
```

You can think of this as sort of like quicksort -- we're breaking both strings into parts. But unlike quicksort it doesn't matter that the second string can be broken unequally. No matter what k is, the recurrence solves to O(mn). The easiest way to see this is to think about what it's doing in the array L. The main part of the algorithm visits the whole array, then the two calls visit two subarrays, one above and left of (m/2,k) and the other below

and to the right. No matter what k is, the total size of these two subarrays is roughly mn/2. So instead we can write a simplified recurrence

```
T(mn) = O(mn) + T(mn/2)
```

which solves to O(mn) time total.
