1 Format
You will have 75 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.

As usual, 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. For example, if a problem asks you to explicitly 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 min heap. There are $n$ INSERT operations, at most $n$ DELETEMIN operations, and at most $m$ DECREASEKEY operations. Therefore, using a binary heap, the runtime is $O((m+n) \log n) = O(m \log n)$. With a Fibonacci heap, the runtime is $O(m + n \log n)$.
   - **Kruskal’s Algorithm.** Uses the Union-Find data structure (see next subsection). Takes $O(m \log m)$ time to sort, but then takes $O(m \log^* m)$ time for the union-find part.

2.2 Union Find
   - **Main Idea.** Disjoint set data structure that supports the operations 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.
   - **Biohazard!** Remember that greedy algorithms can often seem correct, so it’s extra important to prove the correctness and optimality of your algorithm.
   - **Horn Formulae.** Set all variables to false, and greedily set variables as true when forced to.
   - **Huffman Coding.** Find the best encoding by greedily combining the two symbols of lowest frequency.
   - **Set Cover.** We have a 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, 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 to polynomials of degree $n$ in $O(n \log n)$ time.

• **Integer Multiplication.** We can treat an (base-10) integer as a polynomial evaluated at 10.

• **Successive Dot Product.** By cleverly selecting the coefficients on our polynomials (reversing the coefficients on the $j$-degree polynomial), for polynomials of degree $j$ and $k$, with $j \leq k$, we can compute the $k - j + 1$ successive dot products (shifting $j$ coefficients along the $k$ coefficients) simply by multiplying these polynomials together in $O(k \log k)$ time.

• **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 use SDP. We saw in pset 5 problem 3 that we can get $O(n \log m)$ time for arbitrary sized alphabets with 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.

• **All Pairs Shortest Paths.** Uses the idea that the shortest path to a node must have come via one of the neighboring nodes.

• **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.

• **Matrix Chain Multiplication.** We take advantage of the associativity of matrix multiplication to find the optimal parentheses to minimize the cost of multiplying a chain of matrices (of varying dimensions) together.

• **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 $f$ that can be computed recursively, so that the 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) Define $f$ in math. 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. Problems 7 and 8 will help demonstrate how dynamic programming problems should be approached.

### 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 $\text{FIND}(x)$ 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)

**Solution**

(a) Consider a chain of $d$ elements. The first $\text{FIND}(x)$ will take $O(d)$ work while every subsequent call will take $O(1)$ work. So in total, it takes $O(d + n)$ work.

(b) Consider $A, B, C, D$ each with frequency 0.25. We will connect $A, B$ into a supernode first. Then connect $C, D$ in a supernode next and then connect $A, B, C, D$ into one big supernode. Now each takes 2 bits to encode

(c) Consider elements $\{1, 2, 3, 4, 5, 6\}$ and set $\{1, 2, 3\}, \{1, 4\}, \{2, 5\}, \{3, 6\}$. Then greedily gives us 4 sets to cover it while I can just throw away $\{1, 2, 3\}$ which only needs 3 to cover.

**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.

**Solution**

Doing $n$ unions on head costs $O(n)$ times. Now doing $m$ finds will at most change $n - 1$ edges and after that every operations takes $O(1)$. So $m$ finds take $O(m + n)$ time.
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.

Solution
Let $r$ be the root of the tree. If $r < r_{\text{left}}$ and $r < r_{\text{right}}$, then $r$ is a local minimum and we are done. Otherwise, we recurse on the subtree rooted by the smaller of $r_{\text{left}}$ and $r_{\text{right}}$. We know that the weight of this child we recurse on will be smaller than the weight of $r$, so we are in the same situation as before where we check whether this child is a local minimum and the recurse on its children appropriately. This algorithm is correct because it either finds a local minimum or it reaches a leaf, in which case the leaf is smaller than its parent and that would be a local minimum. The run-time is $O(d)$ because we go down the tree at most $d$ levels.

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.

Solution
(a) Merge the two sorted arrays (which takes $O(m + n)$ time) to preserve the order of the array and find the median directly.

(b) Pick the median $m_1$ for $A$ and median $m_2$ for $B$. Note that the overall median has to be between $m_1$ and $m_2$, inclusive. This is because the median being smaller than both $m_1$ and $m_2$ would mean at least half the elements in both $A$ and $B$ are bigger than the median, which makes no sense. We can say symmetrically the median isn’t bigger than both $m_1$ and $m_2$. Therefore, if $m_1 = m_2$ we just return $m_1$. Else, we look at the bigger half of $A$ and the smaller half of $B$ if $m_1 < m_2$ and the other way if $m_1 > m_2$ and recursively compute the median of the union of these two halves. This works because we have removed an equal number of terms that we know are at least the median and at most the median. We repeat this $O(\log m)$ times.

(c) Assume WLOG that $m > n$. Then, we can remove the first $\frac{m-n}{2}$ and last $\frac{m-n}{2}$ elements since even if the elements of $B$ are extremely large or extremely small, these elements must be in the smaller half and larger half, respectively. Then, we have two arrays of size $n$, so use part b.

Problem 5.
Describe an efficient algorithm that, given a sequence $x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_n$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given
Solution
We will use a greedy algorithm. Make the first interval \([x_1, x_1 + 1]\), remove all of the points in \([x_1, x_1 + 1]\) and repeat this process on the remaining points. The algorithm takes \(O(n)\) time. To see that it is correct. Suppose that a set of intervals \(S\) is an optimal solution. We will show that there must exist some solution \(S'\) that contains the interval \(I_1\) and is also optimal. Suppose in the solution \(S\), the point \(x_1\) is covered by an interval \([p, p + 1]\) where \(p < x_1\). Since \(x_1\) is the leftmost point, there are no points in our set in \([p, x_1]\). Therefore, we can simply replace the interval \([p, p + 1]\) in \(S\) with the interval \([x_1, x_1 + 1]\) to make a new optimal solution that contains the greedy algorithms first choice. Removing the points in \([x_1, x_1 + 1]\), we can repeat the above argument to show that the greedy solution is optimal.

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 010100101011 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.

Solution
(a) Let \(x\) run from 0 to \(n - 1\) and \(y\) run from 1 to \(n/2\). Check for each \((x, y)\) whether the indices \(x, x + y, x + 2y\) are all between 0 and \(n - 1\) and that they are all 1’s. If this ever happens, we will find a triple and return YES, otherwise we return NO.

(b) Note that \(C_k = \sum_{x+y=k,0 \leq x, y \leq n-1} a_x a_y\). Then, if \(k\) is odd, this sum is even, because for every \(a_x a_y\), there is a corresponding \(a_y a_x\). For \(k\) even, there is a term \(a_{k/2} a_{k/2}\), which is 1 if \(a_{k/2} = 1\) and 0 otherwise. Therefore, \(C_k\) is odd if and only if \(a_{k/2} = 1\), assuming \(k\) is even.

(c) If we look at any coefficient \(C_{2j}\), it equals \(a_j^2\) plus the sum over all \(i \neq k, i + k = 2j\) of \(a_i a_k\). This is the same as \(a_j^2\) plus the sum over all \(i < j < k\) where \(i, j, k\) are a well spaced triple, of \(2a_i a_k\), since \(a_i a_k + a_k a_i = 2a_i a_k\). Therefore, if there are \(r\) well spaced triples of ones with middle element \(a_j\), the value of \(C_{2j}\) will be \(2r + 1\), as there will be a term \(a_j^2\) and \(r\) terms of the form \(2a_i a_k\) where \(i, j, k\) are in an arithmetic progression (i.e. \(k - j = j - i \geq 1\)). If there are no such well spaced triples going through \(a_j\) as the middle element, then either \(a_j = 0\), so \(C_{2j}\) is even, or \(a_j = 1\) and \(C_{2j} = 1\).

Therefore, to count the total number of well spaced triples of ones, we just have to look at all
elements $C_{2j}$ and check which ones are odd and at least 3. If $C_{2j} = 2r + 1$ for some $r \geq 1$, then there are $r$ triples going through $j$ as a middle element, so our answer is the sum over all $C_{2j}$ that are odd of $\frac{C_{2j} - 1}{2}$. This can easily be computed in $O(n)$ time after computing the FFT in $O(n \log n)$ 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.

Solution

Definition. Let $X[i, j]$ be the minimum cost to break the segment from cut point $i$ to $j$ into $j - i$ pieces. We are looking for $X[0, n + 1]$.

Recurrence. We search over where the optimal first cut will take place:

$$X[i, j] = \begin{cases} 
0 & j = i + 1 \\
\min_{i+1 \leq k \leq j-1} X[i, k] + X[k, j] + (d_j - d_i) & \text{otherwise}
\end{cases}$$

This recurrence is correct because we are exhaustively checking all the possible places where the first cut can be made, and then recursively finding the best way to cut the resulting two pieces.

$X[i, j]$ finds the cost of the best sequence of cuts, but if we wanted the actual cost, we could have $X[i, j]$ store the location of the first cut point to make on this segment from $i$ to $j$. In other words, $X[i, j]$ is the $k$ that we chose in the minimum above. In order to recover the entire sequence of cut points, we would find $k = X[i, j]$ and recursively compute the best sequence of cut points for the segment between $i$ and $k$ as well as the segment between $k$ and $j$, concatenating those together and appending to $[k]$.

Analysis. The runtime is $O(n^3)$ because there are $n$ possible inputs to $X$ and each one takes $O(n)$ time to compute (to find that optimal $k$). The space complexity is $O(n^2)$ because we store 1 number per $X[i, j]$.

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.
Solution

(a) Definition. Let $X[k, m]$ be the highest achievable value when we are trying to pack the first $k$ items into a knapsack with capacity $m$. We are looking for $X[n, M]$.

Recurrence. When we are calculating $X[k, m]$, we can either choose to use or not use the $k$th item. If we choose to use it, then we must subtract $s_k$ from the capacity of the knapsack, but add on $u_k$ amount of value. If we do not choose to use it, then the capacity of the knapsack does not change. In either case, we recurse on the first $k - 1$ elements with a possibly smaller knapsack.

$$X[k, m] = \begin{cases} 
0 & k = 0, m \geq 0 \\
\infty & m < 0 \\
\max\{X[k - 1, m], X[k - 1, m - s_k] + u_k\} & \text{else}
\end{cases}$$

Analysis. The runtime is $O(nM)$ because there are $n \cdot M$ possible inputs to $X$ and each one takes $O(1)$ time to compute. The space complexity seems to be $O(nM)$ at first, but we can use bottom-up dynamic programming to improve upon the space. We notice that in order to calculate $X[k, m]$, we only need the row $X[k - 1, \cdot]$. Therefore, the space can be improved to $O(M)$ by only keeping one row $X[k - 1, \cdot]$ at a time.

(b) Definition. Let $X[k, v]$ be the minimum size of a subset of the first $k$ items whose values sum up to exactly $v$. To calculate the final answer, we look at $X[n, v]$ for all $v$ from $V$ to 0 in order and find the largest value of $v$ such that $X[n, v] \leq M$.

Recurrence. We again consider whether or not we will be taking the $k$th item when computing $X[k, v]$. If we decide to take it, then we need to compute the minimum size where we use a subset of the first $k - 1$ items and try to sum to achieve the value $v - v_k$. If we don’t decide to take it, then we change $k$ to $k - 1$ and recurse.

$$X[k, v] = \begin{cases} 
0 & k = 0, v = 0 \\
\infty & k = 0, v \neq 0 \\
\min\{X[k - 1, v], X[k - 1, v - v_k] + s_k\} & \text{else}
\end{cases}$$

Analysis. The runtime is $O(nV)$ because there are $n \cdot V$ possible inputs and each takes $O(1)$ time to compute. The space complexity is $O(V)$ because we only need to store $X[k - 1, \cdot]$ in order to compute $X[k, \cdot]$. 