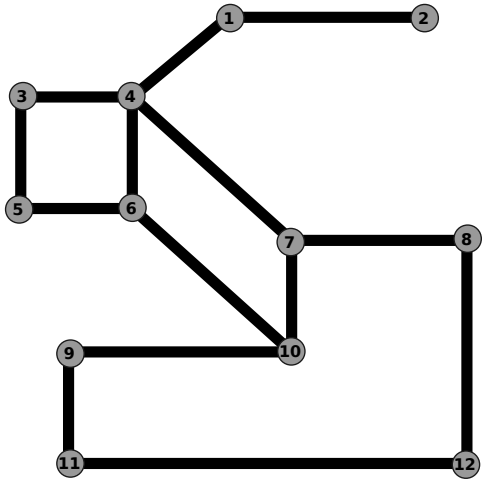
This homework covers Chapter 1 in the textbook. It is due in class on Wednesday, January 24. You can work with a partner on the written part of the homework, but you should write up your own answers in your own words to turn in. The homework also includes a computer program, which you should work on alone; see question number 4 for more information.

1. Consider this "corridor guard" problem: Given a network of intersecting corridors, you want to post guards in the intersections so that every corridor segment has a guard at at least one of its two endpoints. The problem is to do this using the minimum number of guards. Here is an example of such a network. The lines represent corridors and the twelve numbered circles are their intersections. Note, for example, that a guard in intersection number 6 can cover three corridor segments.



Note that there is a horribly inefficient algorithm that gives the correct solution: Consider all possible subsets of the set of intersections, throw out any subsets that don't cover all of the corridor segments, and then return a set of minimal size among those subsets that remain.

a) Find a reasonably efficient heuristic algorithm for selecting a set of intersections where guards will be posted. (It should be an algorithm that looks like it might give a correct solution.)

b) Show how your algorithm works for the example corridor network shown above.

c) Find a counterexample that shows that your algorithm does not, in fact, give the correct solution in all cases. That is, find a corridor network for which your algorithm produces a larger-than-necessary set of intersections.

2. You are in one node in an infinitely long doubly-linked list stretching off to the left and to the right. Exactly one node in the list contains a treasure, but you don't know which one. The only thing you can do is move left or right, one node at a time (and look in the current node for the treasure). Give an algorithm for finding the treasure. The algorithm must specify the sequence of left and right moves that you will make, and it must be certain to find the treasure eventually. Try for an algorithm that is as efficient as you can make it. Also, say what you can about the number of moves that your algorithm will take to find the treasure, if the treasure is N nodes away from your starting position.

3. (This is a variation on exercise 1-5 from the textbook.) You are given a list of numbers $s_1, s_2, \ldots, s_n$ and a target number T. (Think of T as the maximum weight that you can carry in your knapsack, and the numbers s_i as the weights of things that you want to put into the knapsack.) The problem is to select numbers from the list that will have the largest possible sum without going over T. For each of the following algorithms, find a counterexample that shows that the algorithm does not always give the best possible answer. That is, find a list of numbers and a target value for which the algorithm does not give the largest possible sum. (Remember to look for counterexamples that are as simple as possible!)

a) Try adding the numbers $s_1, s_2, \ldots, s_n$ to the sum in the order given, discarding any numbers that would make the sum bigger than T. (That is: Let $S = 0$; for i from 1 to n, if $S + s_i <= T$, add s_i to the set of included numbers, and let $S = S + s_i$.)

b) Sort the numbers into increasing order, and then apply the same procedure as in part **a)**.

c) Sort the numbers into decreasing order, and then apply the same procedure as in part **a)**.

4. This is a programming exercise. You should write the program yourself, without help from anyone except your professor. Turn in your work by copying your program into your homework folder in /classes/cs327/homework. The name of the program must be *Intervals.java*.

Your program should implement the correct solution to the "Select the Right Jobs" problem from Section 1.2 in the textbook. The solution is the algorithm *OptimalScheduling* given on page 11.

The essential problem is as follows: You are given a list of intervals $[a_i, b_i]$ for $i = 1, \ldots, n$, and you want to select intervals from the list that don't overlap; the goal is to find the largest possible number of non-overlapping intervals. (Although the textbook doesn't make it clear, we will consider intervals to be overlapping even if they only intersect at an endpoint; for example, the intervals $[1, 3]$ and $[3, 7]$ are overlapping.)

The *OptimalScheduling* algorithm is as follows: While the list of intervals is non-empty: Select an interval, I, that has the minimal right endpoint among all the intervals in the list, and add one to the count of non-overlapping intervals; delete I from the list and also delete all intervals that intersect I from the list.

For the program, you are given an input file that contains the list of intervals, in the following format: Each line of the file contains two integers, a and b, with $a < b$, representing the interval $[a, b]$. The two numbers are separated by a space. The output of the program should state the maximum number of non-overlapping intervals. You are not required to output the actual intervals. The program can either get the name of the file from a command-line argument, or it can ask the user to type in the name of the file.

The folder /classes/cs327/hw1-data contains sample data files for the program. The correct answers for these files are, I believe, as follows: *test0.txt*: 3, *test1.txt*: 4, *test2.txt*: 24, *test3.txt*: 232.