

CMPS 102 Solutions to Homework 7

Kuzmin, Cormen, Brown, lbrown@soe.ucsc.edu

November 17, 2005

Problem 1. 15.4-1 p.355 LCS

Determine an LCS of $x = (1, 0, 0, 1, 0, 1, 0, 1)$ and $y = (0, 1, 0, 1, 1, 0, 1, 1, 0)$. We use $\text{LCS-LENGTH}(x,y)$ (p. 353) to build tables c and b .

Table c :

		0	1	2	3	4	5	6	7	8
			1	0	0	1	0	1	0	1
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
2	1	0	1	1	1	2	2	2	2	2
3	0	0	1	2	2	2	3	3	3	3
4	1	0	1	2	2	3	3	4	4	4
5	1	0	1	2	2	3	3	4	4	5
6	0	0	1	2	3	3	4	4	5	5
7	1	0	1	2	3	4	4	5	5	6
8	1	0	1	2	3	4	4	5	5	6
9	0	0	1	2	3	4	5	5	6	6

Table b :

		1	2	3	4	5	6	7	8
		1	0	0	1	0	1	0	1
1	0	↑	↘	↘	←	↘	←	↘	←
2	1	↘	↑	↑	↘	←	↘	←	↘
3	0	↑	↘	↘	↑	↘	←	↘	←
4	1	↘	↑	↑	↘	←	↘	←	↘
5	1	↘	↑	↑	↘	↑	↘	↑	↘
6	0	↑	↘	↘	↑	↘	↑	↘	↑
7	1	↘	↑	↑	↘	↑	↘	↑	↘
8	1	↘	↑	↑	↘	↑	↘	↑	↘
9	0	↑	↘	↘	↑	↘	↑	↘	↑

We use $\text{PRINT-LCS}(b, x, 9, 8)$ (p. 355) to construct an LCS of x, y from b . We follow the path of arrows from $b[9, 8]$; whenever we encounter a d in entry $b[i, j]$ it implies that $x_i = y_j$ is an element of the LCS. We get that an LCS of x and y is $(0, 0, 1, 1, 0, 1)$.

Problem 2. 16.2-4 p.384 Filling Up Greedily

The optimal strategy is the obvious greedy one. Starting with a full tank of gas, Professor Midas should go to the farthest gas station he can get to within n miles of Newark. Fill up there. Then go to the farthest gas station he can get to within n miles of where he filled up, and fill up there, and so on.

Looked at another way, at each gas station, Professor Midas should check whether he can make it to the next gas station without stopping at this one. If he can, skip this one. If he cannot, then fill up. Professor

Midas doesn't need to know how much gas he has or how far the next station is to implement this approach, since at each fillup, he can determine which is the next station at which he'll need to stop.

This problem has optimal substructure. Suppose there are m possible gas stations. Consider an optimal solution with s stations whose first stop is at the k th gas station. Then the rest of the optimal solution must be an optimal solution to the subproblem of the remaining $m - k$ stations. Otherwise, if there were a better solution to the subproblem, i.e., one with fewer than $s - 1$ stops, we could use it to come up with a solution with fewer than s stops for the full problem, contradicting our supposition of optimality.

This problem also has the greedy-choice property. Suppose there are k gas stations beyond the start that are within n miles of the start. The greedy solution chooses the k th station as its first stop. No station beyond the k th works as a first stop, since Professor Midas runs out of gas first. If a solution chose a station $j < k$ as its first stop, then Professor Midas could choose the k th station instead, having at least as much gas when he leaves the k th station as if he'd chosen the j th station. Therefore, he would get at least as far without filling up again if he had chosen the k th station.

If there are m gas stations on the map, Midas needs to inspect each one just once. The running time is $O(m)$.

Problem 3. 16.3-6 p.392 Huffman's alg for ternary codewords

We generalize Huffman's algorithm on page 388 to work on ternary codewords. This algorithm is implemented with a priority queue (i.e. a heap) so that the many calls to extract the minimum node are most efficient.

TERN-HUFF(C)

0. **if** $|C|$ is even **then** add a new character γ to C with frequency 0
1. $n \leftarrow |C|$
2. $Q \leftarrow C$
3. **for** $i \leftarrow 1$ **to** $\lfloor \frac{n}{2} \rfloor$
4. **do** allocate new node z
5. $left[z] \leftarrow u \leftarrow \text{EXTRACT-MIN}(Q)$
6. $mid[z] \leftarrow v \leftarrow \text{EXTRACT-MIN}(Q)$
7. $right[z] \leftarrow w \leftarrow \text{EXTRACT-MIN}(Q)$
8. $f[z] \leftarrow f[u] + f[v] + f[w]$
9. INSERT(Q, z)
10. **return** EXTRACT-MIN(Q)

To encode the n characters with codewords from the alphabet 0, 1, 2, label left edges with the symbol 0, mid edges with 1, and right edges with 2. This ensures that no codeword is a prefix of another codeword.

If $|C| = n$ then after one iteration through the **for** loop there will be $n - 2$ nodes. Each time through the **for** loop we remove 3 nodes from the queue, and add 1 back in. We ensure that $|C|$ is odd so that we exit the **for** loop having a single node in Q .

When implemented with a heap, EXTRACT-MIN takes $O(\lg n)$ time and INSERT is $O(\lg n)$ as well. We make $3 \cdot \lfloor n/2 \rfloor$ calls to EXTRACT-MIN, and $\lfloor n/2 \rfloor$ calls to INSERT. Therefore TERN-HUFF has a running time of $O(n \lg n)$.

Problem 4. 15.5-2 p.363 Opt bin search tree

We determine the cost and structure of an optimal binary search tree for a set of $n = 7$ keys with the following probabilities:

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

We do this by running OPTIMAL-BST($p, q, 7$) given on page 361 of Cormen. This algorithm returns two matrices, e and $root$. The matrix e gives the expected search costs for the optimal binary search tree of all subproblems. And $root$ allows us to construct the optimal binary search tree.

We get that matrix e is:

	0	1	2	3	4	5	6	7	8
7	3.12	2.61	2.13	1.55	1.20	.78	.34	.05	
6	2.44	1.96	1.48	1.01	.72	.32	.05		
5	1.83	1.41	1.04	.57	.30	.05			
4	1.34	.93	.57	.24	.05				
3	1.02	.68	.32	.06					
2	.62	.30	.06						
1	.28	.06							
0	.06								

And $root$ is:

	1	2	3	4	5	6	7
7	5	5	5	6	6	7	7
6	3	3	4	4	6	6	
5	3	3	4	5	5		
4	2	3	3	4			
3	2	3	3				
2	2	2					
1	1						

Then the optimal binary search tree has the following structure:

5 is the root	d_3 is the left child of 4
2 is the left child of 5	d_4 is the right child of 4
1 is the left child of 2	7 is the right child of 5
d_0 is the left child of 1	6 is the left child of 7
d_1 is the right child of 1	d_5 is the left child of 6
3 is the right child of 2	d_6 is the right child of 6
d_2 is the left child of 3	d_7 is the right child of 7
4 is the right child of 3	

Now we calculate the expected search cost node by node. This step is not necessary, but good check to do to verify that your calculations are correct:

node	depth	probability	contribution
1	2	.04	.08
2	1	.06	.06
3	2	.08	.16
4	3	.02	.06
5	0	.10	0
6	2	.12	.24
7	1	.14	.14
d_0	3	.06	.18
d_1	3	.06	.18
d_2	3	.06	.18
d_3	4	.06	.24
d_4	4	.05	.20
d_5	3	.05	.15
d_6	3	.05	.15
d_7	2	.05	.10
Total			2.12

Problem 5. Longest Common Substring

1. Non-dynamic programming solution

We can run along the two strings, incrementing a counter whenever the 2 characters are the same in both strings, resetting the counter back to zero if they are not. The problem is that we don't know exactly where in the strings the max substring starts. So we offset one of the strings in relation to the other and try all such offsets (with one string starting both on the left and on the right in relation to the other substring). On one of these offsets the maximum substring is bound to be lined up against its counterpart and we will detect it on our run through the 2 strings. An example of several offsets looks like this (<, > signs mark the beginning and the end of the detected common substrings for that offset):

```

photograph ph<o>t<o>graph>  photograph      pho<to>graph
tomography  t<o>m<o>graph>y  tomography      <to>mography

ph<o>tograph  photograph  photograph  photograph
tom<o>graphy  tomography tomography tomography

```

Pseudocode for this algorithm:

```
//returns maximum common substring length
```

```

int max_substring(string x, string y)
{
    int n = length(x);
    int m = length(y);
    int i, j;
    int max, cur_max; //variables for
    int cnt;

    max = 0;
    for (i = -n+1; i<n; i++)
    {
        cur_max = 0; cnt = 0;
        for (j = max(1, 1+i); j<= min(m, n+i); j++)
        {
            if (y[j] == x[j - i]) cnt++ else cnt = 0;
            if (cnt > cur_max) cur_max = cnt;
        }
        if (cur_max > max) max = cur_max;
    }
    return max;
}

```

The external loop goes through $2n - 1$ iterations (all possible offsets on both sides, including zero). The internal loop takes varying times, but its not bigger then m iterations (it iterates over the intersection of the first, shifted string and the second one). Therefore the execution time of this algorithm is in $O(nm)$.

2. Dynamic programming solution

Our subproblems will be the lengths of the longest substrings of all possible prefixes of our original strings. Thus, if $X = x_1x_2 \dots x_n$ and $Y = y_1y_2 \dots y_m$ then $L[i, j]$ will denote the length of the longest common substring that ends at x_i and y_j . Thus we consider only those substrings that suffixes of some $x_1x_2 \dots x_i$ and $y_1y_2 \dots y_j$ (which are prefixes of the original strings). Since the longest common substring has to end somewhere, there will be a i, j pair that corresponds to it. But since we don't know i, j beforehand the answer will be the maximum entry in the table (in other dynamic programming problems we had the answer to our optimization problem always in a specific cell). How is the table filled out? If some pair of i and j we have a symbol match - $x_i = y_j$, then we are extending the previously longest common substring that ended at $i - 1, j - 1$ by one symbol. If there is no match - then there is no common substring ending in this place. The recurrence looks like this:

$$L[i, j] = \begin{cases} L[i - 1, j - 1] + 1 & \text{if } x[i] = y[j] \text{ and } i - 1 > 0, j - 1 > 0 \\ 1 & \text{if } x[i] = y[j] \text{ and } i - 1 = 0 \text{ or } j - 1 = 0 \\ 0 & \text{if } x[i] \neq y[j] \end{cases}$$

We depend on the subproblem in the cell one row up and one column to the left. Filling in the table by row, left-to-right starting at the top will ensure that we will have all the necessary cells we need to fill in the current one.

Note that it might feel more natural to choose different subproblems for this problem - namely the length of longest common substring for all prefixes of the 2 strings (instead of a longest substring that ends in a specific place). But then we would be somewhat stuck on the recursive formulation. Knowing that the 2 endpoints match would no longer allow us to conclude that the length of the longest substring increases by one, since it would depend on where the previous longest substring ended. Thus the trick for this problem was to impose additional restriction on the subproblems and then see that we can still extract the solution to the original problem from the restricted subproblems.

An example subproblem matrix looks like this:

```

      1 2 3 4 5 6 7 8 9 10
      p h o t o g r a p h
1    t 0 0 0 1 0 0 0 0 0 0
2    o 0 0 1 0 2 0 0 0 0
3    m 0 0 0 0 0 0 0 0 0 0
4    o 0 0 1 0 1 0 0 0 0
5    g 0 0 0 0 0 2 0 0 0 0
6    r 0 0 0 0 0 0 3 0 0 0
7    a 0 0 0 0 0 0 0 4 0 0
8    p 1 0 0 0 0 0 0 0 5 0
9    h 0 2 0 0 0 0 0 0 0 6
10   y 0 0 0 0 0 0 0 0 0 0

```

Extracting the actual longest substring means remembering at which i or j the max value in the matrix occurs and then taking $x_{i-max+1} \dots x_i$ or $y_{j-max+1} \dots y_j$.

The pseudocode for the dynamic programming version of the solution might look like this:

```

int max_substring(string x, string y)
{
    int n = length(x);
    int m = length(y);
    int L[n][m];
    int i, j;
    int max = 0;

    for (i=1; i<=n; i++)
        for (j=1; j<=m; j++)
        {
            if (x[i] == y[j])
                if ((i>1) && (j>1)) L[i][j] = L[i-1][j-1] + 1; else L[i][j] = 0;
            else L[i][j] = 0;
            if (L[i][j] > max) max = L[i][j];
        }

    return max;
}

```

Note that there actually is substantial similarity between the first solution and the dynamic programming solution. From the recurrence we see that $L[i, j]$ possibly depends only on $L[i-1, j-1]$. Thus if we follow any diagonal through the matrix we will also have no problems. And this is basically what the first solution does. The inner loop can be seen as filling in some diagonal of the matrix (the cell values would be the values of `cnt` on successive iterations of the loop). In terms of the textbook, the first solution corresponds to following a different reverse topological order on the subproblem graph. Also this order is more space-efficient since we don't have to keep the entire matrix around.

Extra Credit. 15.7 p. 369 Job Scheduling

There are n jobs a_1, a_2, \dots, a_n and each job a_j has a processing time t_j , a profit p_j and a deadline d_j . If job a_j finishes after the deadline, then the profit is 0. We want to schedule the jobs to maximize the profit.

First order the jobs by increasing deadline, so that a_1, a_2, \dots, a_n have increasing deadlines $d_1 \leq d_2 \leq \dots \leq d_n$. Any optimal schedule can be converted to one which has this property and has the same total profit.

Our subproblem is to maximize the profit by scheduling jobs a_1, a_2, \dots, a_i with a final deadline of d_i . Let $Profit[i, j]$ be the maximum profit for doing a subset of jobs $\{a_1, a_2, \dots, a_i\}$ by time j (the rows of $Profit$

are jobs and the columns represent time, from 1 to j).

The recursive definition of an optimal subproblem is:

$$Profit[i, j] = \begin{cases} 0 & \text{if } i \text{ or } j = 0, \\ Profit[i - 1, j] & \text{if } t_i > j \text{ or } d_i < j, \\ \max(Profit[i - 1, j], Profit[i - 1, j - t_i] + p_i) & \text{otherwise.} \end{cases}$$

Let $D = d_n$ the latest deadline. Then we will build an $n \times D$ profit matrix P . We need a way to construct the best schedule, so we build a second matrix S :

$$S[i, j] = \begin{cases} 0 & \text{if } i \text{ or } j = 0, \\ i & \text{if } i, j > 0, t_i < j, \text{ and } Profit[i - 1, j - t_i] + p_i \text{ is max,} \\ i - 1 & \text{if } i, j > 0, t_i < j, \text{ and } Profit[i - 1, j] \text{ is max.} \end{cases}$$

We can fill-in these matrices top to bottom, left to right.

Example: Let A be a set of 7 jobs.

a_i	p_i	d_i	t_i
1	1	1	1
2	2	3	2
3	3	4	3
4	4	5	2
5	2	6	2
6	12	6	5
7	2	8	3

The final deadline is $D = 8$. So the *profit* table is 8×9 :

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0
2	0	1	2	3	0	0	0	0	0
3	0	1	2	3	4	0	0	0	0
4	0	1	4	5	6	7	0	0	0
5	0	1	4	5	6	7	8	0	0
6	0	1	4	5	6	12	13	0	0
7	0	1	4	5	6	12	13	8	14

Table S :

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0
2	0	1	2	2	0	0	0	0	0
3	0	1	2	2	3	0	0	0	0
4	0	1	4	4	4	4	0	0	0
5	0	1	4	4	4	4	5	0	0
6	0	1	4	4	4	6	6	0	0
7	0	1	4	4	4	6	6	7	7

From the *profit* table we know that 14 is the greatest pay we can get, and from the last row of table S we construct the job schedule that gives the optimum pay. We need to use the values t_1, \dots, t_7 to construct the optimal schedule. We start at $S[7][8]$ and determine that the last job is job a_7 . Subtract $8 - t_7 = 5$ and determine from $S[7][5]$ that the previous job is a_6 . Subtract $5 - t_6 = 0$. So the best schedule is a_6, a_7 .