CS380: Algorithm Design and Analysis
Review for Exam 3

Friday, April 28th, 2017
Material Covered: Lectures 14-20

Brief Summary of Material:

Lecture 14: Dynamic Programming
Summary: The main point of this lecture was to illustrate that recursive algorithms, while often comparatively simply to explain and implement, do not always yield computationally-feasible runtimes. The main issue with recursive algorithms is that they make no attempt to store solutions to subproblems, and the Fibonacci recursive computation is a good example of how this impacts runtimes. To mitigate this issue, we introduce a strategy called Dynamic Programming that memorizes intermediate results and avoids this re-computation. A general strategy for developing such an algorithm is to follow the “Four Steps for Dynamic Programming”. The rod cutting example is designed to illustrate each step of this process.

What you need to know and/or be able to do:
- Understand both the runtime and why the recursive algorithm for computing Fibonacci numbers is slow.
- What problem does Dynamic Programming attempt to address? When should we use Divide and Conquer versus Dynamic Programming?
- Understand the initial optimal revenues slide on page 6, but that the simplification on page 7 also covers all of these cases.
- Understand that the recursive Top-Down solution, even though it mimics the page 7 simplification, still has issues with recomputing values (the tree illustrates this). The point is that even a well-crafted recursive solution needs to store intermediate results to be efficient.
- Understand and be able to use both the Memoized Cut-Rod (top-down approach) and the Bottom Up approach to solving the Cut-Rod problem. Each of these approaches returns the “Initial Optimal Revenues” table on page 6. Typically, bottom-up is the preferred method because of it is no less computationally expensive but is generally easier to understand and implement.
- Understand the meaning of the $s[i]$ term in the table that the PRINT-CUT-ROD-SOLUTION algorithm produces. If given such a table, be able to explain how to cut up a rod of a given length.

Lecture 15: Longest Common Subsequence (LCS)
Summary: The goal of this section is to illustrate an application of dynamic programming to the problem of determining the Longest Common Subsequence of two strings.

What you need to know and/or be able to do:
- You should be able to explain the equation $c[i,j]$ and $b[i,j]$ in the context of sequence alignment. Specifically, be able to answer such questions as “when $x_i=y_i$, why do we set $c[i,j]=1=c[i-1,j-1]$”?
- Given two DNA sequences, you should be able to produce the table that illustrates how to generate the LCS (that is, the combined table on page 6).
**Lecture 16: Edit Distance**

**Summary:** The goal of this section is to present an algorithm for computing a numerical value (Edit Distance) that measures the extent to which two strings are similar. The assumption in this model is that all changes (insertion, deletion, substitution) required to convert string X into string Y have the same cost (1 “unit”), a simplification that we remedy with the sequence alignment algorithms in the next lecture. This is another example of the use of dynamic programming, in that these distances are built from the bottom-up.

**What you need to know and/or be able to do:**
- Be able to describe the d[i,j] expression on the Recursive Solution slide, and in particular be able to answer questions such as: If there is a match, why do we set this value do d[i-1,j-1]? If there is not a match, why do we consider the minimum of the three values given? What does each of these three values in the MIN statement mean?
- Produce an Edit Distance matrix for two strings (as on page 4).
- Know that, while the Edit Distance number is well-defined, the algorithm for actually producing the sequence of manipulations required is non-deterministic (that is, it doesn’t tell you how to resolve the case in which the MIN computation is not well-defined). As such, there may be various paths through your Edit Distance matrix from d[x.length,y.length] to d[0,0], and this is ok.

**Lecture 17: Sequence Alignment**

**Summary:** In this lecture, we quickly reviewed the computation of both the LCS and Edit Distance of two sequences and showed more clearly how the Edit Distance matrix can be used to detail the steps required to convert sequence X into sequence Y (see slide on top of page 3). The heart of this lecture, however, was a discussion of a modification of the Edit Distance algorithm known as the Needleman-Wunsch algorithm that assumes that the various changes required to convert a sequence X into a sequence Y may have DIFFERENT costs. If we assume that the cost of replacing a character with itself is 0, and that both the gap cost and the cost of replacing one character with a different character is 1, then this computation degenerates to the Edit Distance computation. We also mentioned briefly that the time and space complexity of this algorithm is O(X.length*Y.length), and in the context of DNA analysis this can be an issue. Thus, various algorithms have been developed to mitigate the space requirements of Needleman-Wunsch that use a divide and conquer approach.

**What you need to know and/or be able to do:**
- Understand the intent the Recurrence equation on the top of page 7.
- Be able to produce an alignment-cost matrix (as on the bottom of page 8 or the top of page 9), and be able to use such a table to produce the actual alignment of the two sequences.

**Lecture 18: String Matching and Searching: Brute Force, Horspool**

**Summary:** In this section to discuss the final string algorithm Horspool, which is used for determining if one string is a substring of another. The idea is to shift the pattern as far as possible in the case in which a mismatch occurs, and this is done using a pre-computed table (indexed by the characters) that computes the right most occurrence (ignoring the right-most position) in the pattern of each character in the alphabet.

**What you need to know and/or be able to do:**
- Generate a shift table for a given string and pattern and show each step of the Horspool algorithm (so, an example similar to that on the last slide in the lecture that we did in class).
**Lecture 19: Elementary Graph Algorithms**

**Summary:** We introduced the two ways to represent graphs for algorithms: Adjacency lists and Adjacency matrix. Depending on whether the graph is dense or sparse dictates (at least in part) which representation might be preferable. As we’ve seen in class, the order in which vertices populate the various adjacency lists is not prescribed but is integral in the execution of these algorithms. We also discussed both Breadth-First Search (which does gives the shortest distance from each vertex to a prescribed root node) and Depth-First Search (which also visits each vertex, but is less concerned about distance than “time”). The parenthesis notation that we discussed in class is very helpful when associating times in DFS. We also briefly discussed the classification of edges within the DFS tree, which is useful in such associated algorithms like Topological Sorting and finding the Strongest Connected Component (SCC) within a graph.

**What you need to know and/or be able to do:**
- Given a (directed or undirected) graph, be able to give both the corresponding adjacency list and adjacency matrix.
- Know the space complexity for both the adjacency lists and adjacency matrix representations of a graph. Also know for each representation the costs associated with listing all vertices adjacent to a given vertex and for determining if a given pair is an edge in the graph.
- Be able to illustrate the steps in a BFS given a particular graph (such as the example given on the bottom of page 6).
- Know the space and time complexity of the BFS algorithm.
- Be able to illustrate the steps in a DFS given a particular graph (such as the example given on the bottom of page 9). Parenthesis notation is very helpful.
- Know the space and time complexity of the DFS algorithm.
- Be able to explain how the use of a stack versus a queue in BFS and DFS is important.
- Know and be able to use the classification of edges on the last slide (Tree, Back, Forward, Cross).

**Lecture 20: Single Source Shortest Path**

**Summary:** In this lecture, we characterized the shortest path problems and the associated definitions. We learned that there are a variety of problems that fall under the guise such shortest path problems, but that the implementations all begin with the INIT-SINGLE-SOURCE(G,s) algorithm. We then discussed the idea of relaxation, which essentially tests if we can improve the current distance we have associated with a given vertex by instead traveling through a (potential) parent vertex. If so, we update this distance and reset the parent node.

The two implementations of Single Source Shortest Path that we discussed were Bellman-Ford and Dijkstra. Bellman-Ford proceeds in a haphazard way, instead sweeping through ALL of the edges of the graph in no particular order a total of |V| - 1 times. It has the advantage of being able to handle graphs in which there are edges with negative weights, and it also detects whether a negative cycle is reachable from the source vertex (why is this an issue?). Dijkstra, on the other hand, does not allow for edges with negative weights, and proceeds through the edges and vertices of a graph in a much more organized manner by using a MIN priority queue, whose objects are the vertices and whose keys are the current estimate for the distance from that vertex to the source.

**What you need to know and/or be able to do:**
- Be able to illustrate one edge-sweep in the Bellman-Ford Algorithm
- Be able to illustrate Dijkstra on a relatively simply graph (such as on the bottom of page 10).