Homework #7: Programming Project
Due Date: Wednesday, 9 March 2016

1 Instructions

This project has both a theory component and a programming component. Both parts are due on Wednesday, March 9, no later than 5:00 pm. While you can complete the programming component independently of the theory component, it is useful to complete (or at least read) the theory component first to get a better understanding of the algorithms you are asked to implement.

As with the homework assignments, you may collaborate in groups of up to three people. Unlike the homework assignments, a single writeup of the theory component and a single copy of the code for the programming component is sufficient per group. You may refer to the textbook, course lectures, and supplemental material, in obtaining a solution — but you may not reference outside resources in general.

2 Motivation

Recall the *Longest Common Subsequence* problem (*LCS*), discussed in lecture and described in detail in Section 15.4 of the textbook. As the textbook mentions, one application for this problem is to determine the similarity of DNA strands. In this project, we will explore a variation of this problem that applies to the case where we want to match plasmids instead of regular DNA strands. Plasmids (http://en.wikipedia.org/wiki/Plasmid) are circular strands of DNA, and any means of comparing these circular strands requires that we consider all the possible alignments of the two circles. Your goal is to use ideas we have studied throughout the quarter to design an algorithm that solves this problem efficiently, which you will then implement.

It may interest some of you to learn that while the vast majority of the algorithms covered in this course thus far were developed in the 60s and 70s, the algorithm you will design was first published (in a slightly more general form) in 1990, and has seen improvements and adaptations well into this century. The building blocks you have learned in this course are still very much applicable to algorithms research today.

3 Theory (30 points)

Note that we can model a cyclic string by taking a regular string, cutting it into two pieces, and pasting the pieces back together in the other order. This models all the possible ways we could flatten a cyclic string into a regular one. Given a string A of length n and an integer

k, $0 \leq k < n$, we can define $cut(A,k)$ = $substring(A,k,n) + substring(A,0,k)$. For example, $cut(\text{“}\texttt{abcd}\text{”},2) = \text{“}\texttt{cd}\text{”} + \text{“}\texttt{ab}\text{”} = \text{“}\texttt{cdab}\text{”}$, while $cut(\text{“}\texttt{abcd}\text{”},0) = \text{“}\texttt{abcd}\text{”} + \text{“”} = \text{“}\texttt{abcd}\text{”}$. We can extend this definition to any integer k by defining $cut(A,k) = cut(A,k \mod n)$ when $k < 0$ or $k \geq n$.

Let us define the following problem:

Let A and B be strings of length m and n respectively, where $m \leq n$. A *cyclic longest common subsequence* of A and B $CLCS(A,B)$ is defined to be a longest $LCS(cut(A,i),cut(B,j))$ over all possible choices of i and j.

The statement of *CLCS* invokes *LCS*, so let's see how those two problems compare. The first thing we can establish is that *CLCS* is no easier than *LCS*.

(a) (5 points) Show that if *CLCS* could be solved in $T(m,n) = o(mn)$ time, then *LCS* could also be solved in $O(T(m,n))$ time.
Hint: Given two input strings to *LCS*, can we modify those strings (possibly augmenting the alphabet) and feed them to *CLCS* such that we can extract an *LCS* solution from the output of *CLCS*?

On the other hand, we can also show that *CLCS* is not much harder than *LCS*. We could solve *CLCS* by solving mn instances of *LCS*, corresponding to each possible pair of cuts of the strings A and B. This implies that we have a solution to CLCS that runs in $O(m^2n^2)$ time. In fact, with a little thought, we can do better by showing that we don't need to test every possible pair of starting points. In fact, it suffices to find the longest $LCS(cut(A,k),cut(B,0))$ over all possible choises of k, which we can do in $O(m^2n)$ time.

(b) (5 points) Show that the proposed algorithm still returns a correct answer.
Hint: Show a relationship between $LCS(cut(A,i),cut(B,j))$ and $LCS(cut(A,k),cut(B,0))$ for an appropriate choice of k.
Hint: As tempting at it is, $k = i - j$ does NOT work. Can you think of a counterexample? In fact, NO equation of the form $k = f(i,j,m,n)$ will work, because k can be affected by the actual characters in A and B. But that's not a problem; you just need to show that such a k always has to exist, not necessarily identify what it is.
Note: You may find it easier to solve this problem after you have completed later parts of this project, part (c) in particular. You ARE allowed to use the result of part (c) in your solution to this problem.

With these two observations we can see that our target runtime for *CLCS* lies somewhere between $\Omega(mn)$ and $O(m^2n)$. Now let's see if we can tighten those bounds. To do so, let's take a short detour to look at regular *LCS*. Recall that we solved *LCS* in $O(mn)$ time with the use of a two-dimensional DP table. Normally we look at this table as a two-dimensional array; however, we can also look at this table as a directed grid graph G' as follows (Figure 1):

- For each entry in the table we have a single node.

		A	B	A	B	B
	0	0	0	0	0	0
B	0	0	1	1	1	1
B	0	0	1	1	2	2
A	0	1	1	2	2	2
A	0	1	1	2	2	2

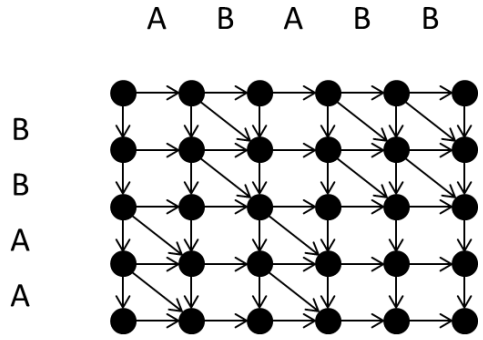

Figure 1: Visualizing the DP table as a grid graph.

- For each recurrence relation in the table we have a directed edge that points in the direction that new values are produced in the table. Note that all edges point down and/or to the right, so the graph is acyclic. We see that all of the horizontal and vertical edges are present in the graph, and in addition we have diagonal edges wherever the corresponding characters in the two strings match.

(c) (5 points) Show a correspondence between paths from $(0,0)$ to (m,n) in G' and common subsequence between strings A and B. Use this to show how to, given a shortest path from $(0,0)$ to (m,n) in G', recover a longest common subsequence between A and B (where every edge has length 1).

This means that we can solve *LCS* by solving a shortest path problem on a directed acyclic graph with unit edge lengths. (Note that the solution to this graph problem (BFS) is neither faster nor slower, at least asymptotically, than the DP solution.) According to the solution to part (b), we can solve *CLCS* by solving m shortest-path problems, one for each cut of A. However, we notice that the graphs associated with each cut have a lot in common; in fact, we can build one big graph G (and a corresponding DP table) by concatenating $cut(A,0)$ with itself and following the construction above (Figure 2). Then the shortest paths we want to find are the ones from $(0,0)$ to (m,n), from $(1,0)$ to $(m+1,n)$, ..., and from $(m-1,0)$ to $(2m-1,n)$. Now, we haven't done anything that will save us computation time; it will still cost us $O(m^2n)$ time to find these m shortest paths. However, the hope is that since these paths all live on the same graph, we can reuse some work between shortest path computations.

(d) (5 points) Fix i, j such that $i < j$. Let p_i be a shortest path from $(i,0)$ to $(m+i,n)$. We can use this path to define two subgraphs G_{U_i} and G_{L_i} whose nodes consist of the nodes above and below p_i respectively (Figure 3). Let x be any node in G_{U_i}. Show that there cannot be a path from $(j,0)$ to $(m+j,n)$ through x that is shorter than the shortest path from $(j,0)$ to $(m+j,n)$ in $G_{L_i} \cup p_i$.

By similar logic, if we know p_j, we can show that for any x in G_{L_j}, there cannot be a path from $(i,0)$ to $(m+i,n)$ through x that is shorter than the shortest path from $(i,0)$ to $(m+i,n)$

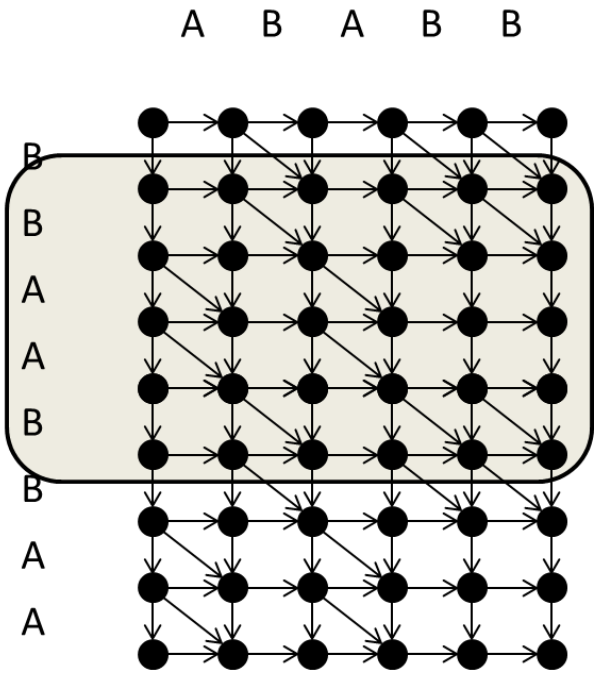

Figure 2: Constructing the graph G.

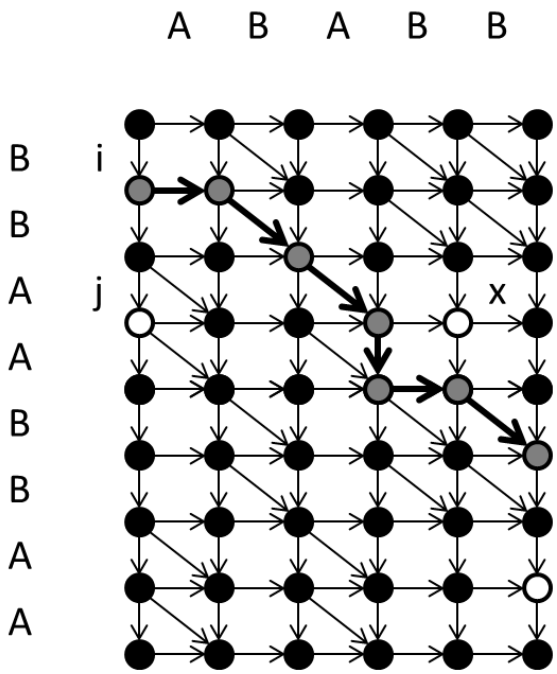

Figure 3: The setup described in part (d).

in $G_{U_j} \cup p_j$. What this means is that once we find a shortest path starting at some point i, we can use that path to limit our search for the shortest path starting at subsequent points j. More specifically, we observe that for any i, j, k such that $i < j < k$, if we know p_i and p_k, we can use them as upper and lower bounds respectively on the section of the graph we need to explore when looking for p_j (Figure 4. This suggests a divide-and-conquer strategy where we recursively split the graph to give us progressively tighter bounds on the sections of the table we need to compute for the remaining paths.

We propose the following algorithm: Allocate storage for the paths p_i. Compute p_0 using the whole table. This gives us p_m by shifting the path down by m units. Now we define the following procedure FINDSHORTESTPATHS(A, B, p, l, u) for computing p_i for all $i, l < i < u$:

```
FINDSHORTESTPATHS(A, B, p, l, u)
1   if u − l ≤ 1 return
2   mid ← (l + u)/2
3   p[mid] ← SINGLESHORTESTPATH(A, B, mid, p[l], p[u])
4   FINDSHORTESTPATHS(A, B, p, l, mid)
5   FINDSHORTESTPATHS(A, B, p, mid, u)
```

Here, SINGLESHORTESTPATH(A, B, m, p_l, p_u) computes p_m by running the DP on the table bounded by p_l and p_u. Finally, to solve the full problem, we call FINDSHORTESTPATHS$(A, B, p, 0, m)$, then return the common subsequence associated with the shortest p_i.

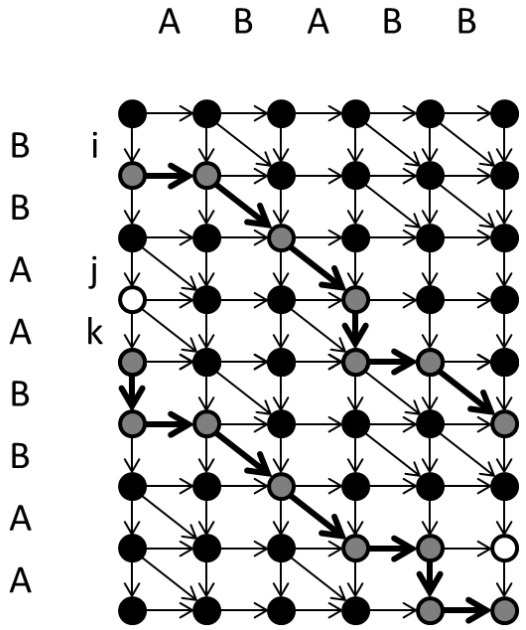

Figure 4: Bounding paths in the graph.

(e) (10 points) Show that this algorithm solves *CLCS* in $O(mn \lg m)$ time. You may assume m is a power of 2 in your analysis.
Hint: It isn't actually possible to write a useful recurrence for the runtime here (why?). Instead, ask yourself, how much total work is done at each depth of recursion? Be careful when computing the amount of work done at the boundaries.

4 Implementation (70 points)

Now that you have designed an algorithm for *CLCS*, it is time to implement it. As a warmup, you will first implement the slower version you designed in part (b). We have provided starter code for you in the `/usr/class/cs161/project` directory on AFS, which you should make a copy of for yourself. In there are the following files:

- `LCS.py`: This is an implementation of *LCS*. Feel free to make copies of this file to use as a starting point for the two files for the project.
- `sample.in`: This is a sample input file. Each line contains two strings A and B, separated by a space.
- `sample.out`: This is the expected output of *CLCS* being run on the sample input file. The output format is as follows: Each line contains a single integer equal to the length of the cyclic longest common subsequence (same number of lines as sample.in).

- `sample-lcs.out`: This is the expected output of *LCS* being run on the sample input file.

- `runsample.sh`: This is a convenience script provided to run your program on the sample input, and to check its runtime and correctness. Pass in the name of the source file as the sole argument.

- `sample2.in`, `sample2.out`, `runsample2.sh`: These are another set of sample inputs for testing your code, and should be used in the same way as the original set of sample data. This set includes a target time: Your fast solution should run within 10 seconds. (Don't run your slow solution on this one; it will take several minutes.) The script reports three time values: real, user, and sys; the relevant value is the *sum* of the user and sys times.

- `judge.py`: The provided convenience scripts rely on this file. Do not modify, remove, or relocate it.

For convenience of coding, we will place some additional constraints on the input:
All strings will consist solely of uppercase letters from the English alphabet.
All strings will be of length at most 2000 (you may use this fact to preallocate your arrays).

We also will only require that you output the length of *CLCS*, not the subsequence itself.

Your program should read input from standard input, and produce output to standard output. Note that you should NOT read directly from or write directly to files. The included implementation of *LCS* follows all of these details; as a result, by starting from that file you can focus on the logic and circumvent most of the technical details.

When you are ready to submit your solutions, add comments at the top of your files containing the names, SUID numbers, and SUNetIDs of the people in your group. Then copy your code onto a cluster machine such as the myths, and run the following script in the directory that contains your code: `/usr/class/cs161/scripts/submit.sh`. You may submit as many times as you like; we will only grade the latest version submitted. Please verify that your submission passes the sample input as determined by `runsample.sh`; that is a good way to make sure you haven't left any extraneous debugging output in your code. Also please verify that your two source files are copied to the `/usr/class/cs161/submissions/SUNETID` directory after you run the submission script. The best way to do so is to run
`./runsample.sh /usr/class/cs161/submissions/SUNETID/CLCSSlow.py`,
`./runsample.sh /usr/class/cs161/submissions/SUNETID/CLCSFast.py`
and
`./runsample2.sh /usr/class/cs161/submissions/SUNETID/CLCSFast.py`
after the submission completes.

(f) (10 points) Implement your solution to *CLCS* using the method you proved correct in part (b). Name your source file `CLCSSlow.py`

(g) (60 points) Implement your solution to *CLCS* using the method you proved correct in part (e). The output to your solution to part (f) will be useful in verifying the correctness of your implementation. Name your source file `CLCSFast.py`
Hint: Spend some time thinking about how you want to represent a "path" to use as a bound. What parts of the path are actually relevant to the problem at hand? You should NOT construct an actual graph; we only introduced the graph to allow us to justify the use of these bounds on the DP table.
Hint: It can be useful to reuse the same large preallocated array for your DP subproblems within a given test case. Be careful with how you do so, however; on the one hand, accidentally using entries "left over" from a previous subproblem can be a source of mysterious errors; on the other hand, clearing/initializing the entire preallocated array just before each subproblem within a test case constitutes $\Theta(m^2n)$ work and is therefore NOT allowed. (Note that clearing/initializing all the arrays BETWEEN test cases is not only allowed but also strongly encouraged.)
Hint: As covered in Homework 4, when iterating over a two-dimensional array, it is better to iterate through the data one row at a time than one column at a time, as the former generates fewer cache misses than the latter. Keep this in mind when designing your algorithm. If you find yourself really wanting to iterate one column at a time instead, then you might want to rotate the entire table, swapping the roles of rows and columns.
Hint: A corollary of the observation above is that running BFS instead of computing the DP table is a really bad idea, because the "wavefront" generated by BFS will be roughly diagonal. This means *both* orientations of the table will result in a large number of cache misses in BFS. DO NOT DO THIS! The graph analogy is only present in the theory portion to make the proof of correctness easier, and should NOT be used anywhere in your actual code.

You will be graded based on the performance of your code on undisclosed test files. You will receive full credit if your code produces correct output within an appropriate time bound. We have given a target time for running your code on the sample input; if your code correctly solves the sample data within this target time, you should be confident that you will meet the time requirements on the grading input. Another benchmark to keep in mind is that your fast solution should solve a maximum-size case in no more than 2 seconds. Also, note that the sample data, while a useful starting point, is by no means comprehensive. Consider creating your own test cases, obtaining the answer by running your slow solution (which is much easier to verify the correctness of), and then comparing this against what you get from your fast solution.