Instructions

Homework should be done in groups of one to three people. You are free to change group members at any time throughout the quarter. Problems should be solved together, not divided up between partners. A single representative of your group should submit your work through Gradescope. Submissions must be received by 10:00pm on the due date, and there are no exceptions to this rule.

You will be able to look at your scanned work before submitting it. Please ensure that your submission is legible (neatly written and not too faint) or your homework may not be graded.

Students should consult their textbook, class notes, lecture slides, instructors, TAs, and tutors when they need help with homework. Students should not look for answers to homework problems in other texts or sources, including the internet. You may ask questions about the homework in office hours, but not on Piazza.

Your assignments in this class will be evaluated not only on the correctness of your answers, but on your ability to present your ideas clearly and logically. You should always explain how you arrived at your conclusions and justify your answers with mathematically sound reasoning. Whether you use formal proof techniques or write a more informal argument for why something is true, your answers should always be well-supported. Your goal should be to convince the reader that your results and methods are sound.

For questions that require pseudocode, you can follow the same format as the textbook, or you can write pseudocode in your own style, as long as you specify what your notation means. For example, are you using “=” to mean assignment or to check equality? You are welcome to use any algorithm from class as a subroutine in your pseudocode. For example, if you want to sort list A using InsertionSort, you can call InsertionSort(A) instead of writing out the pseudocode for InsertionSort.

Required Reading Rosen 10.4 through Theorem 1, 11.1, 11.2 through Theorem 2

Key Concepts DAGs and topological orderings, graph search and reachability, rooted and unrooted trees
1. Given a piece of code, construct a directed graph where the methods (also called routines or functions) are the vertices and there is an edge from method $A$ to method $B$ if method $A$ calls method $B$.

(a) (2 points) Will the resulting graph be a rooted tree? Always, sometimes, or never? If it is a tree, what is the root?

(b) (2 points) What does it say about your program if the resulting graph is not a DAG?

(c) (4 points) Suppose that the resulting graph is a DAG. How can we use the graph to help us determine a debugging strategy? For our debugging strategy, we want to look for a bug in each method of the program, but only after we’ve determined that the methods it calls as subroutines don’t have bugs. Can your strategy be thought of as a modification of an algorithm from class? Which algorithm, and how did you modify it?

2. A daily flight schedule is a list of all the flights taking place that day. In a daily flight schedule, each flight $F_i$ has an origin city $OC_i$, a destination city $DC_i$, a departure time $d_i$ and an arrival time $a_i > d_i$. This is an example of a daily flight schedule for November 1, 2016, listing flights as $F_i = (OC_i, DC_i, d_i, a_i)$:

$$F_1 = (\text{Portland, Los Angeles, 7:00am, 9:00am})$$
$$F_2 = (\text{Portland, Seattle, 8:00am, 9:00am})$$
$$F_3 = (\text{Los Angeles, San Francisco, 8:00am, 9:30am})$$
$$F_4 = (\text{Seattle, Los Angeles, 9:30am, 11:30am})$$
$$F_5 = (\text{Los Angeles, San Francisco, 12:00pm, 1:00pm})$$
$$F_6 = (\text{San Francisco, Portland, 1:30pm, 3:00pm})$$

(a) (2 points) Given any daily flight schedule, describe how to construct a DAG so that paths in the DAG represent possible sequences of connecting flights a person could take. What are the vertices, and when are two vertices connected with an edge?

(b) (2 points) Why is your graph always a DAG?

(c) (2 points) What problem would you need to solve in your DAG to help you determine the maximum number of flights a person could take on a given day?

(d) (2 points) Draw the DAG you described for the given example of November 1, 2016 and give the maximum number of flights a person could take on that day.

3. In this puzzle, there are three rainbowfish who are swimming in a line to the right, and three lionfish who are swimming towards them in a line to the left. The rainbowfish and lionfish notice each other when there is enough space between them for one fish, and they must figure out how to trade places so that they can continue swimming. Only certain moves are allowed:

- Rainbow fish can only swim to the right, and lionfish can only swim to the left.
- A fish can swim one space forward into an empty space, or it can swim past one other fish into an empty space.
- A fish cannot swim past fish of the same species, only fish of the other species.
A configuration of this puzzle can be described using three R’s (representing the rainbowfish), three L’s (representing the lionfish), and one blank (representing the empty space). The starting configuration is \( \text{RRR}_L\text{LLL} \), and solving the puzzle means reaching the ending configuration \( \text{LLL}_R\text{RRR} \).

To help us solve this puzzle, we can use the graph search algorithm from class to construct a directed graph where the vertices are configurations and there is an edge from configuration \( A \) to configuration \( B \) if we can get from configuration \( A \) to configuration \( B \) in a single move. We would start with the configuration \( \text{RRR}_L\text{LLL} \) and determine that there are two outgoing edges from there, to \( \text{RR}_R\text{LLL} \) and \( \text{RRRL}_L\text{LL} \). We would continue from each of those vertices until we have a large graph where it is not possible to create any more outgoing edges.

For this problem, continue the graph from the vertex \( \text{RR}_R\text{LLL} \) until you get to a point where it is not possible to add further outgoing edges. This corresponds to considering all possible sequences of moves, where the first move is a rainbowfish swimming to the right. You could similarly continue the graph from the vertex \( \text{RRRL}_L\text{LL} \) to consider all possible sequences of moves where the first move is a lionfish swimming to the left, but you do not need to construct that part of the graph.

Use the graph you constructed to answer the following questions. You do not need to turn in a copy of your graph, and you do not need to explain your answers for this question unless prompted.

(a) (2 points) Is your graph a rooted tree? Why or why not? If it is a tree, what is the root?
(b) (2 points) Draw the part of your graph that shows \( \text{RRL}_R\text{LLL} \) and all the configurations that can be reached from there.
(c) (2 points) Call a configuration a dead end if it has no outgoing edges. How many dead end configurations appear in your graph?
(d) (2 points) Give a solution to the puzzle that starts with a rainbowfish move.
(e) (2 points) Use your answer to part (d) to give a different solution that starts with a lionfish move.
(f) (2 points) How many different configurations can be reached from \( \text{RR}_R\text{LLL} \)? Include the configuration \( \text{RR}_R\text{LLL} \) as reachable from itself.
(g) (2 points) Your graph should be a DAG. Use the algorithm from class for layering a DAG to find all configurations in your graph that are in the same layer as \( \text{RL}_R\text{LLL} \).
(h) (2 points) Suppose you are using the graph search algorithm from class, and at some point in the algorithm, \( X = \{ \text{RRR}_L\text{LLL}, \text{RR}_R\text{LLL}, \text{R}_R\text{RLLL}, \text{RRL}_R\text{LL}, \text{RRL}_R\text{LL}\} \). What will the set \( F \) be at this point in the algorithm?

4. Let \( T_1 \) be the rooted tree consisting of a single root vertex. Let \( T_2 = T_1 \). For \( n \geq 3 \), let \( T_n \) be the rooted tree whose left subtree is \( T_{n-2} \) and whose right subtree is \( T_{n-1} \).

(a) (2 points) Draw the tree \( T_5 \).
(b) (2 points) Let \( L(n) \) be the number of leaves in \( T_n \). Find a recurrence that \( L(n) \) satisfies.
(c) (2 points) Let \( I(n) \) be the number of internal vertices in \( T_n \). Find a recurrence that \( I(n) \) satisfies.
(d) (2 points) Let \( V(n) \) be the number of vertices in \( T_n \). Find a recurrence that \( V(n) \) satisfies.
(e) (2 points) Let \( E(n) \) be the number of edges in \( T_n \). Find a recurrence that \( E(n) \) satisfies.

5. A sorting algorithm that uses a binary tree to sort a list of positive integers \( a_1, a_2, \ldots, a_n \) from largest to smallest can be described by the following \( n \) steps.

**Step 1: Construct the binary tree. Output the root value.**

The elements \( a_1, a_2, \ldots, a_n \) are the leaves of the tree, and we build up the tree one level at a time from there. From left to right, compare the elements in pairs, and put the larger of the two as the parent vertex. Do this at each level until reaching the root, which will be the largest element. Output the value at the root. For example, if the list to be sorted is 22, 8, 14, 17, 3, 9, 27, 11, the tree would look like this:
**Step 2: Recompute labels. Output the root value.**

In the second step, remove the leaf corresponding to that largest element, and replace it with a leaf labeled 0, which is defined to be smaller than all the other list elements. Recompute the labels of all vertices on the path from this 0 to the root. That is, relabel all vertices on the path from the 0 to the root by choosing the larger of the values of their two children. Then the root will become the second-largest element. Output the value at the root. In our example, the tree would now look like this:

![Tree Diagram]

**Steps 3 through $n$: Recompute labels. Output the root value.**

Repeat the same process as described in step 2. At the end, we will have output the entire list in decreasing order.

(a) (2 points) Trace through the algorithm as applied to the list 17, 4, 1, 5, 13, 10, 14, 6. Show the tree at each step.

(b) (4 points) If $n$ is a power of two, as in the example, how many comparisons are done at step 1? How many comparisons are done in each of the other steps?

(c) (2 points) If $n$ is a power of two, as in the example, how many comparisons are done throughout the entire algorithm? What is the order of the algorithm in $\Theta$ notation?