CS1510 Greedy 14, Dynamic Programming 5 & 6

Rebecca Negley, Sean Myers

September 21, 2011

5. The input to this problem is a pair of strings $A = a_1...a_m$ and $B = b_1...b_n$. The goal is to convert A into B as cheaply as possible. The rules are as follows. For a cost of 3 you can delete any letter. For a cost of 4 you can insert a letter in any position. For a cost of 5 you can replace any letter by any other letter. For example, you can convert $A = abcabc$ to $B = abacab$ via the following sequence: *abcabc* at a cost of 5 can be converted to *abaabc*, which at cost of 4 can be converted to *ababc*, which at cost of 3 can be converted to *abac*, which at cost of 4 can be converted to *abacb*, which at cost of 4 can be converted to *abacab*. Thus the total cost for this conversion would be 19. This is almost surely not the cheapest possible conversion.

   Solution: In order to write a dynamic algorithm, we will first write a recursive algorithm that can solve the problem. We will call this the *convert* algorithm.

   We first need a base case. The base case for this problem would be when the length of at least one string is 0. If $n = 0$, that means that in order to turn the string A into B, we would need to delete all the letters $A[1], \ldots, A[m]$. The cost of doing this would be $m * 3$, so:
   if(n == 0): return m*3
   Now if $m = 0$, then we would need to insert $B[1], \ldots B[n]$, and each insertion costs 4, so:
   if(m==0): return n*4;

   Now we work on the non-base case recursive calls, considering the last character in each string. There are generally 2 cases that can occur when we compare the last characters in the string; either they are equal or they not. If they are equal, we need to add no further cost, and we continue to check the inner substrings of both. That would look like:
   if(A[m] == B[n]): return convert(m-1, n-1);

   If the final characters of the two strings do not match, there are three possible actions that could produce the optimal solution: delete a letter, insert a letter, or replace a letter. We cannot initially tell which of these is optimal, so we take the minimum cost of thre three. If we choose to delete a letter off of A, then the recursive call would have $m - 1$ characters and B would still have n characters. If we choose to change a letter, the final letters would then be equal, and so we can move to both $n - 1$ and $m - 1$. If we insert a letter into A, we know this additional letter will be equal to $B[n]$, so we can move to m and $n - 1$. The final equation would then look like this:
   else: return min(convert(m-1,n) + 3, convert(m-1,n-1) + 5, convert(m, n-1)+ 4)

   The +3, +4, and +5 are the costs of doing that operation. This solves the problem, and the overall form would look like:

```
convert(m,n):
        if(n==0):
           return m*3
```

```
elif(m==0):
  return n*4
elif(A[m]==B[n]):
  return convert(m-1,n-1)
else:
  return min(convert(m-1,n) + 3, convert(m-1,n-1) + 5, convert(m, n-1)+ 4)
```

Of course, this is exponential in nature. If you have a 100 letters for each string, each recursive call branches three times at most. The depth would be at most n+m, and at least would be m or n on each branch. If they are equal strings, you would have 100 recursive calls, and if they are completely different strings, then you would have a recursive call tree of at most 3^{200} calls and at least 3^{100}. We know that there are 10,000 recursive calls that could happen, so, suing the pidgeon-hole principal, a lot of redundant calls must be made in the worst case. Hence, we will rewrite the algorithm as a dynamic programming algorithm. We use the base cases as the initialization of an $n \times m$ table. We put the main part of the recursive calls into a loop (in this case a double for loop to fill in the table) and use table lookup calls instead of recursive calls. The bottom right spot, (n,m), in the table, is the cost we are looking for.

```
convert(m,n):
    for(i=0 to n):
        convert[0, i] = i*3
    for(i = 0 to m):
        convert[i, 0] = i*4
    for(i = 1 to m)
        for(j = 1 to n)
            if(A[i] == B[j]): convert[i,j] = convert[i-1,j-1]
            else:
                convert[i,j] = min(convert[i-1,j] + 3, convert[i-1,j-1] + 5, convert[i, j-1]+ 4)
```

If you wish to find the exact replacements, you can traverse the table backwards, and at each junction figure out the operation done.

6. Find the optimal binary search tree for keys $K_1 < K_2 < K_3 < K_4 < K_5$ where the access probabilities/weights are .5, .05, .1, .2, .25 respectively. Using the algorithm discussed in class and in the notes construct one table showing the optimal expected access time for all subtrees considered in the algorithm, and another showing the roots of the optimal subtrees computed in the other table. Show how to use the table of roots to recompute the tree.

Solution: For simplicities sake, I will leave out the description of how to build the table. I just referenced the algorithm used in the notes and applied a mechanical process to do so.

Table 1 of costs:

	0	1	2	3	4	5
1	0	.5	.6	.85	1.4	2.15
2	0	0	.05	.2	.55	1.05
3	0	0	0	.1	.4	.9
4	0	0	0	0	.2	.65
5	0	0	0	0	0	.25

Table of root nodes:

	0	1	2	3	4	5
1	0	1	1	1	1	1
2	0	0	2	3	4	4
3	0	0	0	3	4	4
4	0	0	0	0	4	5
5	0	0	0	0	0	5

We can create a tree from this table by backwards traversing the root nodes, in a recursive fashion. So if we start at the top-right, or the (left,right)-most node, we take that and user our algorithm tree(left,right) and then split it into: Left = tree(left, current-1) and Right = tree(current+1, right). The base cases would be if the right integer is less than the left integer, which is a null node. This would happen if say, you choose the first key as the root node. The left would be Tree(1, 1-1), which means we return null if this happens. Another base case is if right equals left. At this point, we do a simple table look up at root[left, right], and create a node from that key. Otherwise, we have the normal case of making a root node at that location, and filling in the left and right sub-trees. The final pseudocode would look something like:

```
tree(int left, int right):
    if(left > right): return null
    else if(left == right): return new node(root[left,right])
    else:
        root = new node(root[left,right])
        left = tree(left, root[left,right]-1)
        right = tree(root[left,right]+1, right)
        return root
```