CS 31: Midterm Exam

Thomas Monfre

February 12, 2019

Problem 1

So I don't know how to do this...

Problem 2

This is a classic example of the hat retrieval problem. Let us define A as the event in which Professor Cormen gets his hat back on a given visit to the tavern. Let X_A be the indicator random variable of A such that $X_A = I\{A\}$. Then, $X_A = 1$ if Professor Cormen gets his own hat back, and 0 if he does not. Let us note that the expected value $E[X_A]$ of X_A is equal to the probability of X_A. We can prove this with the following. Let $\bar{A}$ be the complement of A.

$$\begin{aligned} E[X_A] &= E[I\{A\}] \\ &= 1 * Pr\{A\} + 0 * Pr\{\bar{A}\} \\ &= Pr\{A\} \end{aligned} \tag{1}$$

Let us assume the tavern has k customers in it. Since Professor Cormen is one customer among k and the hat retrieval professional returns hats back to customers in a random order, the probability that Professor Cormen gets his hat back is equal to the probability that the ith customer gets their hat back. Let X denote the number of customers that get their own hat back on a given visit. Then, $X = \Sigma_{i=1}^{k} X_A$.

Using the definition of expected value and linearity of expectation, we can then compute the expected number of customers that get their own hat back among the k customers on a given visit as the following.

$$\begin{aligned} E[X] &= E\left[\Sigma_{i=1}^{k} X_i\right] \\ &= \Sigma_{i=1}^{k} E[X_i] \\ &= \Sigma_{i=1}^{k} Pr\{A_i\} \\ &= \Sigma_{i=1}^{k} 1/k \\ &= 1 \end{aligned} \tag{2}$$

Therefore, among the k customers in the tavern, we expect 1 to get their hat back on a given visit.

Generalizing this to the scope of the problem, we know Professor Cormen visits the tavern n times. Let k denote the number of customers in the tavern on each visit. On the first visit, $k = 1$. On the second, $k = 2$, and so on until on the nth visit, $k = n$. Since the expected number of customers that get their own hat back among the k customers on a given visit is 1, Professor Cormen is one customer among the k, and each customer is equally likely to be the single customer who gets their hat back, the probability Professor Cormen gets his own hat back on a given visit is $1/k$.

Therefore, across all n visits, the expected number of times Professor Cormen gets his hat back is equal to a sum of the probability that he gets his hat back on a given visit. We therefore have $\Sigma_{k=1}^{n} 1/k$. Let us note that this is equal to the harmonic series. As shown in class, $\Sigma_{k=1}^{n} 1/k = \Theta(\lg n)$. Therefore, the expected number of times Professor Cormen gets his own hat back in his first n visits to the tavern is $\Theta(\lg n)$.

Problem 3

Let us first observe that $X[1...n]$ and $Y[1...n]$ each contain n numbers, therefore they are of the same length. Furthermore, they are both in sorted order. Knowing this, let us then observe the following cases. If the last element in X is less than or equal to the first element in Y, then since X and Y are in sorted order and are of the same length, the lower median is the last element in X because the two arrays put side to side will be in sorted order. Similarly, if the last element in Y is less than or equal to the first element in X, then since X and Y are in sorted order and are of the same length, the lower median is the last element in Y because the two arrays put side to side will also be in sorted order. Lastly, if neither of the above cases holds, but X and Y are each of length 2, then the lower median is the larger of the two elements at index 0 in each array. This occurs because X and Y strattle each other, indicating the last element in X is greater than the first element in Y. Then, the two smallest elements between X and Y are the first elements in each array. Since X and Y are each of length 2, the lower median is therefore the larger of the two smallest elements.

If neither of these cases hold, then the lengths of X and Y are greater than 2 and they strattle each other, indicating some overlap occurs between X and Y. Then, since X and Y are of the same length and are each in sorted order, we can begin to prune X and Y in size in order to find the lower median of the two. To do so, let us compute the lower median of each of X and Y. Let us denote X_m as the lower median of X and Y_m as the lower median of Y.

Then, if $X_m = Y_m$, the lower median of the two arrays put together will be either of X_m or Y_m because when arranged in sorted order, X_m and Y_m will both appear in the middle. We know this to be the case because all elements to the left of X_m in X must be less than or equal to X_m and all elements to the left of Y_m in Y must also be less than or equal to Y_m. Similarly, all elements to the right of X_m in X must be greater than or equal to X_m and all elements to the right of Y_m in Y must also be greater than or equal to Y_m. So, if X and Y were arranged in order, X_m and Y_m would both appear in the middle. Hence, the lower median would be either of the two since they are equal.

If $X_m < Y_m$, then we know all elements to the left of X_m in X cannot be contendors to be the lower median of the two arrays. If we were to arrange X and Y in sorted order we would see that these elements would occur to the left of the lower median since they are less than X_m and X_m will either be the final lower median of X and Y or it will occur to the left of the true lower median. Similarly, all elements to the right of Y_m in Y cannot be contendors to be the lower median of the two arrays. If we were to arrange X and Y in sorted order we would see that these elements would occur to the right of the lower median median since they are greater than Y_m and Y_m will either be the final lower median of X and Y or will occur to the right of the true lower median.

An almost identical case holds if $Y_m < X_m$. All elements to the left of Y_m in Y and all elements to the right of X_m in X cannot be contendors to be the lower median of the two arrays. In this case and that outlined above, we can then reduce the scope of our problem and only consider elements that are still in contention for the lower median.

Using these cases, let us then define a recursive function that computes the lower median of all $2n$ elements in arrays X and Y.

```
COMPUTE-MEDIAN(X, Y)
 1  k = X.length
 2  if X[k − 1] ≤ Y[0]
 3       return X[k − 1]
 4
 5  elseif Y[k − 1] ≤ X[0]
 6       return Y[k − 1]
 7
 8  elseif k = 2
 9       return MAX(X[0], Y[0])
10
11  else
12       X_m = X[⌊(k−1)/2⌋]
13       Y_m = Y[⌊(k−1)/2⌋]
14
15       if X_m = Y_m
16            return X_m
17
18       elseif X_m < Y_m
19            X = remove all elements to the left of X_m in X
20            Y = remove all elements to the right of the upper median in Y
21            // note: the upper median of Y is Y_m if Y is of odd length
22            // note: the upper median of Y is the element to the right of Y_m if Y is of even length
23            return COMPUTE-MEDIAN(X, Y)
24
25       elseif Y_m < X_m
26            Y = remove all elements to the left of Y_m in Y
27            X = remove all elements to the right of the upper median in X
28            // note: the upper median of X is X_m if X is of odd length
29            // note: the upper median of X is the element to the right of X_m if X is of even length
30            return COMPUTE-MEDIAN(X, Y)
```

We can prove that the above algorithm abides by the cases outlined above with the following. First, let us observe that we always remove the same number of elements from X and Y on each recursive call. This is because the number of elements to the left of the lower median in one is identical to the number of elements to the right of the upper median in the other. To prove this, if A and B are both of odd length, then the upper and lower median in both are identical. Since X and Y begin of identical length and the median of each is the middle value, the same number of elements occur on either side. If A and B are both of even length, then the number of elements to the left of the lower median of one is 1 less than the number of elements to the right of the lower median in the other. This is because the upper and lower medians are not the same element in each array. Therefore, one additional element exists to the right of the lower median in each. This additional element is the upper median. So, if we only consider elements to the right of the upper median, we then have an identical number of elements as those to the left of the lower. Since we remove all elements to

the left of the lower median in one and all elements to the right of the upper median in the other on each recursive call and we have proven that these subarrays are of identical length, we therefore always remove the same number of elements from X and Y on each recursive call.

Second, let us observe that we will never reach the case where X and Y are each of length 1. Since we always prune around the upper and lower medians, if X and Y are of length 4, we will recurse on two arrays of length 3 since the lower median will be at index 1 and the upper median will be at index 2 in each array. This is because only one element exists at index less than 1 and only one element occurs at index greater than 2. If X and Y are of length 3, then we will recurse on two arrays of length 2 since the lower and upper median will both occur at index 1 and only one element exists to the left of index 1 and only one element exists to the right of index 1. Lastly, if X and Y are both of length 2, then we return at line 9 and do not recurse further.

Then, the base cases in which the elements of each array would be in sorted order if one was concatenated to the other are handled at lines 2 and 5. There we return the last element in the array that would comprise the first half of the combined arrays. The base case in which the length of each array is 2 is handled on line 8. There, we return the larger of the two smallest elements among all of X and Y. The recursive cases are handled at line 11 and below. There we compute the medians and appropriately prune X and Y according to their values. Note that the case in which the two medians are identical is handled at line 15.

We can prove this algorithm runs in $O(\lg n)$ time with the following. On each recursive call we make, we prune each of X and Y in half (or slightly less than half if of even length). In the worst case, we never find that $X_m = Y_m$ and we never find that X and Y do not overlap (the base cases). In this case, we keep halving each array until we hit the base case in which each is of size 2. Then, we make a constant time call to MAX and return its result. Since we halved each array on each iteration and continued doing so until each array was of size 2, then in the worst case we have $\lg n$ recursive calls. Since the computations made in each case all take constant time, we then have a run-time of $O(\lg n)$ for this algorithm.

Problem 4

Part A: In order to implement a min-priority queue so that each INSERT and DECREASE-KEY operation takes $O(1)$ time and each EXTRACT-MIN operation takes $O(k)$ time, let us perform the following. We will back this data structure with a hash table implemented with an ArrayList. We will use the chaining method, such that each index in the array (represented by a hash value) will be a Circular Doubly Linked List with a Sentinel Node. Let us define the hash function for this problem as each element's key. Since the keys are integers in the range 0 to k, we therefore will have $k+1$ possible hash values corresponding to $k+1$ indices in the hash table. Note for this problem we initialize the hash table to be of length $k+1$ where each index is initialized to an empty linked list. Also note that we will not keep a notion of a load factor and therefore will not grow the array to be of larger size at any point.

Then, for a call to INSERT, where x is the element to be inserted, we will first compute x's hash as $x.key$. Then, since our hash table is backed with an ArrayList, we will have a constant-time lookup to grab the linked list at index $x.key$ in the table. Then, since each index holds a Circular Doubly Linked List with a Sentinel Node, we will simply append the given element to the end of the linked list. We can perform this operation in constant time by first grabbing a reference to the last element in the linked list with $sentinel.next.prev$. We then adjust pointers to insert the element by setting this object's $next$ value to $sentinel.next$, setting $sentinel.next.prev.next$ to this object, setting this object's prev value to $sentinel.next.prev$, then setting $sentinel.next.prev$ to this object. Since each of these operations takes constant time, each INSERT operation takes $O(1)$ time.

For a call to DECREASE-KEY, we are given a reference x to an object and a value p to change x's key to. We can therefore update x's key in constant time by first setting $x.key = p$. Then, since the key of x has changed, we must move it to the linked list associated with key p. To do so, we will first remove x from the linked list it is currently in by setting $x.next.prev = x.prev$ and $x.prev.next = x.next$. We will then call INSERT(x) which will move x to the linked list associated with the proper hash index. Since updating x's key and removing x from its old linked list takes constant time, and we have shown above INSERT takes constant time, each DECREASE-KEY operation will take $O(1)$ time.

For a call to EXTRACT-MIN, we have a bit more work to do. Specifically, we must find the first index in the table at which an element is stored in that index's linked list. To do this, we must iterate over each index in the table. For each index we will grab a reference to the $sentinel$ of the linked list held at that position. If $sentinel.next = sentinel$, then no element is stored in this linked list. Then, we proceed to the next index in the table. If $sentinel.next \neq sentinel$, then there exists a value in this linked list. So, we will remove $sentinel.next$ from the linked list, updating pointers as necessary, then will return it since this is the first element we found in the table. Since elements are stored in the table by hash value from smallest to largest and the hash function we are using is just an element's key, the first element we come across in the table is a valid element to be returned by EXTRACT-MIN. If multiple elements exist with the same key at a given index (indicating the linked list has more than one value), then the element we return will be that of the lowest key that was first inserted into the queue.

In the worst case, all elements inserted into the queue will have key k. Therefore, on a given call to

EXTRACT-MIN, we must iterate over all $k+1$ indices in the table until we find the first element in the linked list at index k to remove. In this case, we have a run-time of $O(k)$. Therefore, we have shown how to implement a min-priority queue so that each INSERT and DECREASE-KEY operation takes $O(1)$ time and each EXTRACT-MIN operation takes $O(k)$ time.

Part B: If the key values in the objects returned by calls to EXTRACT-MIN are monotonically increasing over time, then we know after all objects of key i are removed from the table, never again will objects with keys less than or equal to k exist in the table. In this case, we can improve our EXTRACT-MIN algorithm to take $O(m+k)$ time for m calls to EXTRACT-MIN. To do so, we will maintain our prior implementation of INSERT and DECREASE-KEY. For EXTRACT-MIN, however, we can take advantage of the monotonically increasing keys.

Specifically, let us hold as an instance variable a reference to the last hash index in which we removed an element. Let us call this variable $last_index$. We will initialize $last_index$ to 0. Then, on each call to EXTRACT-MIN, we know we can ignore all hash table indices less than $last_index$. This is because the keys of the elements removed by EXTRACT-MIN are monotonically increasing. So, for a given call to EXTRACT-MIN, we will determine if any elements exist in the linked list stored at index $last_index$ in the table. If so, we remove the first element in the linked list. If not, we increment $last_index$ by 1 and check if the linked list at index $last_index$ is empty or not. We repeat this process until we have removed an element on this call to EXTRACT-MIN.

For m calls to EXTRACT-MIN, this algorithm will have a run-time of $O(m+k)$ altogether. This is because we are consistently moving to the right in the table. After we have exhausted a possible key value (i.e. found that its linked list is empty or removed the last element from it), we never must consider it again because the keys are monotonically increasing. Therefore, we can split the run-time of this algorithm into two pieces: the cost required to iterate over each possible key, and the cost required to remove an element from a linked list. As shown above, the run-time required to remove an element from a linked list is $O(1)$. So, for m calls to EXTRACT-MIN, we have $O(m)$ time required to remove m elements from a given set of linked lists. We then must determine the run-time required to iterate through each possible linked list. Since we initialize $last_index$ to 0, then in the worst case, $last_index = k$ on the mth call to EXTRACT-MIN. In this case, we iterated through all $k+1$ indices in the table. Therefore, the run-time required to iterate through each possible linked list is $O(k)$. So, for m calls to EXTRACT-MIN, we have $O(m)+O(k) = O(m+k)$.

Part C: In this case, we cannot have our hash function simply be the key of each object, since the keys are arbitrarily large. However, since we know the difference between the smallest key in the queue and the largest key in the queue is at most $k-1$ at any time, we only need at most k indices in the hash table. This will require changing our hash function. To do so, we will hold as an instance variable the smallest key in the table. Let us call this variable $smallest\_key$. Then, we will define our hash function as an element's key minus $smallest\_key$, such that $h(x) = x.key - smallest_key$. Since the keys are monotonically increasing, we know $smallest\_key$ is also monotonically increasing.

Then, for a call to EXTRACT-MIN, we first start at index 0 and determine whether or not the linked list at that location is empty. If it is, we move to index 1 and search again. If the linked

list at that location has an element x, we remove x from that linked list by updating pointers. Because we've removed an element however, it may be the case that the linked list is now empty, indicating $smallest\_key$ must increase. So, prior to returning x, we iterate forwards in the table until we've found an element y in a linked list at index i. Once we have, we set $smallest_key = y.key$. Furthermore, so as not to waste memory, we update our linked list by constructing a new table and inserting all linked lists at current index i or greater into the new table starting at index 0. This will require iterating through all possibly k remaining indices in the table and moving them to a new table. Then, new elements will hash into the proper index since $smallest\_key$ was updated and we updated the table to reflect the proper index values. So, after all of this work is done, we will have removed x from its previous linked list, updated $smallest\_key$ if necessary, and updated the table to reflect the new hash function and not waste memory. Then, we simply return x.

Therefore, we can split the run-time of this algorithm into three pieces: the cost to remove x from its original linked list, the cost to find the new $smallest\_key$ value, and the cost to move all linked lists at and to the right of the new $smallest\_key$ to a new table. As before, removing x from its original linked list takes $O(1)$ time since we are simply updating pointers. Since the difference between the smallest key in the queue and the largest key in the queue is at most $k-1$ on a given iteration, finding the new $smallest\_key$ will take $O(k)$ time in the worst case since we must iterate over each index in the table. Similarly, the run-time required to construct a new table is also $O(k)$ in the worst case since we must iterate over each index in the table and move the sentinel of the list to the new table. That being said, if we iterate over a positions to find a new $smallest\_key$, then we must only move $k-a$ linked lists into the new table since the values returned by calls to Extract-Min are monotonically increasing. Therefore, each Extract-Min operation takes $O(k)$ time.

We will implement Insert and Decrease-Key the same as part b, however note that our hash function has changed, causing slight changes to the implementation of these functions. Since the hash function is still constant time, however, we still have that each Insert and Decrease-Key operation takes $O(1)$ time.

Problem 5

Part A: In order to show the number of possible seams grows at least exponentially in m assuming $n > 1$, let us observe the following. When $n > 1$ there exist at least two columns in A. In this case, for every possible pixel in a row i, there are at least two possible pixels in row $i+1$ that could connect to a valid seam from rows $1...i$. There are exactly 2 possible pixels to choose from when the currently observed pixel has a column that occurs on a vertical edge of the photo (i.e. column 1 or column n) since there either exists no pixel to the left or no pixel to the right of the current. For all columns j such that $1 < j < n$ (i.e. the internal columns), there are three possible pixels in each row that could create a connected seam.

Therefore, since there exist at least 2 possible choices of pixels in each row, for a photo with m rows we have at least 2^m choices to make. This yields at least 2^m possible combinations of chosen pixels in the m rows, and therefore at least 2^m possible seams. Therefore, the number of possible seams grows at least exponentially in m when $n > 1$.

Let us note that the requirement that $n > 1$ makes sense. If $n = 1$ then there exists one column and hence 1 choice to make for all m rows. This yields $1^m = 1$ possible seams since $m > 0$ which does not grow exponentially. For all other positive values of n, however, we have exponential growth in m for the number of possible seams.

Part B: Let us first argue this problem has optimal substructure. Let i denote the current row and let j denote the current column such that $A[i, j]$ is the currently observed pixel and $d[i, j]$ is its disruption measure. Then, let us define our sub-problem as the optimal seam for rows $1...i-1$ such that the column of the seam at row $i-1$ is $j-1$, j, or $j+1$. Essentially, we seek to find the optimal way (associated with lowest disruption measure) to create a seam such that the current pixel $A[i, j]$ is reachable from the seam. Therefore, when computing the optimal way to construct a seam that reaches pixel $A[i, j]$, we rely on the optimal solution to the subproblem for pixels $A[i-1, j-1]$ (adjacent left), $A[i-1, j]$ (directly above), and $A[i-1, j+1]$ (adjacent right) since pixel $A[i, j]$ is reachable from all of and only these pixels. Since the optimal way to reach a given pixel therefore relies on the optimal solutions to these subproblems and there are three options available when moving to a new row in a seam (i.e. the pixel directly below, adjacent left, or adjacent right), the solution to a given sub-problem needs to be accessed more than once.

We can prove this with a simple example. When computing the optimal path to a pixel $A[i, j]$, we rely on the optimal solution to the sub-problem for pixel $A[i-1, j-1]$ since it is above and to the left of the current pixel. When computing the optimal path to a pixel $A[i, j-1]$, we also rely on the optimal solution to the sub-problem for pixel $A[i-1, j-1]$ since it is directly above the current pixel. Since i and j are arbitrary pixels in this example, the optimal solution to a given sub-problem therefore applies to more than one solution.

Let us prove that the solution to each sub-problem is optimal with a classic cut-paste style proof. When observing a pixel $A[i, j]$, there are three possibilities for how to construct a seam path to $A[i, j]$: take the optimal solution for a path to the pixel above and adjacent to the left of $A[i, j]$, directly above, or above and adjacent to the right. Therefore, the optimal solution for a path to pixel $A[i, j]$ will be the optimal solution to each of the above cases that yields the smallest total

disruption measure plus the disruption measure $d[i, j]$ of the current pixel. Since we rely on an optimal solution to a sub-problem for each pixel that is reachable from $A[i, j]$, if there exists a better way to construct a path to $A[i, j]$ through pixels $A[i-1, j-1]$, $A[i-1, j]$, or $A[i-1, j+1]$ such that the sum of the disruption measures of each pixel on the path is smaller than that achieved with the optimal solution to the chosen pixel above $A[i, j]$, then such a solution was not optimal in the first place since there exists a more optimal way to achieve a path to $A[i-1, j-1]$, $A[i-1, j]$, or $A[i-1, j+1]$. We have therefore contradicted our assumption that the solution to each sub-problem is optimal since we found a more optimal way to reach $A[i, j]$ through $A[i-1, j-1]$, $A[i-1, j]$, or $A[i-1, j+1]$. Since we have contradicted this assumption, we know our optimal substructure holds.

Using this, let us define our recursive solution for this problem as follows. We seek to minimize the sum of the disruption measures for a seam that is reachable by the current pixel. Therefore, the optimal cost of a seam that ends at pixel $A[i, j]$ is the minimum cost of a seam that reaches pixel $A[i-1, j-1]$, $A[i-1, j]$, or $A[i-1, j+1]$ plus the disruption measure of the current pixel $A[i, j]$. Whatever sum yields the smallest total cost will be the optimal solution to constructing a seam that reaches pixel $A[i, j]$. If no pixels exist above the current pixel, we define the cost of the non-existent above pixel to be 0 so we only take into account the disruption measure of the current pixel. If no pixels exist to the left or to the right of the current pixel, we define the cost of those non-existent pixels as infinite so we never consider them an optimal solution. We can therefore define a seam-cost matrix c with the piece-wise function below.

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \\ \infty & \text{if } j = 0 \text{ or } j = n + 1 \\ \min(c[i-1, j-1], c[i-1, j], c[i-1, j+1]) + d[i, j] & \text{otherwise} \end{cases}$$

The matrix c therefore holds the minimal cost associated with the optimal way to construct a seam from some pixel in row 1 to pixel $A[i, j]$. Since we defined our optimal substructure as the minimal cost associated with this optimal choice, we then know that the last element in a seam from rows 1 to m is located at row m and column b where $c[m, b]$ is the smallest for all pixels in row m.

Since we therefore have a series of repeated sub-problems that need to be accessed multiple times, each of which relies on a series of optimal solutions to other subproblems, and we have defined an optimal recursive solution above, let us construct a dynamic programming algorithm in order to efficiently compute the optimal seam to remove. We cannot use a greedy approach since we must iteratively observe all possible seams. We cannot assume a local solution will lead to a global solution because we may have to make concessions at one point on the optimal seam in order to find a better seam segment later.

Let us therefore define our bottom-up dynamic programming algorithm as follows. We will first initialize our matrix c. We will store values of 0 for all m possible rows and all n possible columns as well as for row 0 in order to handle the base case shown in the recursive function. For columns 0 and $n + 1$ we set their values to ∞ to handle the side column case shown in the recursive function. Then, we iterate over all rows i and all columns j. For each one, we compute the optimal cost based on the piece-wise function shown above. For each pixel $A[i, j]$ we visit, we must consider the most optimal way to reach $A[i, j]$ through pixels $A[i-1, j-1]$, $A[i-1, j]$, and $A[i-1, j+1]$. Then,

after we have computed and stored all the optimal costs, we will walk back through the matrix c in order to output the chosen pixels in the optimal seam. To do so, we will first find the pixel in the last row with the smallest value in c. This will be the last pixel in the seam. Let us define the column of this pixel as b such that this pixel can be referenced as $A[m, b]$. We then will check the cost in c for pixels $A[m-1, b-1]$, $A[m-1, b]$, and $A[m-1, b+1]$. The smallest is the next pixel in the seam. We will then visit that pixel and repeat this process until we choose a pixel in row 1. The pseudocode for this algorithm is provided below.

COMPUTE-INTERVALS(S)

```
1  c = new matrix
2  // note that we iterate from row 0 through m in order to handle the base case
3  for i = 0 to m
4       for j = 1 to n
5            c[i, j] = 0
6       // set the side columns to cost infinity for this row in order to handle the side case
7       c[i, 0] = ∞
8       c[i, n + 1] = ∞
9
10 // compute seam segment costs
11 for i = 1 to m
12      for j = 1 to n
13           min_cost = c[i − 1, j − 1]
14           if c[i − 1, j] < min_cost
15                min_cost = c[i − 1, j]
16           if c[i − 1, j + 1] < min_cost
17                min_cost = c[i − 1, j + 1]
18
19           c[i, j] = min_cost + d[i, j]
20
21 // determine optimal seam
22 output = [ ]
23 for i = m down to 1
24      best_cost = ∞
25      best_column = 0
26      for j = 1 to n
27           if c[i, j] < best_cost
28                best_cost = c[i, j]
29                best_column = j
30
31      output.insert(0, [i, best_column])
32
33 // print chosen pixels in seam
34 for pixel in output
35      PRINT("Choose pixel at row " + pixel[0] + " and column " + pixel[1])
```

We can prove this algorithm runs in $\Theta(mn)$ time with the following. At lines 3 and 4, 11 and 12, and 23 and 26, we have a set of doubly nested for loops. The outer iterates m times and the inner iterates n times. Since each loop iterates to its full range (i.e. it does not break early), we have an upper bound of $O(mn)$ and a lower bound of $\Omega(mn)$ for each of these loops, yielding $\Theta(mn)$. Since there are no other loop structures in this algorithm, we therefore have a run time of $\Theta(mn) + \Theta(mn) + \Theta(mn) = \Theta(mn)$.

We can prove this algorithm has a space requirement of $O(mn)$ with the following. On lines 1 through 8, we construct an $m \times n$ matrix to store seam segment costs and initialize each value. Since we iterate over each possible row and column, we use this entire matrix yielding an upper bound space requirement of $O(mn)$. The only other data structure used in this algorithm is the *output* array constructed at line 23. Since this array is one-dimensional and holds m values in it, we have a space requirement here of $O(m)$. Therefore, since $n > 1$, we have a total space requirement of $O(mn) + O(m) = O(mn)$.

Problem 6

Part A: Let us define the potential function $\Phi(T)$ on the heap T to be the number of elements in the heap times the height of the heap when represented as a binary tree. For an empty heap T_0 from which we start, we have $\Phi(T_0) = 0$. Since the number of objects in the heap is never negative and the height of the heap when represented as a binary tree cannot be negative, the heap T_i (for some integer i that represents the number of elements in the heap) has non-negative potential. Therefore, $\Phi(T_i) \geq 0 = \Phi(T_0)$.

We can then formally define our potential function as $\Phi(T) = n \times \lceil \lg(n+1) \rceil$, where n represents the number of elements in the heap. Using this, we can construct a table to show the actual cost, change in potential, and amortized cost of each operation. Observe that the amortized cost is equal to the actual cost plus the change in potential for each operation.

operation	actual cost	$\Delta\Phi$	amortized cost
INSERT	1	$(n+1) * \lceil \lg(n+1) \rceil - n * \lceil \lg(n) \rceil = \lg(n+1)$	$1 + \lg n + 1 = \Theta(\lg n)$
EXTRACT-MIN	$1 + \lg n$	$(n-1) * \lceil \lg(n-1) \rceil - n * \lceil \lg(n) \rceil = -\lg(n-1)$	$\lg n - \lg(n-1) + 1 = \Theta(1)$

The thought process here is as follows. For INSERT, the actual cost of this operation is 1 since we must add a new element to the bottom of the heap. Given our potential function, we then add $\lg(n+1)$ units of potential, creating a total amortized cost of $\Theta(\lg n)$. This reflects a pre-payment for potentially bubbling up the heap on insertion or down the heap on deletion. For EXTRACT-MIN, we have an actual cost of $\lg n$ since we must bubble down the the heap. Given our potential function, we then lose $-\lg(n-1)$ units of potential. This makes sense since we have removed an element from the heap. Therefore, we have a total amortized cost of $\Theta(1)$.

As an explanation for these results, let us acknowledge the following. On a given call to INSERT in which we add an element x to the heap, there exist at most $n/2$ elements above x. Furthermore, half of those $n/2$ elements exist one level above x, indicating there exist only $\lg n$ possible elements that we could swap with x. Therefore, on a call to INSERT in the average case, we will not swap x far up the tree because only one case exists where we swap x at all $\lg n$ positions above it. For EXTRACT-MIN, however, we take an element y at the leaf level and move it to the root, then bubble y down at most $\lg n$ levels until it is in the appropriate location in the tree. Since y came from a leaf level and at any given time $\lfloor n/2 \rfloor$ elements exist above the leaf level in the heap, in the average case, we will swap y $\lg n$ positions down the tree.

Let us acknowledge, however, that an element y will not be swapped to the root and bubbled-down until the number of elements inserted into the heap after x are removed. Those individual elements may not be removed, but that number will. Therefore, both INSERT and EXTRACT-MIN will not take $\Theta(\lg n)$ time on each function call because the added and removed elements will not bubble up/down exactly $\lg n$ times. Therefore, on the average case, we can allocate the cost associated with bubbling an element down to the INSERT method, yielding an amortized cost of $\Theta(\lg n)$ for the INSERT operation and $\Theta(1)$ for the EXTRACT-MIN operation.

Part B: Professor Fifofum has wasted his life because we can never guarantee the amortized cost of INSERT and the amortized cost of EXTRACT-MIN will be $o(\lg n)$. $o(\lg n)$ states the amortized

cost of each operation is strictly less than $\lg n$. However, it only takes one counter example to illustrate this cannot be. If the elements inserted into the heap arrive in strictly decreasing order, then on each call to INSERT, we must make $\lg n$ swaps since the added element belongs at the root. Then, the amortized run-time of INSERT cannot be $o(\lg n)$ since we have $\Theta(\lg n)$ swaps on each function call. Therefore, we cannot use an amortized cost of INSERT to pre-pay for the costs associated with EXTRACT-MIN, since no extra allocated costs exist – we use all $\lg n$ swaps on each iteration. Therefore, we simply cannot guarantee to get an amortized run-time of $o(\lg n)$ for EXTRACT-MIN.

So, Professor Fifofum has wasted his life because we cannot guarantee the amortized cost of INSERT and the amortized cost of EXTRACT-MIN will be $o(\lg n)$ for all possible input values, since we have provided an example to the counter.

Problem 7

Part A: For this problem, let us first outline some basic notation. Let us refer to some chord c as an ordered pair of angles at which each of c's two endpoints intersect the circle relative to the positive x-axis that starts at the circle's center. We will denote this ordered pair as (c_1, c_2) where $c_1 < c_2$. By requiring $c_1 < c_2$, for all chords, the start endpoint of each chord is that which produces the smaller angle with the center of the circle, and the finish endpoint of each chord is that which produces the larger angle with the center of the circle. For example, for a chord c that has endpoints intersecting the circle at 45° and 175°, we denote c as $(45, 175)$ where $c_1 = 45$ and $c_2 = 175$.

Using this, let us then think spatially about this problem. Each endpoint of every chord is represented as a numeric value corresponding to the angle that endpoint makes with the positive x-axis starting at the center of the circle. In this regard, the support of possible values for each endpoint is $[0, 360]$. Furthermore, the values for each endpoint not only denote the angle that endpoint makes with the center of the circle, but it denotes the location of that endpoint on the outer edge (think circumference) of the circle. In this regard, rather than thinking in terms of angles endpoints make, let us think in terms of location on the outer edge of a circle. Then, on a counter-clockwise walk of the outer edge of the circle starting at 0° and going to 360°, we will come across various endpoints based on the angle our current position makes with the center of the circle.

Using this, let us then outline the cases in which two chords i and j intersect. Using the spatial thought of performing a counter-clockwise walk of the outer edge of the circle, i and j intersect if we hit the start endpoint of i before the start endpoint of j then after hitting the start endpoint of j hit the finish endpoint of i before hitting the finish endpoint of j. More formally, two chords i and j intersect if and only if $i_1 < j_1$ and $j_1 < i_2 < j_2$ using the notation defined above. Observe that we never have to consider the case in which two endpoints are equal because no two chords share an endpoint, as described in the problem definition.

Therefore, in order to compute the number of intersections, let us observe the following characteristics. We can identify a single intersection between two chords by ordering each endpoint by the value of the angle that point makes with the center of the circle, then computing the number of start endpoints that occur between the start and finish points of a single chord. Once we've computed this number, if we then remove the given chord from contention for producing further intersections, we can ensure we do not double-count future intersections. In order to maintain this data structure and compute the proper number of intersections for a given chord, let us use an order-statistic tree T. T will hold nodes representing individual endpoints of chords. Let us observe that an order-statistic tree is optimal for this problem because it is a balanced binary tree that allows us to compute the rank of certain elements easily. Furthermore, using the approach to computing inversions (as solved in Homework 2), it allows us to easily compute the number of nodes with a larger or smaller value than a currently observed node efficiently.

Let us then define this algorithm as follows. We are given an array C of chords represented as ordered pairs (tuples), as discussed above. We will then first construct EndPoint objects for each chord in C. An EndPoint object will hold a *key* attribute denoting the angle that endpoint makes with the center of the circle, as well as a $classification$ attribute denoting whether or not that endpoint is a "start" or "finish" point on a chord. Recall we define a start point on a chord as one that is strictly

less than the endpoint. Lastly, each EndPoint object will also hold a $partnerValue$ attribute. For a given start endpoint, this will be the angle value of the corresponding finish endpoint. For a given finish endpoint, this will be the angle value of the corresponding start endpoint. Constructing these objects will allow for quick look-ups, which we will take advantage of later.

Then, after constructing all EndPoint objects for each chord, we will sort our collection of EndPoint objects by their *key* attribute. To do so, we will make a call to MERGE-SORT. By ordering the endpoints by the angle they make with the center of the circle, we can then make a counter-clockwise walk around the edge of the circle, visiting endpoints that we see. Doing so lets us take advantage of the characteristics shown above of order-statistic trees in that computing the number of inversions can be performed efficiently. Then, after sorting our collection of EndPoint objects, we will construct our order-statistic tree T and hold onto a count of intersections and a count of elements in the tree, both initialized to 0.

Then, for each EndPoint object e, if e was classified as a start endpoint, we will add a new node with e's key and e's $partnerValue$ to T. We will then increment our count of number of elements in T. If e was classified as a finish endpoint, then since we ordered our endpoints by key and defined a start endpoint as strictly less than a finish endpoint, we know e's corresponding start endpoint must exist in T. Therefore, we will find a reference to it in T by making a call to TREE-SEARCH from CLRS Chapter 12. Let z denote this corresponding node. With this reference, since T is an order-statistic tree and we only ever insert start endpoints into T, all nodes in T that occur to the right of z have a greater value for the angle they make with the center of the circle. Since we iterate through nodes in the order of their angle and the only case in which a start node exists in T is if its finish endpoint has not been discovered, then all elements greater than z's key in T exist on a chord that intersects the chord (z, e). This occurs because all nodes i that exist to the right of z in T satisfy the constraint shown above that $z < i < e$. Therefore, we can compute the number of elements to the right of z in T with OS-RANK. Then, we remove z from T with RB-DELETE and continue iterating. After iterating through each endpoint, we then have a sum of all intersections. The pseudocode for this algorithm is provided below.

```
CALCULATE-NUMBER-OF-INTERSECTIONS(C)
 1  endpoints = [ ]
 2  for chord c in C
 3      start = new EndPoint object with key c_1, classification "start", and partnerValue c_2
 4      finish = new EndPoint object with key c_2, classification "finish", and partnerValue c_1
 5      endpoints.add(start)
 6      endpoints.add(finish)
 7
 8  endpoints = MERGE-SORT(endpoints)
 9  T = new OrderStatisticTree()
10  total = 0
11  numInTree = 0
12  for endpoint e in endpoints
13      if e.classification = "start"
14          z = new node with key: e.key
15          RB-INSERT(T, z)
16          numInTree = numInTree + 1
17      else
18          z = TREE-SEARCH(T, e.partnerValue)
19          sum = numInTree − OS-RANK(T, z)
20          total = total + sum
21          RB-DELETE(T, z)
22
23  return total
```

Let us analyze the run-time of this algorithm with the following. Since there are n chords in C, iterating over each on lines 2-6 takes time $\Theta(n)$. Since there exist two endpoints for each chord, then our call to MERGE-SORT at line 8 takes time $\Theta(2n \lg(2n)) = \Theta(n \lg n)$. Then, for each endpoint, we call RB-INSERT or TREE-SEARCH, OS-RANK, and RB-DELETE. RB-INSERT takes time $O(\lg n)$, as shown in CLRS 13.3. TREE-SEARCH takes time $O(h)$, where h represents the height of the tree, as shown in CLRS 12.2. Since our data structure is backed with a red-black tree, h is at most lgn, yielding $O(\lg n)$ for this operation. OS-RANK takes time $O(\lg n)$ as shown in CLRS 14.1. Lastly, RB-DELETE takes times $O(\lg n)$ as well, as shown in CLRS 13.4. Therefore, for n operations, we have $O(\lg n)$, yielding $O(n \lg n)$. We therefore have, $\Theta(n) + O(\lg n) = O(\lg n)$.

Part B: In order to output each pair of chords that intersect inside the circle, we will take advantage of the same structure and relationships outlined in part A, but will use a different data structure. Specifically, we will take advantage of the fact that our collection of EndPoint objects is ordered and use a Doubly Linked List as our main data structure instead of an order statistic tree. By using a Linked List, we will maintain outside references to vertices in the list even when the order of the list changes.

Then, as in part A, we are given a collection C of chords in the form of ordered pairs. So, we will first iterate through each chord c and construct EndPoint objects for each endpoint in c. Each EndPoint object now has four attributes: a key corresponding to the angle that endpoint makes

with the center of the circle, a classification as start or finish depending on which angle value is smaller, and two pointers to objects stored in memory. The first pointer called *partnerPointer* will be a reference to the EndPoint object associated with this chord c. This allows us to later grab a reference to the corresponding finish EndPoint object associated with each start EndPoint object when we iterate over each and add to our data structure. We will only use this reference for the start endpoint, so we will set *partnerPointer* to the finish endpoint object created for each chord c. The other pointer called *LLPointer* will be a reference to a node in the (yet to be created) Linked List that serves as our main data structure instead of the order statistic tree from part A. Since we have not constructed the Linked List, we will set the value of this pointer for each EndPoint object as *null*. We will only use this reference for the finish endpoint. This will be used when determining the intersections a given chord makes. After constructing these EndPoint objects, we then add them to a simple array *endpoints*.

Then, after we have iterated over all chords and constructed EndPoint objects for each, we will sort our collection of EndPoint objects using MERGE-SORT just like in part A. This allows us to take advantage of the aforementioned cases required to create an intersection. Then, we will create a Doubly Linked List called *currentCollection* that will act as our main data structure for this problem similar to the order statistic tree from part A. We then also initialie an integer *chordCount* to 0 that will be used to print the chord numbers that intersect.

Then, we iterate over each EndPoint object e in *endpoints*. For each one, we again have two cases: either e is a start endpoint or a finish endpoint. If e is a start endpoint, then we have not yet seen any part of the chord e is a part of. So, we construct a new node object for our Doubly Linked List with a key value of $e.key$ and a chordNumber of *chordCount*. Since *endpoints* is sorted, we can insert this node to the end of the linked list in constant time since we know that is the sorted position it belongs in. Now that we have constructed a node in the linked list, recall that we created two pointer attributes for all EndPoint objects that we previously constructed. Specifically, *partnerPointer* was a reference to the associated finish EndPoint object for a given start EndPoint object, and *LLPointer* was a reference to the node in the Linked List associated with the start EndPoint object. Since that node was not yet created, however, we initialized *LLPointer* to *null*. Now that we have constructed such a node for the EndPoint object e that we have determined is a start endpoint (based on its classification), we must update this *LLPointer* attribute for use when we are observing a finish EndPoint object. To do so, let us observe that the finish EndPoint object needs to hold this reference, and each start EndPoint object has a reference to its corresponding finish EndPoint reference in *partnerPointer*. Therefore, in order to set *LLPointer* for the finish EndPoint object of e, we set $e.partnerPointer.LLPointer$ to the newly created node in the Linked List. Since *partnerPointer* points to a finish EndPoint object associated with e and each finish EndPoint object has an initialized *LLPointer* attribute, we have successfully set the desired pointer to our node for e in the Linked List. After performing this action, we then increment *chordCount* for the next time we come across a start EndPoint object, then move onto the next iteration.

If instead e is a finish endpoint, then since *endpoints* is ordered by key, the corresponding start EndPoint object for e must exist in *endpoints*. Therefore, we must have set its *LLPointer* attribute to a node in the Linked List on a previous iteration of the loop. Then, the chords that intersect are those that have a start EndPoint between the start EndPoint associated with e and e itself since these endpoints must strattle each other. It cannot be the case that any chord exists completely

within the range of the start EndPoint associated with e and e because *endpoints* is ordered and each time we come across a finish endpoint, we remove it from the Linked List. Therefore, all nodes in the Linked List starting at the node associated with the start EndPoint of e and e itself must create intersections with the chord $(e.LLPointer, e)$. So, in order to output each one, let us simply iterate forwards in the Linked List starting at $e.LLPointer$ through the end of the list. For each node we visit, we print the chord number associated with the current start EndPoint as well as the chord number associated with the next start EndPoint in the Linked List. After iterating through each intersection, we then will have printed all intersections associated with $(e.LLPointer, e)$. We then remove $e.LLPointer$ from the Linked List in order to maintain our structure for this problem then continue iterating. After we have iterated over all EndPoint objects in *endpoints*, we will have printed all intersections between the chords in C. The pseudocode for this algorithm is given below.

```
PRINT-INTERSECTIONS(C)
 1  endpoints = [ ]
 2  for chord c in C
 3       finish = new EndPoint object with key c_2, class "finish", partnerPointer null,  LLPointer null
 4       start = new EndPoint object with key c_1, class "start", partnerPointer finish,  LLPointer null
 5       endpoints.add(start)
 6       endpoints.add(finish)
 7
 8  endpoints = MERGE-SORT(endpoints)
 9  currentCollection = new DoublyLinkedList
10  chordCount = 0
11
12  for endpoint e in endpoints
13       if e.classification = "start"
14            z = new LinkedList node with key: e.key and chordNumber: chordCount
15            currentCollection.add(z)
16            e.partnerPointer.LLPointer = z
17            chordCount = chordCount + 1
18       else
19            current = e.LLPointer
20            while current.next ≠ null
21                 PRINT(current.chordNumber, current.next.chordNumber)
22                 current = current.next
23            currentCollection.remove(e.LLPointer)
```

Let us now prove this algorithm has a run-time of $O(n \lg n + m)$ where m is the number of intersections. At lines 2-6 we iterate through all n chords in C, performing constant time operations on each iteration. This yields $\Theta(n)$. The call to MERGE-SORT at line 8 takes time $\Theta(n \lg n)$ just as in part A. Then at line 12, we iterate over each of the $2n$ EndPoint objects. For the control structure at lines 13 - 17 (i.e. e is a start endpoint), we perform constant time operations. For the else case at lines 18-23, we first perform a constant time operation of setting *current*, then we iterate over each intersection associated with e. Since there are m intersections in total and we remove $e.LLPointer$

from the Linked List at line 23 after iterating over all intersections associated with e, we never iterate again over a given intersection and we never iterate over a node that is not a part of an intersection. Therefore, each iteration of the inner loop over intersections at lines 20-22 iterates over a distinct fraction of the m intersections. So, over all iterations of the outer loop, we iterate exactly once over each intersection. This yields $\Theta(n)$ for the outer loop and $\Theta(m)$ for **all** iterations over intersections at lines 20-22. We therefore have $\Theta(n) + \Theta(n \lg n) + \Theta(n) + \Theta(m) = O(n \lg n + m)$.

Problem 8

Let us define the input set $\{x_1, x_2, ..., x_n\}$ of real numbers as S. Since S is given as a set, it has no implicit order. Therefore, our first operation will be to sort S and put its sorted contents into the array A. We can do so in $\Theta(n \lg n)$ time using MERGE-SORT. Then, A holds n real numbers in sorted order.

To begin, let us first define the optimal solution for this problem and use it to identify a dynamic programming solution. An optimal solution is the minimum number of unit-length closed intervals required to hold all given elements in A. Then, let us define the optimal substructure for some element x_j in this problem as the minimum number of total intervals used to store elements $x_1, ...x_{j-1}$ in A. Let us note however, that not just elements $x_1, ...x_{j-1}$ have the possibility to yield an optimal solution. All values within a unit-length range before x_j could fit into a single interval. Therefore, the optimal substructure for this problem will be the optimal number of intervals required to store elements $x_1, ..., x_i$ for all i such that $x_i \in A$ and $x_j - 1 \leq x_i < x_j$.

Therefore, a dynamic programming solution would be as follows. For each element $x_j \in A$, compute the minimum number of intervals required to store elements $1...x_j$ by trying all possible elements in A within a unit-length range to the left of x_j. For each of these x_i, the number of intervals required to store $1...x_j$ is the optimal number of intervals required to store $1...x_{i-1}$ plus 1 since all elements between x_i and x_j fit into a single unit-length interval. We will compute this value for all possible x_i. This will require iterating through a portion of A so we only iterate through observed values. Then the optimal solution for elements $1...x_j$ is the minimum number of intervals required among all possible x_i. If no possible x_i exist, then we put x_j into an empty interval. The number of required intervals will then be the optimal solution for elements $1...x_{j-1}$ plus 1.

We will then store the optimal number of intervals required for each input as well as the choice of i that yielded that optimal result in two tables, then proceed to element x_{j+1}. Once we've computed the case in which $x_j = x_n$ (i.e. we've made it through the entire array), the optimal number of intervals required to store all elements in A is the number of intervals required for the best choice of x_i on the last iteration of the loop. Since we stored the best choice of i for each chosen interval, we can then walk back through the table of intervals and output the optimal choices. For example, if at index n in A we found the best choice of i was $n - 2$, then one interval in our optimal solution contains elements x_{n-2}, x_{n-1}, x_n. Then, we proceed to index $n - 3$ in the table of chosen intervals and grab the interval used there. We repeat this process until we hit the first element in A.

Let us observe, however, that multiple optimal solutions may exist for a given input. Specifically, in searching through all choices of x_i for a given element x_j, the choice of x_i that is furthest from x_j will always yield an optimal solution. Let us define this x_i specifically as the choice of i such that $x_j - x_i|$ is maximized. Let a denote this first possible choice of i such that x_a is the first possible x_i for a given element x_j. We can prove x_a will yield an optimal solution for each x_j with the following. Let us observe that all choices of x_i fit into a unit-length interval for some x_j. Therefore, x_a maximizes the number of elements that fit into the interval from x_a to x_j. This gives us an optimal solution for the values x_a through x_j.

Then, there exist two cases for the elements to the left of x_a: either one or more elements are within a unit-length interval to the left of x_a, or they are not. If no elements to the left of x_a are within

a unit-length interval of x_a, then the optimal solution to x_a is the optimal solution to x_{a-1} plus 1 since we then would put x_a into its own interval. Given this, since we can pack all elements in the range x_a to x_j into a single unit-length interval, we then have an optimal solution since no elements to the left of x_a were comprimised in constructing an interval from x_a to x_j.

If there are elements to the left of x_a that are within a unit-length interval of x_a, then let us denote the furthest element to the left of x_a in a unit-length interval as x_b. In this case, an optimal solution for the elements in the range x_b to x_j is two intervals. We know it cannot be 1 interval because x_b was not included in the choice of values for x_i for x_j since $x_a \neq x_b$. Then let us observe the following, we could pack elements x_b through x_{a-1} into a single interval then put x_a to x_j into another and yield a solution of two required intervals. In this case, we know all elements x_b through x_{a-1} fit into a single interval because x_b is within a unit-length range of x_a and all elements in x_{b+1} to x_{a-1} are greater than x_b. We also know all elements x_a through x_j fit into a single interval as shown above. Another option would be to pack all elements x_b through x_a into a single interval and all elements x_{a+1} through x_j into another. Similar to above, this results in 2 required intervals since each choice constitutes a valid unit-length range. Since either of these options yields an optimal solution and all elements to the left of x_b are not comprimised by our choice of x_a, we then know an optimal solution to x_j is to use the optimal solution for elements 1...x_{a-1} then pack all elements x_a through x_j into a single interval.

Therefore, since we have shown an optimal solution always arises when we choose x_a for each x_j, rather than pursuing a dynamic programming solution, we can instead use a greedy solution. Specifically, for each element in A, we will fit as many elements as possible into each interval from right to left. To do so, we will iterate through each element in A from back to front. On each iteration, we will store a record of the value of the last element in the interval we are constructing. If the currently observed element fits into this interval, we will stick it in. If it does not, we will then close the previous interval, and stick the currently observed element in a new interval. Doing so abides by the greedy solution shown above. The first element in each interval is the value x_a for the last element in the interval. After we close a given interval, we then simply construct the optimal solution to all elements to the left of x_a. This is a subproblem of the larger problem that has its own greedy solution. Since we proved above that the greedy choice is always safe to make, our final solution will threfore be optimal. Since we iterate from back to front in the array, we ensure all elements in A exist within an interval and that the final solution is optimal.

```
COMPUTE-INTERVALS(S)
 1  A = MERGE-SORT(S)
 2  intervals = [ ]
 3  curr_interval = [null, A[n]]
 4  for i = n down to 1
 5        if A[i] < curr_interval[1] - 1
 6              curr_interval[0] = A[i + 1]
 7              intervals.add(curr_interval)
 8              curr_interval = [null, A[i]]
 9  curr_interval[0] = A[0]
10  intervals.add(curr_interval)
11  return intervals
```

Let us argue the correctness of this algorithm with the following. At line 4 we loop through each element in A from back to front. On each iteration, if we have found that the current element does not fit in the interval we are constructing, then x_a is the last element we iterated over. So, we grab that element to complete the previous interval, then construct a new interval from the current element. This abides by the greedy solution proven above. We continue doing this until all elements but the last have been found in A. Then, we complete the current interval by filling in the first element x_0. We then return the array of intervals.

The run-time of this algorithm is comprised of two pieces. The first is the call to MERGE-SORT that runs in $\Theta(n \lg n)$ time. The second is our iteration through each element in A which takes time $\Theta(n)$. Therefore, we have $\Theta(n \lg n) + \Theta(n) = \Theta(n \lg n)$.

Problem 9

Part A: If both the path-compression and union-by-rank heuristics are used, then a sequence of m operations in which all the MAKE-SET operations run first, followed by all the LINK operations, followed by all the FIND-SET operations will run in $\Theta(m)$ time. This occurs because each MAKE-SET and each LINK operation takes constant time. MAKE-SET simply constructs a new node with the given value and LINK simply connects the parent pointers of two disjoint-set forests.

For the FIND-SET operation, we can get an amortized cost of $\Theta(1)$ with the following. Using the path-compression heuristic, the first FIND-SET operation for a given path from a node to the root will set the parent of all nodes on the path to be the root. So, after a node is compressed on a path during an initial FIND-SET operation, it will be at most one node away from the root node for all future calls to FIND-SET in this series of m operations. Therefore, each future FIND-SET operation on a node that was previously compressed will take constant time. Then, we must iterate only over all m nodes in the tree once (perhaps in pieces from different function calls) in order to set each node's parent to the root. After this, each future FIND-SET operation will take constant time since we simply must return a node's parent since that will be the root. This yields a constant time amortized cost, which we can prove with the accounting method.

To do so, we will charge \$2 for each MAKE-SET operation. One will go towards creating a node for the given element. The other will be stored on that node for when we iterate over it on a FIND-SET operation and set its parent to the root. We will charge \$1 for the LINK operation. This will go towards the constant time cost of this operation as outlined above. Similarly, we will charge \$1 for the FIND-SET operation. This will pay for accessing the root from the parent after path compression.

Let us prove we can pay for this series of m operations by charging the amortized costs. Since all MAKE-SET operations come first, each node in the tree will have \$1 sitting on it after all sets have been constructed. This is because of the \$2 allocated to each MAKE-SET operations, \$1 goes towards constructing the node and the other acts as a pre-payment for the path compression. Since all LINK operations follow the MAKE-SET operations and each LINK operation is allocated \$1 for connecting parent pointers, we can pay for each LINK operation with the amortized cost associated with this operation. Then, the only operations left are FIND-SET operations. Since the first call to FIND-SET in which a given element is iterated over will set its parent pointer to the root and we showed above that each node has \$1 sitting on it from the MAKE-SET operation, we can pay for setting the parent pointer of each node to the root with this allocated \$1. Then, since the path compression is paid for by the MAKE-SET operation, we spend the \$1 allocated to the FIND-SET operation when returning the parent pointer on each function call. Therefore, by charging the amortized costs, each operation is paid for.

So, since we pre-pay for the path compression in the initial pass of the FIND-SET operation for each element, and we have shown above that each node is compressed once in a series of m calls, we then have a constant-time amortized cost for each of the three operations outlined above. So, for a sequence of m operations in which all the MAKE-SET operations run first, followed by all the LINK operations, followed by all the FIND-SET operations, we have a run-time of $\Theta(m)$.

Part B: If only path compression is used and not union by rank, then we still have a run time of $\Theta(m)$ for m operations in which all the MAKE-SET operations run first, followed by all the LINK operations, followed by all the FIND-SET operations. This is because the union by rank heuristic only provides increased efficiency for run-time when path compression is not used. Union by rank provides gain in that it adds balance to the tree structure, decreasing the number of elements we must traverse on a given call to FIND-SET. Since we compress each path immediately upon its first visit from FIND-SET, the improved run-time we gained from union by rank is lost when we set the parent of each node to the root because we never traverse each node again and therefore never have the opportunity to reap the benefits of the increased balance. However, this is of no concern as path compression increases efficiency more than union by rank because it allows us to traverse fewer nodes in order to find the representative.

Therefore, if only path compression is used and not union by rank, we still have a run-time of $\Theta(m)$.