CS1510 Greedy 14, Dynamic Programming 2, 3, & 4

Rebecca Negley, Sean Myers

September 19, 2011

14. Consider the generalization of the bridge crossing problem where n people with speeds $s_1,\ldots,s_n$ wish to cross the bridge as quickly as possible. Give a linear time algorithm to find the fastest way to get a group of people across the bridge. You must have a proof of correctness for your method.

    Solution: Consider the following algorithm:

    If there are fewer than 3 people, have everyone cross on the first trip.

    If there are 3 or more people, consider $\alpha(i)$ as the index associated with the ith fastest person, so that $s_{\alpha(1)}$ is the speed of the fastest person, $s_{\alpha(2)}$ the speed of the second fastest person, etc. Let p_i be the person with speed s_i, and let t_i be the time it takes person p_i with speed s_i to cross the bridge. Ties are broken arbitrarily. In the first step, have $p_{\alpha(1)}$ and $p_{\alpha(2)}$ cross the bridge together. Then, while the fastest remaining person on the original side, $p_{\alpha(j)}$, satisfies $t_{\alpha(1)} + t_{\alpha(j)} \leq 2 * t_{\alpha(2)}$, have person $p_{\alpha(1)}$ return and cross again with $p_{\alpha(j)}$. Let k be the highest index such that $t_{\alpha(1)} + t_{\alpha(k)} \leq 2 * t_{\alpha(2)}$. If $(n-k)$ is odd, have $p_{\alpha(1)}$ return again and cross again with $p_{\alpha(k+1)}$. Next, while there are still people on the original side of the bridge, do the following: Have $p_{\alpha(1)}$ cross again and send the two fastest people other than him/herself, $p_{\alpha(l)}$ and $p_{\alpha(l+1)}$, across together. Then have $p_{\alpha(2)}$ return and cross again with $p_{\alpha(1)}$. Repeat until everyone has crossed.

    Observe that this algorithm runs in linear time. If $n \geq 3$, it makes $(n-1)$ trips across the bridge and $(n-2)$ trips back. Also, if $n < 3$, the algorithm is obviously correct.

    Observe that this algorithm gives a total time of $t_{\alpha(n)} + t_{\alpha(n-2)} + \ldots + t_{\alpha(n-2*l)} + t_{\alpha(n-2*l-1)} + t_{\alpha(n-2*l-2)} + \ldots + t_{\alpha(3)} + (2*l+1)t_{\alpha(2)} + (l + (n - 2*l - 2))t_{\alpha(1)} = t_{\alpha(n)} + t_{\alpha(n-2)} + \ldots + t_{\alpha(n-2*l)} + t_{\alpha(n-2*l-1)} + t_{\alpha(n-2*l-2)} + \ldots + t_{\alpha(3)} + (2*l+1)t_{\alpha(2)} + (n-l-2)t_{\alpha(1)}$ where l satisfies $t_{\alpha(1)} + t_{\alpha(n-2*(l+1))} \leq t_{\alpha(2)}$ and $t_{\alpha(1)} + t_{\alpha(n-2*(l-1))} > t_{\alpha(2)}$. Since the optimal solution must have two people cross forward and one back on every trip (so that progress is being made), it must also have $(n-1)+(n-2) = 2n-3$ crossings. Suppose this algorithm is not optimal for some input. Then, any optimal solution must have a sum of $2n-3$ terms (each being the time of a crossing) with a total less than above. Order these terms from greatest to least in each optimal solution. Let $OPT(I)$ be the optimal solution that agrees with the greedy sum for the greatest number of terms before disagreement. Then, let the kth term be the first term where the greedy sum and the $OPT(I)$ sum disagree. Let the kth term in the $OPT(I)$ sum be t_a and the kth term in the greedy sum be t_b. First examine the case where $(k-1) \leq l$ so that b satisfies $b = \alpha(n - 2(k-1))$. Suppose $t_b > t_a$. Clearly, at most $2(k-1)$ people can cross in the first $(k-1)$ terms. (Observe that this is not the order the people crossed, but rather each crossing time ordered from highest to lowest.) However, there are at least $2(k-1)+1$ people with crossing times greater than t_a (because $t_b > t_a$). Therefore, not all of these people could cross according to the optimal solution, which is a contradiction. Suppose then that $t_b < t_a$. Since $b = \alpha(n-2(k-1))$, we get $t_{\alpha(n-2(k-2))} \geq t_a > t_b$ from the ordering and the first $k-1$ terms being the same. It cannot be optimal for t_a to simply replace t_b because the optimal sum would be greater

than the greedy sum. The only way t_a can be in the optimal sum at position k is if it is more optimal for $p_{\alpha(n-2(k-2))}$ to cross with someone faster than p_a than for $p_{\alpha(n-2(k-2))}$ to cross with p_a. The only reason it would be more optimal for $p_{\alpha(n-2(k-2))}$ to cross with someone faster would be because the faster person's return time is so much better than that of the next fastest person on the return side that it is worth incurring the cost of the slower people walking over separately. That faster partner could at best be $p_{\alpha(1)}$. However, $t_{\alpha(n-2(k-2))} + t_{\alpha(1)} > 2t_{\alpha(2)}$, so it is not worth incurring the cost, which is a contradiction.

Now suppose $l + 1 < k < n - l - 2$. If $t_b > t_a$, the optimal solution must have had p_b walk over with someone slower. If p_b walks over with $p_{\alpha(n-2*j)}$ for some j such that $0 \leq j < l$, then $p_{\alpha(n-2*j-1)}$ has no one to walk with, a contradiction. Then, p_b must walk over with some $p_{\alpha(n-2*l-m)}$ such that $0 \leq m < n - 2 * l - 2$. However, then $p_{\alpha(2)}$ must cross back and over again to give the flashlight to $p_{\alpha(1)}$ (because $p_{\alpha(1)}$ must have given the flashlight to p_b and $p_{\alpha(n-2*l-m)}$). If we have $p_{\alpha(1)}$ return to walk with both p_b and $p_{\alpha(n-2*l-m)}$ individually, it takes time $t_{\alpha(1)} + t_b + t_{\alpha(1)} + t_{\alpha(n-2*l-m)}$ time units. The optimal solution's method takes $t_{\alpha(1)} + t_{\alpha(n-2*l-m)} + t_{\alpha(2)} + t_{\alpha(2)}$ time units, which by assumption is less than or equal to the revisal. Hence, we can revise $OPT(I)$ to create an $OPT'(I)$ that agrees with the greedy sum for one more step and is still optimal, contradicting our assumption that $OPT(I)$ agrees with greedy for the most steps. If $t_b < t_a$, t_a must be a duplicate of some greater term (that is not duplicated in greedy, meaning it is not just indicating two people with the same speed). This means someone other than $p_{\alpha(1)}$ and $p_{\alpha(2)}$ crosses back. This, however, cannot be optimal because it would incur extra time on the walk back and not save any time on its eventual return to the destination side (because $t_a > t_{\alpha(1)}, t_{\alpha(2)}$). Hence, we have a contradiction.

Lastly, we need to consider the case where $p_a = p_{\alpha(1)}$ and $p_b = p_{\alpha(2)}$. Then, there are more than $(n-l-2)$ instances where $p_{\alpha(1)}$ crosses back in the optimal solution. However, there are only $(n-l-2)$ steps where pairings containing someone other than $p_{\alpha(1)}$ or $p_{\alpha(2)}$ cross the bridge, and $p_{\alpha(1)}$ can only cross back one time for each of them. Hence, we have a contradiction. Therefore, the greedy time must be the same as the optimal time, so the greedy algorithm must be correct.

2. Give a polynomial time algorithm that takes three strings, A, B, and C, as input, and returns the longest sequence S that is a sequence of A, B, and C.

Solution: Let string A be length n_1, string B be length n_2, and string C be length n_3. The following pseudocode gives a cubic time algorithm to find the length of the longest common subsequence:

```
SubSeq(A,B,C,n1,n2,n3):
    int Sol[n1 + 1][n2 + 1][n3 + 1];
    Sol[0][0][0] = 0;
    for i from 1to n1
        Sol[i][0][0] = 0;
    for i from 1 to n2
        Sol[0][i][0] = 0;
    for i from 1 to n3
        Sol[0][0][i] = 0;

    for i from 2 to n1
        for j from 2 to n2
            for k from 2 to n3
                if A[i] = B[j] = C[k]
                    Sol[i][j][k]=Sol[i-1][j-1][k-1]+1;
```

```
            else
                Sol[i][j][k] = max{Sol[i − 1][j][k], Sol[i][j − 1][k], Sol[i][j][k − 1]};
    return Sol[n1][n2][n3];
```

We can then find the longest subsequence from this matrix by the linear algorithm described by the following pseudocode (run time $\leq n_1 + n_2 + n_3$):

```
findSubstring(Sol,A,B,C,n1,n2,n3):
    i = n1; j = n2; k = n3;
    while i ≠ 0, j ≠ 0, and k ≠ 0
        if A[i] = B[j] = C[k]
            append A[i] to the front of substring
            i − −; j − −; k − −;
        else if Sol[i][j][k] = Sol[i − 1][j][k]
            i − −;
        else if Sol[i][j][k] = Sol[i][j − 1][k]
            j − −;
        else
            k − −;
    return substring;
```

3. Give an efficient algorithm for finding the shortest common super-sequence of two strings A and B. C is a super-sequence of A iff A is a subsequence of C.

   Solution:

```
SCS(char A[m], char B[n]):
    int SCS[m+1,n+1]
    for(i = 0 to n):
        SCS[0,i] = i
        SCS[i,0]=i
    for(i = 1 to m):
        for(j = 1 to n):
            if(A[m] == B[n]):
              SCS[m,n] = SCS[m-1, n-1] +1
            else:
              SCS[m,n] = min{SCS[m-1,n], SCS[m,n-1]}+1
    return SCS[m,n] //Length of Shortest Common Super-Sequence

findSupString(SCS,A,B,m,n)
    i=m, j=n
    while i ≠ 0 and j ≠ 0:
        if A[i]=B[j]:
            append A[i] to front of superstring
            i − −; j − −;
        else if SCS[i[j] = SCS[i][j − 1] + 1:
            append B[j] to front of superstring
            j − −;
        else:
            append A[i] to front of superstring
            i − −;
    return superstring
```

4. Consider the algorithm that you developed for the previous problem.

(a) Show the table that your algorithm constructs for the inputs $A = zxyyzz$, and $B = zzyxzy$

	*	z	x	y	y	z	z
*	0	1	2	3	4	5	6
z	1	1	2	3	4	5	6
z	2	2	3	4	5	5	6
y	3	3	4	4	5	6	7
x	4	4	4	5	6	7	8
z	5	5	5	6	7	7	8
y	6	6	6	6	7	8	9

(b) Explain how to find the length of the shortest common super-sequence in your table.

The strings that were used to get the answer will have lengths m and n. Do an array query on row m and column n to get the optimal length.

(c) Explain how to compute the actual shortest common super-sequence from your table by tracing back from the table entry that gives the length of the shortest common super-sequence.

(See findSupString pseudocode above.) To explain this, first let us look at what all array searches will be comparing:

x_1	x_2
x_3	$x_{current}$

Looking at this sub-table, we can base how to build the string. The current position at the start will be at position $[n, m]$ in the array. If the current spot we are at have equal letters at the array intersection of current (so at the beginning, it will be the nth character of String A, and the mth character of B), then we go to the diagonal position x_1 and add the letter of current to our string. We then repeat the process, making x_1 our new current, and the row above it x_2 and the column to the left x_3. If they are not the same letters, and $x_2 < x_3$, choose x_2 as the new current, and add the letter that was in the row to the string. If it is none of the above cases, then make x_3 your new current and add the current column's letter to the beginning of your string.

The trace would look something like this:

	*	z	x	y	y	z	z
*	Finish						
z	add z						
z		add z					
y		add y					
x			add x				
z			add z	add y			
y					add y	add z	add z

The same table with the original values:

	*	z	x	y	y	z	z
*	0	1	2	3	4	5	6
z	1	1	2	3	4	5	6
z	2	2	3	4	5	5	6
y	3	3	4	4	5	6	7
x	4	4	4	5	6	7	8
z	5	5	5	6	7	7	8
y	6	6	6	6	7	8	9

An interesting little side note: you can derive the largest common substring of this problem by doing the same procedure as above, except only adding the letters when A[n]==B[m]