1 Format

You will have 83 minutes to complete the exam. The exam may have true/false questions, multiple choice, example/counterexample problems, run-this-algorithm problems, and problem set style present-and-prove problems.

Whenever you are asked to give an algorithm for a problem, you are expected to analyze its correctness and running time. More efficient correct solutions receive more credit. Less efficient but correct solutions will receive partial credit unless otherwise stated. We understand and expect that proofs of correctness will be more concise on exams than they would be on homework. Be sure to read directions carefully. If a problem asks you explicitly to prove correctness, then there’s a good chance that you want to focus on that.

2 Topics Covered

2.1 Minimal Spanning Trees

- **Prim’s Algorithm**: uses a binary/fibonacci min heap, takes $O((|E| + |V|) \log |V|)$ time with binary heap and $O(|E| + |V| \log |V|)$ with Fibonacci heap.

- **Kruskal’s Algorithm**: uses union-find data structure (see next subsection). Takes $O(|E| \log |E|)$ time to sort but then takes $O(|E| \log^* (|E|))$ time for the union-find part.

2.2 Union Find

- **Main Idea**: It is a disjoint set data structure that supports Union and Find.

- **Union by Rank and Path Compression**: You are able to achieve $O(m \log^* n)$ with both Union by Rank and Path Compression.

2.3 Greedy

- **Main Idea**: At each step, make a locally optimal choice in hope of reaching the globally optimal solution.

- Remember that greedy algorithms can often seem correct, so it’s more important to prove the optimality of your algorithm.

- **Horn Formula**: Set all variables to false and greedily set ones to be true when forced to.

- **Huffman Coding**: Find the best encoding by greedily combining the two least frequently items.

- **Set Cover**: Greedy approximation algorithm with $O(\log n)$ performance ratio.
2.4 Divide and Conquer

- **Main Idea:** Divide the problem into smaller pieces, recursively solve those, and then combine them in the right way.
- **Mergesort.**
- **Integer Multiplication:** Perform 3 multiplications on $n/2$ digit numbers, and then do some additions
- **Strassen’s Algorithm:** Multiplies two $n \times n$ matrices by doing 7 multiplications on $n/2 \times n/2$ matrices.

2.5 Fast Fourier Transform

- **Main Idea:** We can multiply two polynomials of degree $n$ in $O(n \log n)$ time.
- **Integer Multiplication:** We can treat an integer as a polynomial evaluated at 10.
- **Pattern matching:** To search for the existence of a pattern $P$ of length $m$ in a string $T$ of length $n \geq m$, we can perform successive dot products using FFT. We saw in pset 5 problem 1 that we can get $O(n \log m)$ time for arbitrary sized alphabets and wildcard symbols.

2.6 Dynamic Programming

- **Main Idea:** Maintain a lookup table of correct solutions to sub-problems and build up this table in a particular order.
- **String Reconstruction.** Tries to find where the first dictionary word ends by checking all possible breaking points.
- **Edit Distance.** Tries all possibilities of INSERT, DELETE, CHANGE on letters that are different.
- **All Pairs Shortest Paths.** Uses the idea that the shortest path to a node must have come via one of the neighboring nodes.
- **Traveling Salesman.** DP can provide better exponential-time solutions to NP-hard problems.

You should treat dynamic programming problems as having three parts. Your goal is to find a function $f$ which can be computed recursively so that evaluation of $f$ on a certain input gives the answer to the stated problem.

(a) Define $f$ in words (without mention of how to compute it recursively). You should clearly state how many parameters $f$ has, what those parameters represent, what $f$ evaluated on those parameters represents, and what parameters you should feed into $f$ to get the answer to the stated problem.

(b) Give a recurrence relation showing how to compute $f$ recursively.

(c) Give the running time and space for solving the original problem using computation of $f$ via memoization or bottom-up dynamic programming. If you need to use certain data structures to make computation of $f$ faster, you should say so.

If there are multiple solutions to solve the stated dynamic programming problem, you should describe the most time-efficient one you know. If there are multiple solutions with the same asymptotic time
complexity, you should describe the implementation that gives the best asymptotic space complexity. See problem set 5 solutions for two examples of this (Floyd-Warshall and problem 2).

3 Practice Problems

Problem 1.
Each of the following statements is false. Provide an explanation or counterexample for each.

(a) Suppose we have a disjoint forest data structure where we use the path compression heuristic. Let $x$ be a node at depth $d$ in a tree, then calling FIND($x$) a total of $n$ times takes $\Theta(nd)$ time.

(b) Using Huffman encoding, the character with the largest frequency is always compressed down to 1 bit.

(c) Consider a variation on the set cover problem where each set is of size at most 3. Then the greedy set cover algorithm discussed in class is optimal. (You can choose how the greedy algorithm breaks ties)

Problem 2.
We know that a sequence of $n$ union/find operations using union by rank and path compression takes time $O(n \log^* n)$. What if all the union operations are done first? Show that a sequence of $n$ unions followed by $m$ finds on a graph with $n$ vertices takes time $O(n + m)$, assuming the unions always are of heads.
**Problem 3.**
You are given a perfect binary tree with $2^d - 1$ nodes. Suppose that each node $x$ has a weight $w_x$ and that all the weights are distinct for the nodes of the tree. A node is called a *local minimum* if it is smaller than its parent and both its children (if they exist). The root is a local minimum if it is smaller than its two children, and a leaf is a local minimum if it is smaller than its parent. Given a pointer to the root node of this binary tree, give a $O(d)$ algorithm for finding any local minimum.

**Problem 4.**
(From MIT 6.046 Exam, Spring 2015) Finding the median of a sorted array is easy: return the middle element. But what if you are given two sorted arrays $A$ and $B$, of size $m$ and $n$ respectively, and you want to find the median of all the numbers in $A$ and $B$? You may assume that $A$ and $B$ are disjoint.

(a) Give a naive algorithm running in $O(m + n)$ time.

(b) If $m = n$, give an algorithm that runs in $O(\log m)$ time.

(c) For any $m, n$, give an algorithm that runs in $O(\log \min(m, n))$ time.
Problem 5.
Given a character array representing $n$ tasks CPU need to do. Tasks can be done in any order and each task can be finished in 1 unit time. For each unit time, CPU could finish one task or just be idle. However, there is a non-negative cooling interval $k$ that means between two same tasks, there must be at least $k$ units of time that CPU are doing different tasks or just be idle. Give a greedy approach to find the least amount of time the CPU will take to finish all the given tasks.

Problem 6.
Suppose we are given a string of zeros and ones of length $n$, $S = a_0...a_{n-1}$. We are asked to find if there exists three well-spaced ones in $S$, say, $a_i = 1$, $a_j = 1$, and $a_k = 1$ so that $k - j = j - i \geq 1$. For example, the string 10100101 does not have such a pattern while the string 0101001100111 does.

(a) Give a simple $O(n^2)$ algorithm to find at least one such triple of ones in $S$.

The goal in the next two parts is to use the FFT to find an algorithm that runs in $O(n \log n)$ time to find a well spaced triple of ones.

(b) Suppose we convolve $S$ by itself, $C = C_0...C_{2n-2}$ is the sequence gotten by squaring the polynomial $a_0 + a_1x + ... + a_{n-1}x^{n-1}$. Give an interpretation of $C_k$ both in the case when $k$ is odd and even.

(c) Use your interpretation to count the number of well spaced triples in $O(n \log n)$ time. More specifically, assuming that such a convolution is given for free, your algorithm has to run in $O(n)$ time.
Problem 7.
(From MIT 6.006, Spring 2011) Given a log of wood of length \( k \), Woody the woodcutter will cut it once, in any place you choose, for the price of \( k \) dollars. Suppose you have a log of length \( L \), marked to be cut in \( n \) different locations labeled 1, 2, \ldots, \( n \). For simplicity, let indices 0 and \( n + 1 \) denote the left and right endpoints of the original log of length \( L \). Let the distance of mark \( i \) from the left end of the log be \( d_i \), and assume that \( 0 = d_0 < d_1 < d_2 < \ldots < d_n < d_{n+1} = L \).

Give an algorithm to determine the sequence of cuts to the log that will (1) cut the log at all the marked places, and (2) minimize your total payment to Woody.

Problem 8.
Consider the 0-1 knapsack problem: We have \( n \) items, each of which has sizes \( s_i > 0 \) and value \( v_i > 0 \). We have a knapsack of size \( M \) and want to maximize the sum of the values of the items inside the knapsack without the sum of the sizes of the items exceeding \( M \). Let \( V = \sum_{i=1}^{n} v_i \) be the sum of all the values of the items.

(a) Describe a solution that computes the maximum achievable value in \( O(nM) \) time.

(b) Describe a solution that computes the maximum achievable value in \( O(nV) \) time.