Homework 2

Jed Pulley & Matthew Starr

Question 1: You are given a data structure *H* that currently stores *N* distinct integers. *H* supports an operation minmax() that removes either the minimum or the maximum number in it in constant time – you will know the number, but you won’t know if it was maximum or minimum when it got removed. Using minmax(), **write a pseudo-code** to output the numbers from *H* in a sorted order into an array of length *N*. You are not allowed to use additional data structures/arrays for storage, but you may use variables for additional storage. You are also not allowed to sort the output array directly; so, the complexity should be *O*(*N*). [15]   
  
int minIndex = 0, maxIndex = N;

int[] arr = new int[N];

while(minIndex < maxIndex) do:

int lastVal = minmax();

int currVal = minmax();

if(currVal < lastVal):

arr[minIndex++] = currVal;  
 arr[maxIndex--] = lastVal;

else:

arr[maxIndex--] = currVal;  
 arr[minIndex++] = lastVal;

/\*

if the length of the array is even, then the last two indexes will butt up against each other and already have been selected. If it’s odd, then there is one last index sandwiched b-n the two that needs to be added  
\*/

if(N % 2 == 1):

arr[minIndex] = minmax(); //arbitrarily choose minIndex, since it’s the same as maxIndex

Question 2: Given an array of *N* distinct non-zero integers and an integer *k*, 0 *< k ≤ N*, you have to flip the sign of *k* numbers in the array such that the resultant array has the maximum possible sum. You may flip a number at most once. Design an *O*(*N*) time algorithm for selecting the numbers to flip. [15]

The goal is to find the kth smallest items and flip those signs. Since we are required to flip the signs, doing so to the k smallest numbers will have the least effect on our maximum possible sum.

For this implementation, we will employ QuickSelect to find the kth smallest number in O(N) time. Couple this with the [Median of Medians](https://en.wikipedia.org/wiki/Median_of_medians) approach to ensure O(N) time. Once we have that, we can then scan through the array, check to see if it is <= our kth smallest number, and if it is, flip the sign in constant time. Dropping our constants gives us O(N) time complexity.

Question 3: Professor X has to commute from her university to her home. Her car has a maximum fuel capacity of *C*, it consumes fuel at the rate of *r* gallons per minute, and the tank is filled at a rate of *f* gallons per minute. She has to stop at several fuel stations to be able to make the trip as *C* is small, but luckily big enough to sustain a trip from one fuel station to the next along the way. Assume that she starts with an empty tank at a gas station right by the university.

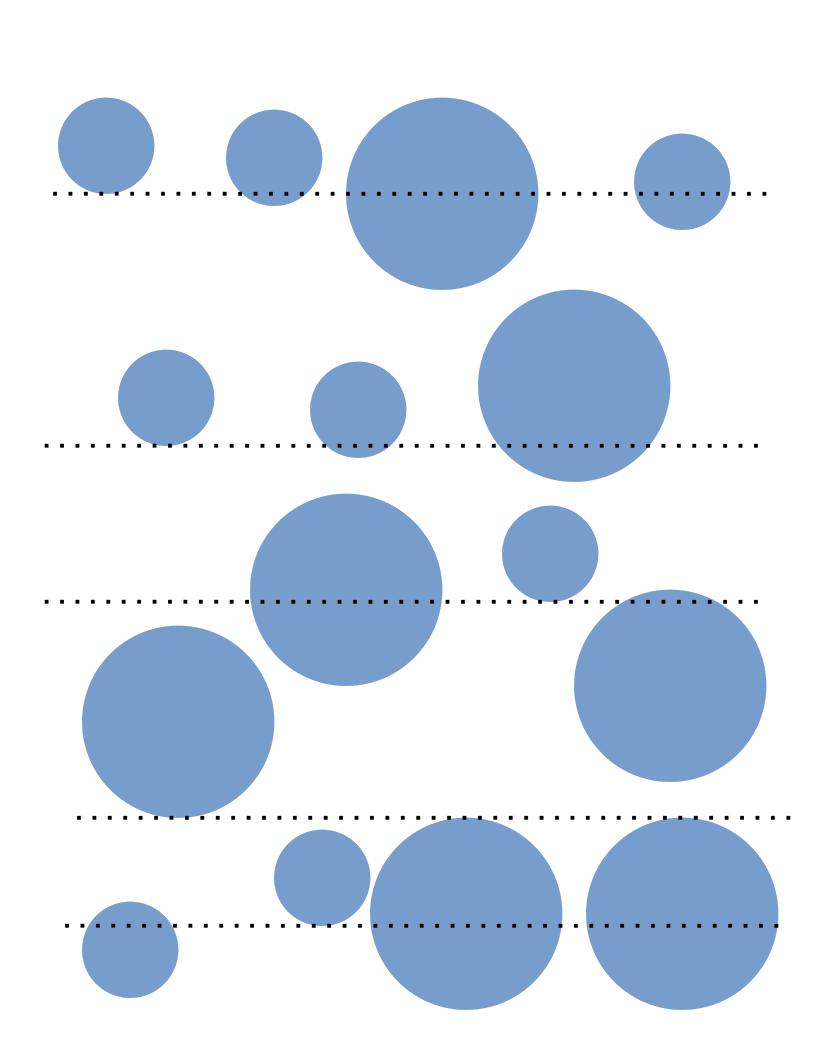
Describe a greedy strategy by which Professor X will reach her home while minimizing the amount of time she has to stop for fuel. [5]  
  
The goal here is to stop at the least number of gas stations as possible to get home. In this regard, every time she is unable to make it to the next gas station, she must fill up at the one she is at. If each fuel station is just far enough away that capacity C can get to each gas station, she must fill up at each gas station. Otherwise, she fills up at a gas station only if she is unable to make it to the following stop. Professor X will have to fill up for at least the amount of gas needed to get her home, so the amount of time it takes to fill up is the same as if she only filled up enough to get to the next fuel station. However, in the case that she can go further than a single stop, this will allow her to maximize her time by filling up only when she is unable to make it to the next one. By taking as much fuel as she can when she does stop, she is filling up greedily.

Question 4: Mr. Y wants to cross a wide street, which is flooded. Having the latest pair of Jordans on, Mr. Y does not want to get them wet. He notices that there are a few strategically laid bricks on the street in a straight line, such that from any brick he can always jump to the next and he can even sometimes jump over a few. His strategy to cross is at any point to jump to the farthest brick that he can. Use an exchange argument to prove that his strategy will result in the fewest jumps. [15]

Greedy Solution: A = {a1, a2,...,ak}  
Optimal Solution: O = {o1, o2,...,om}  
  
We will say that oi differs from ai, giving us:  
A = {a1, a2,...ai,....,ak}  
O = {o1, o2,...oi,...om}  
We will then substitute ai in place of oi. We that oi is the first time we are diverging in this algorithm. Given that our strategy is to jump to the furthest brick possible, by substituting ai, we can rest assured that it is just as good if not improving the optimal solution. Perhaps the optimal solution at oi chose to jump one brick closer instead of the true max. Substituting our greedy algorithm here will only continue to better the optimal solution or match it as we replace it entirely. Therefore, our greedy algorithm is just as good as any optimal solution, and thus is optimal itself.

Question 5: You are given *N* circles in a two-dimensional plane – each circle is identified by its center and its radius. *Design an O*(*N* log *N*) *greedy algorithm to find the minimum number of vertical lines you need to draw so that each circle is intersected by at least one vertical line.* You must clearly explain your greedy strategy and how it can be implemented. **Prove correctness; no need of formal proof, but your arguments should be precise.** [15]

Illustration (Explanation below):



Explanation:

For the algorithm to be greedy, we need each line to cut through as many circles as possible. This algorithm must also be completed in O(nlogn) time. We are given the radius and the center values. We can compute each circle’s ending x vertex by adding the radius value to its center value. For each circle, this takes linear time. Merge-sorting the circles based on the x-values found takes O(nlogn) time, as doing this will allow us to get the greedy output we desire. Each line must be placed wherever the first circle ends, which will allow for the max number of circles to be overlapped. Any circles that had started after the first circle will be touched at this point. The process is then repeated starting at the x-value of the line drawn and continuing until the next circle which starts after the x-value of the previous line ends. At this point, another line is drawn, and the process repeats until the full plane has been analyzed. This process takes linear time. Since the largest part of this algorithm is the merge-sorting procedure, the algorithm takes O(nlogn) time. By assuring each line is as far right as possible, we can show that this is a greedy algorithm.

Question 6: You are given a string *X* of length *n* and another string *Y* of length *m*. The task is to count how many index-disjoint sub-sequences of *X* are the same as *Y* . Thus, if *X* = GAXTYAWBGTAUGBTABGRGTAXB and *Y* = GTAB, then the answer is 3 as shown by the red fonts. Notice that we cannot select the underlined GTAB as it overlaps with a red GTAB, i.e., among overlapping sub-sequences that form *Y* , you can select only one of them. Describe an O(m + n) time greedy algorithm to obtain the count. **Write a pseudo-code.** [15]  
  
Below is java code that solves this problem in **O(n)** time instead of **O(m + n).** I do this by just continually incrementing the target counter and using modulo division to find the proper index. After we’re done looping through, I just use integer division to divide the target counter by the length of the Y array and that gives us the total amount of index-disjoint subsequences. Adding or removing any of the values in the Y array results in our desired output so long as it does not exceed the length of X, which is a constraint from the problem as well..

class subsetCount {

public static void main(String[] args) {

char[] X = {'G', 'A', 'X', 'T', 'Y', 'A', 'W', 'B', 'G', 'T', 'A',  
 'U', 'G', 'B', 'T', 'A', 'B', 'G', 'R', 'G', 'T', 'A', 'X', 'B'};  
 char[] Y = {'A', 'X'};  
 int n = X.length;  
 int m = Y.length;  
 int target = 0;  
 int targetIndex, subCount;  
  
 for(int i = 0; i < n; i++) {  
 targetIndex = target % m;   
 if(X[i] == Y[targetIndex])  
 target++;  
 }  
 subCount = target / m;  
 System.out.println("Subcount is: " + subCount);  
 }  
}