Assignment 2 - CS 3340B

Name: Riley Emma Gavigan

Student Number: 251150776

Student ID: rgavigan

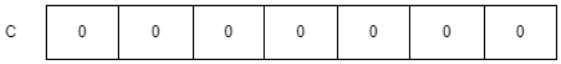
Question 1

8.2-1:

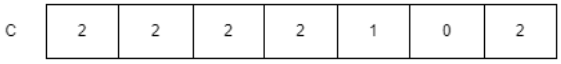
Using Figure 8.2 as a model, illustrate the operation of $COUNTING - SORT$ on the array $A = [6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2]$.

- Each element is a nonnegative integer no larger than $k = 6$.
- n = 11, k = 6

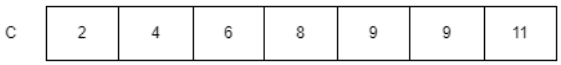
Lines 2-3 [Initialize Array C]



Lines 4-5 [Fill Array C With Equal Elements]

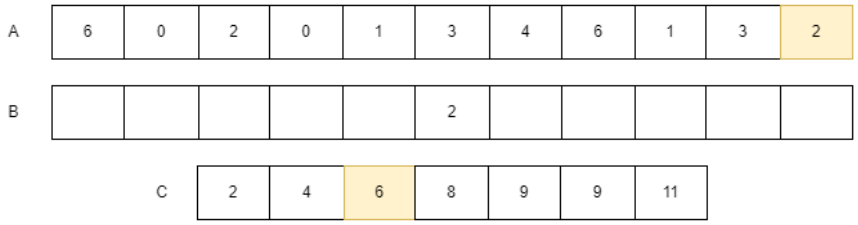


Lines 7-8 [Fill Array C With Less Than O Elements]

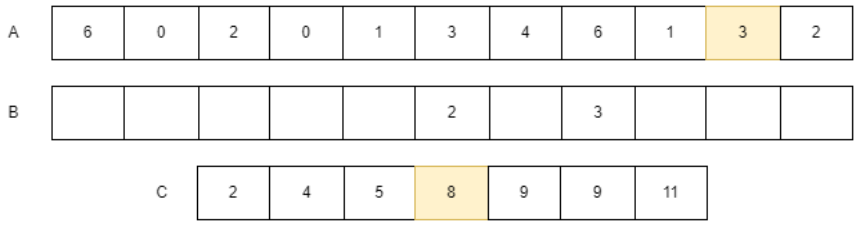


Lines 11-13 [Filling Array B]

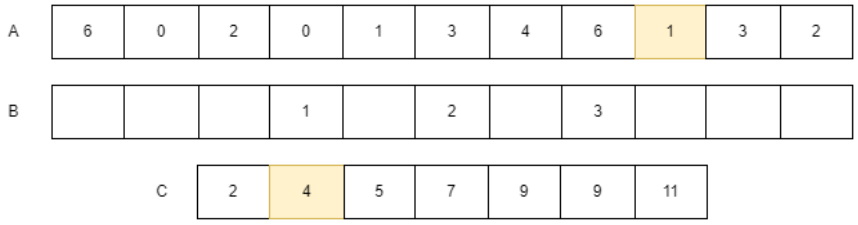
j = 11:



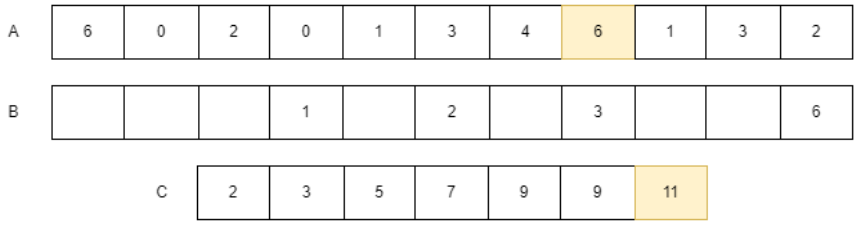
j = 10:


A
6 0 2 0 1 3 4 6 1 3 2
B
2 3
C
2 4 5 8 9 9 11

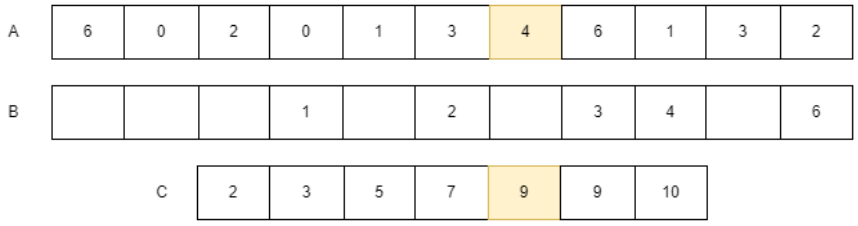
j = 9:


A
6 0 2 0 1 3 4 6 1 3 2
B
1 2 3
C
2 4 5 7 9 9 11

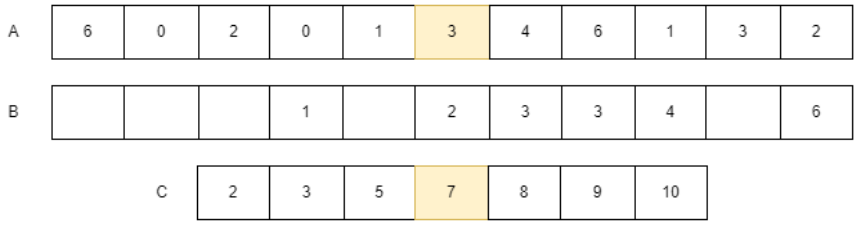
j = 8:


A
6 0 2 0 1 3 4 6 1 3 2
B
1 2 3 6
C
2 3 5 7 9 9 11

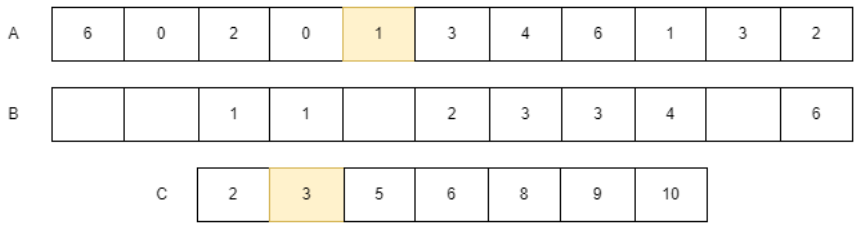
j = 7:


A
6 0 2 0 1 3 4 6 1 3 2
B
1 2 3 4 6
C
2 3 5 7 9 9 10

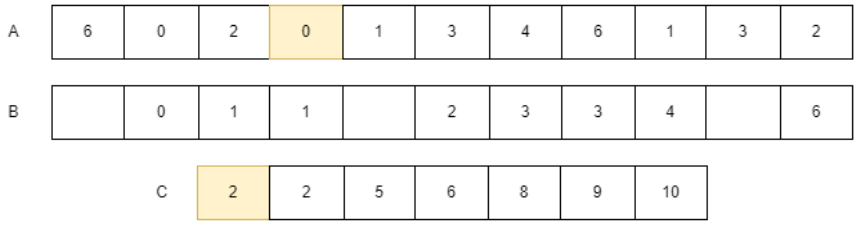
j = 6:


A
6 0 2 0 1 3 4 6 1 3 2
B
1 2 3 3 4 6
C
2 3 5 7 8 9 10

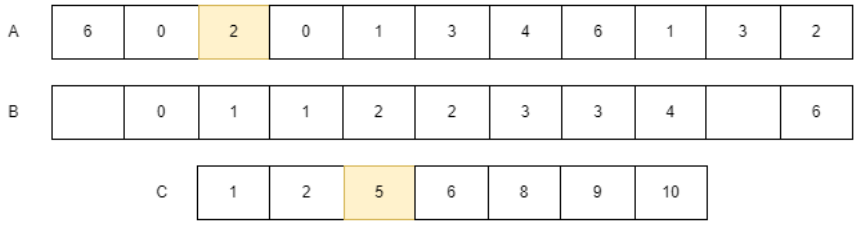
j = 5:


A 6 0 2 0 1 3 4 6 1 3 2
B 1 1 2 3 3 4 6
C 2 3 5 6 8 9 10

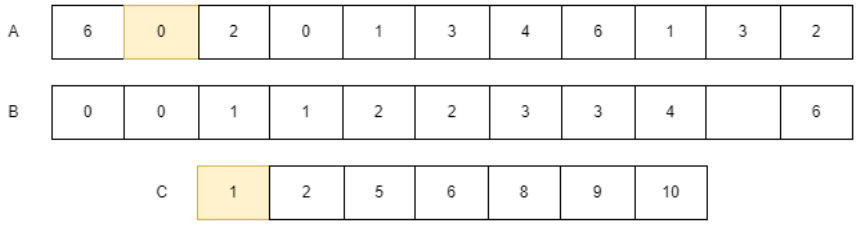
j = 4:


A 6 0 2 0 1 3 4 6 1 3 2
B 0 1 1 2 3 3 4 6
C 2 2 5 6 8 9 10

j = 3:


A 6 0 2 0 1 3 4 6 1 3 2
B 0 1 1 2 2 3 3 4 6
C 1 2 5 6 8 9 10

j = 2:


A 6 0 2 0 1 3 4 6 1 3 2
B 0 0 1 1 2 2 3 3 4 6
C 1 2 5 6 8 9 10

j = 1:

A	6	0	2	0	1	3	4	6	1	3	2
B	0	0	1	1	2	2	3	3	4	6	6
C	0	2	5	6	8	9	10				

Final State: Return B will return the B array, highlighted in pink.

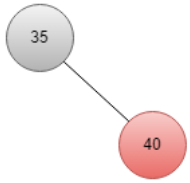
A	6	0	2	0	1	3	4	6	1	3	2
B	0	0	1	1	2	2	3	3	4	6	6
C	0	2	5	6	8	9	9				

Question 2

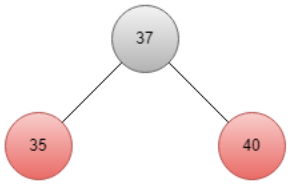
Show the red-black trees that result after successively inserting the keys in the order 35, 40, 37, 19, 12, 8 into an initially empty red-black tree.

Insert 35:

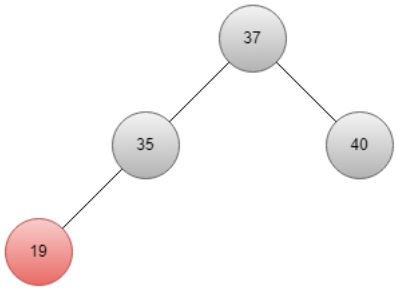
Insert 40:



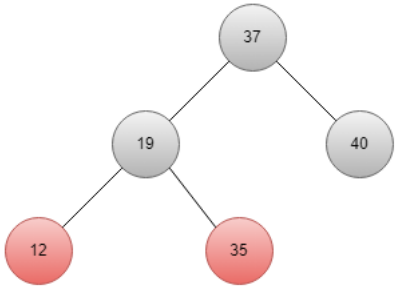
Insert 37:



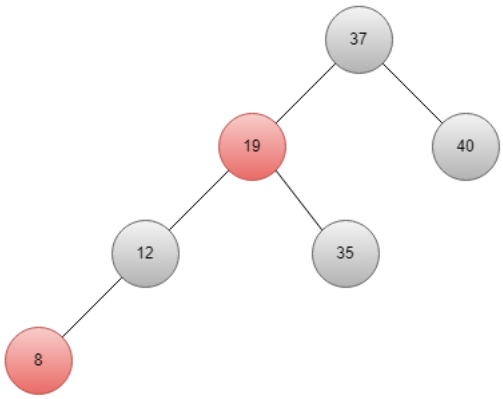
Insert 19:



Insert 12:



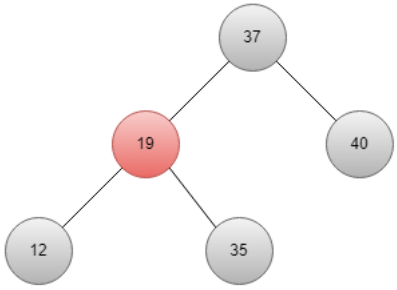
Insert 8:



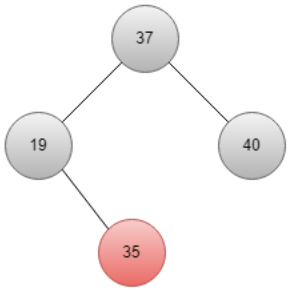
Question 3

Show the red-black trees that result after the successively deletion of the keys in the order 8, 12, 19, 37, 40, 35 from the final red-black tree of question 2.

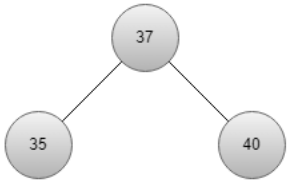
Delete 8:



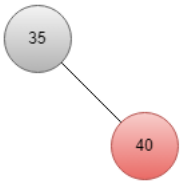
Delete 12:



Delete 19:



Delete 37:



Delete 40:

Delete 35:

Question 4

Given n elements and an integer k. Design an algorithm to output a sorted sequence of smallest k elements with time complexity $O(n)$ when $klog(n) \leq n$.

- Ideas
  - Min Heap [n + klogn time complexity possible, klogn ≤ n. Therefore, O(n)]

Design:

1. **Describe your algorithm in English (not with pseudo code);**
   - The algorithm will begin by **putting the n elements into a minimum heap**. This will result in a min heap, where each node has a value smaller than the values of its descendants.
   - After the min-heap is built, the algorithm will loop k times, **popping the root** of the min-heap in each iteration. After popping the root, the **min-heap will be processed** to ensure the min-heap property remains.
   - The popped value of the root in each iteration of the loop will **append this value to the end of the sorted sequence**, which begins out as an empty sequence. Once the value of the loop counter is greater than k, the loop will stop and the sorted sequence of the smallest k elements will be returned.
2. **Show why the algorithm is correct**
   - The algorithm begins by creating a min-heap with the **heapification** operation, resulting in a minimum heap. The main property of the minimum heap is that the root is the minimum element, and that every element is smaller than its child elements.
   - An **empty sequence** of size k to hold the result will be **initialized** before looping too.
   - The loop will then **loop through k times** using a counter until the looping variable is greater than k.
   - **Each loop iteration:**
     - **Pop the value** from the root of the heap (the minimum element out of all the elements currently in the min heap).
     - **Put this value at the end of the sorted sequence**. Because any values that are already in the sequence are smaller than the value that was popped because of the property of a min-heap, the sorting property will remain.
       - If it is empty: add the smallest element of all n elements to sorted sequence
       - If it is not empty: add the next smallest element of all n elements to sorted sequence
     - Perform an **update on the min-heap to maintain the min-heap property**. This means that the min heap with n - 1 elements, where n is the number of elements before popping in the loop iteration, will be a min heap. Meaning that the root value will be the smallest value in the remaining min heap of n-1 values.
   - After k loop iterations, k values will have been added into the sorted sequence, and n - k values will remain in a min heap that still holds as a min heap because of the update operation. Therefore, **this algorithm is correct because it ensures that only k iterations of popping the min element remaining in the min heap will occur and add the value to the sorted sequence AFTER the values smaller than it that are already in the sorted sequence**.
3. **Analyse the complexity of the algorithm.**
   - The initial heapification of the n elements takes O(n) time. This will only need to be performed once to initially create a min-heap before the loop.
   - An empty sequence of size k to hold the result will be initialized, which takes O(1) time.
   - The loop will loop k times, and in each iteration the following will occur:
     - The minimum value is extracted, Extract_Minimum, which takes logn time

    - The min heap is updated to maintain the minimum heap property, which takes logn time
- After the loop, it will simply return the resulting sorted sequence
- Time Complexity:
    - O(n) [initial heapification] + O(k * 2log(n)] = O(n + k2log(n)) = **O(klog(n))**
- **When klog(n) ≤ n, the algorithm is of time complexity O(n)** because O(klog(n)) will be ≤ O(n), and the time complexity was found to be O(klog(n)).

Question 5

Design an efficient data structure using (modified) red-black trees that supports the following operations:

- $Insert(x)$: insert the key x into the data structure if it is not already there
    - The insert operation can be a standard binary search tree insertion that is followed by the standard red-black tree structure changes when necessary (insertion cases for colouring / rotations). This way, the red-black tree properties hold after insertion.
    - Furthermore, the sub-tree sizes of all the nodes on the path from the root node to the newly inserted node in the tree must be updated, as we need to store the sub-tree sizes of all nodes. If we are inserting the first element into the tree, we simply initialize a sub-tree size of 0. If we are inserting elements into a tree that already has elements, we increment the sub-tree sizes on the path to the node being inserted.
- $Delete(x)$: delete the key x from the data structure if it is there.
    - The deletion operation can be a standard binary search tree deletion that is followed by the standard red-black tree structure changes when necessary (deletion cases for colouring / rotations).
    - Furthermore, the sub-tree sizes of all the nodes on the path from the root node to the deleted node must be updated by being decremented, in order to maintain the correct sub-tree sizes of all nodes.
- $Find\_Smallest(k)$: find the k^{th} smallest key in the data structure.
    - The find_smallest opreation can find the k'th smallest key in the tree by starting from the root node and using a variable to keep track of the rank.
    - The rank will be a variable assigned according to an inorder tree traversal (traverse left, visit root, traverse right). It will store the position of the node visited in the inorder traversal as the tree is traversed.
    - We can get the value of the rank variable for a node from its inorder traversal by the following: current.left.getSubTreeSize() + 1. If there is no left node, let the value be 0 + 1.
    - There are two cases when visiting a node and checking its rank:
        - If the rank of the current node being visited is less than k, move to the right subtree.
        - If the rank of the current node being visited is greater than or equal to k, move to the left subtree.
    - When the algorithm moves to a left subtree, the value of k is updated by subtracting it by the size of the current node's right subtree (k - current.right.getSubTreeSize(), for example.).
    - When the algorithm moves to a right subtree, the value of k is updated by adding it by the size of the current node's left subtree (k + current.left.getSubTreeSize(), for example.).
    - After iterating through the algorithm enough, k will eventually reach the value 1. When this value of k is reached, the k'th smallest node has been found and should be returned. This is the terminating condition of the algorithm.

What are the time complexities of these operations?

- $Insert(x)$

- The time complexity of the insertion operation is $O(logn)$, where n is the number of nodes in the modified red-black tree.
- This time complexity occurs because the insertion operation in a standard red-black tree is $O(logn)$, and this insert operation is the exact same, except for the fact that another constant operation is introduced on the nodes being traversed to update the sub-tree sizes for the nodes in the modified red-black tree.

- $Delete(x)$
  - The time complexity of the deletion operation is $O(logn)$, where n is the number of nodes in the modified red-black tree.
  - This time complexity occurs because the deletion operation in a standard red-black tree is $O(logn)$, and this deletion operation is the exact same, except for the fact that another constant operation is introduced on the nodes being traversed to update the sub-tree sizes for the nodes in the modified red-black tree.
- $Find\_Smallest(k)$
  - The time complexity of the find_smallest operation is $O(logn)$, where n is the number of nodes in the modified red-black tree.
  - This time complexity occurs because on the path from the root node to the k'th smallest node, at most $O(logn)$ nodes are visited. Furthermore, each visit of a single node takes constant $((O(1))$ time. Because of the use of sub-tree sizes being stored for all of the nodes in the tree, the traversal of the tree is made more efficient than otherwise because the find_smallest operation can easily decide which sub-tree to visit from the current node based on the sub-tree size.

Question 6

19.4-2: **Prove that every node has rank at most $\lfloor lg\ n \rfloor$.**

*(Hint: use induction on the sequence of operations, check the proof of Lemma (1) in the notes, **not** the number of elements, of a disjoint set)*

Proof

- Prove by strong inducion on the Make_set and Union operations.
  - Base case: The first operation has to be Make_set, and because we have one node that has height 0 and rank 0, the node has rank $\lfloor lg(1) \rfloor = 0$. Thus, the base case holds.
  - Inductive step: Consider a Union(i, j) operation, letting r_i and r_j be the roots of the trees containing x_i and x_j. We assume that the rank of the tree r_i is $\lfloor lgr_i \rfloor$ and the rank of the tree r_j is $\lfloor lgr_j \rfloor$ by the inductive hypothesis. We know that $n = n(r_i) + n(r_j)$
    - If the ranks of the two trees r_i and r_j are not equal $(rank(r_i) \neq rank(r_j))$:
      - The union operation preserves rank, so if $rank(r_i) > rank(r_j)$, the resulting rank is $rank(r_i)$ which we know is less than $\lfloor lgn \rfloor$ because $n(r_i) < n$, $\lfloor lgr_i \rfloor < \lfloor lgn \rfloor$. Furthermore, if $rank(r_j) > rank(r_i)$, the resulting rank is $rank(r_j)$, which we know is less than $\lfloor lgn \rfloor$ because $n(r_j) < n$, $\lfloor lgr_i \rfloor < \lfloor lgn \rfloor$.
    - If the ranks of the two trees r_i and r_j are equal $(rank(r_i) = rank(r_j))$:
      - The rank of the union will increase by 1, and so the resulting set rank is $\lfloor lgr_i \rfloor + 1 \leq \lfloor lg(n)/2 \rfloor + 1 = \lfloor lg(n) \rfloor$.
  - Therefore, when a union operation is performed between two trees we get that the rank is at most $\lfloor lgn \rfloor$ and by induction the claim is proven. Q.E.D.

Question 7

19.4-3: **In light of Exercise 19.4-2, how many bits are necessary to store $x.rank$ for each node x? Based on your answer, for question 9 a), is it enough to use one byte to store $rank$? Explain your answer.**

- To store $x.rank$ for each node x, because every node has rank of at most $\lfloor lgn \rfloor$, the number of bits that are necessary to store it is $\lfloor lg \lfloor lgn \rfloor \rfloor + 1$. This is equivalent to $\lfloor lglgn \rfloor + 1$.
- Therefore, $\lfloor lglgn \rfloor + 1$ bits are necessary to store $x.rank$ for each node x

For Question 9a:

- 1 byte = 8 bits being used to store $rank$
- Problem: Find out if 8 bits are enough to store $rank$
  - $8 = \lfloor lglgn \rfloor + 1$
  - $7 = \lfloor lglgn \rfloor$
  - $2^7 = \lfloor lgn \rfloor$
  - $2^{128} = \lfloor n \rfloor$
  - $n = 3.403 * 10^{28}$
- Therefore, up to $\sim 3.403 * 10^{28}$ values can be represented using 8 bits to store $rank$.
- Because this large number of values can be represented using one byte to store $rank$, which is a value larger than most computers can handle, it is enough to use one byte to store $rank$ in question 9a.

Question 8

15.3-5: **Given an optimal prefix-free code on a set C of n characters, you wish to transmit the code itself using as few bits as possible. Show how to represent any optimal prefix-free code on C using only $2n - 1 + n\lceil lg\ n \rceil$ bits. (Hint: Use $2n - 1$ bits to specify the structure of the tree, as discovered by a walk of the tree.)**

- If we use a full binary tree that has n leaves, there are $n - 1$ internal nodes in the tree. This results in a total of $2n - 1$ nodes in the full binary tree.
- The structure of the stree can be specified using a pre-order traversal (visit the root, traverse left, traverse right). You can use a 0 to represent that a node is an internal node, and a 1 to represent that the node is a leaf node.
- Because the code is on a set C of n characters, $\lceil lgn \rceil$ bits are required in order to represent each character in the set C.
- This way, if each leaf node in the full binary tree represents a character in the set C, because there are n leaves, $n\lceil lgn \rceil$ bits will represent all of the n characters in set C.
- The characters can be stored in the same order obtained by the pre-order traversal as it visits the leaves. This results in a total number of bits of $2n - 1 + n\lceil lgn \rceil$, because each internal node also is represented by a bit. Because we only need two values (0/1), it is only 1 bit per node.

Question 9

a) A Disjoint-Set data structure should be implemented, with the most efficient algorithm (union by rank and path compression), as an abstract data type (a class in C++ or java) with the following operations: $uandf(n),\ make\_set(i),\ find\_set(i),\ union\_sets(i, j),\ final\_sets()$

Algorithm of final_sets():

- The final sets method works by finalizing the current sets, meaning make_set and union_sets will no longer have any effect. To do this, three loops will be performed on the UnionFind data structure.
- First, the parent list will be looped through and if the parent of an element is not 0 (its a valid element), findSet is performed. This will ensure that the parent of i is set to i for all elements i that are not 0.
- Next, a value representing the number of sets will be instantiated to 1 representing 1 set.

- The second loop through the parent list will check if the parent of i is equal to i (which it should be for all non-zero elements after loop 1), and if it is it will increment the number of sets by 1 and set parent[i] to the size and the rank to 1. If it is 0 the rank is simply set to 0 for that element. This loop is what ensures the representatives of the sets is correctly set from 1 to size.
- The third loop through the parent list will check if the rank at i is 0 and the parent is non-zero. In this case, it handles this by setting the parent of i to its grandparent.
- Finally, a boolean indicator representing final_set is set to true and that will ensure that make_set and union_sets does not do anything as the sets are finalized. size - 1 will be returned at the end, returning the number of sets.

Correctness and Complexity of final_sets():

- Correctness
  - The algorithm correctly returns the number of sets that are finalized through the use of the counter that is incremented in the second loop.
  - The algorithm correctly ensures that make_set and union_sets do not have any effect after final_set by the boolean, which is tested against in those methods to ensure nothing happens when final_set is set to true.
  - The first loop correctly works by setting all elements with a non-zero parent to the representative element of that set.
  - The second loop correctly works by changing the size of the number of sets each time the parent of i is equal to i, as the first loop correctly handles this change for all elements in a set. It correctly does not count a set twice because it changes the value of the parent to the size afterward. By doing this, it ensures that the integers from 1 to size are used as representatives.
  - The third loop correctly works by changing values with a rank of 0 and a non-zero parent, it ensures that their parent is set to a representative correctly.
- Complexity
  - The first loop iterates n times, where n is the length of the parents list (number of elements in the union data structure). It calls find for each iteration, which takes $O(logn)$ time in the rank and path compression structure. Thus, this loop takes $O(nlogn)$ time.
  - The second loop iterates n times, only performing $O(1)$ operations in each iteration and is thus $O(n)$ time.
  - The third loop iterates n times, only performing $O(1)$ operations in each iteration and is thus $O(n)$ time
  - Therefore, the time complexity is $O(nlogn + n + n) = O(nlogn)$

```
/*
 * Name: Riley Emma Gavigan
 * Student #: 251150776
 * Student ID: rgavigan
 * CS 3340B - Assignment 2
 * Disjoint ADT Structure Class
 */

// Class to represent a disjoint set data structure
public class uandf {

    // Initialize private variables
    private int[] parent;
    private int[] rank;
    private int n; // number of elements
    private boolean finalSet = false; // initially not in final set

    /*
     * Constructor for the disjoint set data structure
     * Creates UnionFind structure with n elements
     * @param n - number of elements
     * @return - None
     */
    public uandf(int n) {
        this.n = n;
        // Initialize parent and rank to n + 1 (1 -> n is 1 based indexing, want to ignore 0)
        this.parent = new int[n + 1];
```

```
        this.rank = new int[n + 1];
    }

    /*
     * make set method that creates a new set with only element i (rank is 0)
     * @param i - element
     * @return - None
     */
    public void makeSet(int i) {
        parent[i] = i;
        rank[i] = 0;
    }

    /*
     * find set method that returns the representative of the set that i is an element of
     * @param i - element
     * @return - representative of the set that i is an element of
     */
    public int findSet(int i) {
        // Not in final set
        if (finalSet == false) {
            // If i is not the parent of the set
            if (parent[i] != i) {
                // Recursively call findSet on the parent of i
                return (parent[i] = findSet(parent[i]));
            }
            // If i is the parent of the set
            else {
                return i;
            }
        }
        // In the final set -> just return the parent of i
        else {
            return parent[i];
        }
    }

    /*
     * union set method that unites the sets i and j belong to into a single union set they belong to
     * @param i - first element
     * @param j - second element
     * @return - None
     */
    public void unionSets(int i, int j) {
        // Find the root elements (the set) that i and j belong to
        int iRoot = findSet(i);
        int jRoot = findSet(j);

        // If they are already in the same set
        if (iRoot == jRoot) {
            return;
        }

        // Union by rank: attach the smaller rank tree (i) to the root of the larger rank tree (j)
        if (rank[iRoot] < rank[jRoot]) {
            parent[iRoot] = jRoot;
        }
        // Union by rank: attach the smaller rank tree (j) to the root of the larger rank tree (i)
        else if (rank[iRoot] > rank[jRoot]) {
            parent[jRoot] = iRoot;
        }
        // Union by rank: they are the same size so attach j to i and increment i's rank
        else {
            parent[jRoot] = iRoot;
            rank[iRoot]++;
        }
    }

    /*
     * final sets method that returns the total # of current sets, size, and finalizes the current sets
     */
    public int finalSets() {
        // Loop through the parent list (1st time)
        for (int i = 1; i < parent.length - 1; i++) {
            // If the parent of i is not 0
            if (parent[i] != 0) {
                // Find the set that i belongs to
                findSet(i);
            }
        }
```

```
        // Loop through the parent list (2nd time)
        int size = 1;
        for (int i = 1; i < parent.length - 1; i++) {
            // If the parent of i is the same as i
            if (parent[i] == i) {
                // Increment size and set parent of i to size
                size++;
                parent[i] = size;

                // Change the rank of i
                rank[i] = 1;
            }
            // If the parent of i is not the same as i
            else {
                rank[i] = 0;
            }
        }

        // Loop through the parent list (3rd time)
        for (int i = 1; i < parent.length - 1; i++) {
            // If the rank at i is 0 and the parent is not 0
            if (rank[i] == 0 && parent[i] > 0) {
                // The parent at i is the parent of the parent at i
                parent[i] = parent[parent[i]];
            }
        }

        // Set finalSet to true and return cur - 1
        finalSet = true;
        return size - 1;
    }
}
```

b) Design and implement an algorithm to find the connected components in a binary image using Disjoint-Set data structure in a).

Design:

1. Describe your algorithm in English (not with pseudo code);

- The algorithm to find the connected components in a binary image using the Disjoint-Set from part a) is as follows:
    - The binary image file is read and the number of rows and columns in the image are provided / gotten. After this, one-dimensional arrays for the sets and characters and a two-dimensional array representing the image are created.
    - The image array is initialized according to the number of rows and columns in the binary image. The algorithm then reads each line of the binary image and, if the symbol is a “+” symbol or whatever other symbol represents area being taken up as space, the image array at them coordinates is set to 1 and the boolean flag to true, representing the image pixel existing. Otherwise, nothing happens as it is an empty space.
    - After reading the image file, the Disjoint Set is used and initialized for all the elements in the file (rows by columns). The dimensions of the image get looped through again, this time checking if the image array has a 1 in the position being checked. If it does, make_set is called to create a new set with only i as its member. Then, the spaces to the left and to the top of the current space are checked, and if the elements are also 1, the sets are unioned using union_sets from the Disjoint Set ADT.
    - After looping, the sets array is initialized by the final_sets of the Disjoint Set ADT and chars is initialized to the size of the sets array, representing the connected components with different characters.
    - To find the connected components, the dimensions of the image are looped through again, and find_set is called to find the set for the position of the current pixel in the image, and a letter is assigned to the component to separate the connected components by unique letter identifiers.

2. Show why the algorithm is correct

    - The algorithm begins by doing parsing of the binary image that is being passed to the algorithm. This simply results in an array that represents the image, where if the image is not a space at a certain position, those coordinates are equal to 1.
    - After the parsing, the UnionFind data structure is initialized for all elements in the image array when they are not empty space, and union operations are performed for the upper and left element of the current one. Because the loops will go row-

by-row, from the top left to the bottom right, there is no need to check against the right and bottom side as they will be handled when those positions are reached. Therefore, the union find initializing loop is correct because it will result in all of the sets being created for the image file, where each set represents a unique component of connected '1' spaces in the image array (non-space elements).

- After the UnionFind data structure is initialized and union operations are performed, we know that all union find parsing has been completed. Thus, final_sets is then called and the number of sets is used to create the two one-dimensional arrays representing the sets and the characters representing the sets. The sets are all also finalized at this point because final_sets handles that.
- After these arrays are intialized, looping again through and using find_set results in the connected components (each set) being assigned a unique letter. Therefore, after this stage it is guaranteed that all connected components in a binary image will have been found. This specific implementation finds the connected components in 4-way directions (left, up, right, down), not counting diagonals as connected.

3. Analyse the complexity of the algorithm.
   - Reading the input file is $O(n * m)$, where n is the number of rows and m is the number of columns in the file.
   - Creating the union find and connected components of the file is $O(n * m * logn)$, because n*m represents the number of rows and columns, and logn represents the union method time complexity, which could theoretically occur proportional to n*m times if every character of the file is not empty.
   - From a), finalSet is called and is $O(nlogn)$ and is called only once.
   - Part 2 takes $O(n * m * logn)$, just like creating the union find part, as findSet is called on each iteration and takes $O(logn)$ time.
   - Sorting the sets is $O(logp)$ time and only occurs once, where p is the number of sets resulting from findSet.
   - The third part is $O(p^2)$, and is not a significant amount of the time.
   - Both the fourth and fifth part are the same time complexity as part 1 and 2, as they simply loop over both rows and columns and perform findSet, which is logn on each iteration of the loop.
   - Therefore, the total time complexity is $O(n * m * logn)$, which is the most significant portion of time. Because we are adding each part that is looping, we can remove smaller terms and also only need to count one loop, not all 4 that are $O(nmlogn)$.

Implemented:

```
/*
 * Name: Riley Emma Gavigan
 * Student #: 251150776
 * Student ID: rgavigan
 * CS 3340B - Assignment 2
 * Binary Image Algorithm (9b)
 */

import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Arrays;

 public class BinaryImageAlgo{

    public static void main(String[] args){
        // Initialize variables
        int[][] image = null;
        int[] sets = null;
        int[] chars = null;

        // Image dimensions
        int numCols = 72; // INSERT NUMBER OF COLUMNS HERE
        int numRows = 75; // INSERT NUMBER OF ROWS HERE

        InputStream inputStream = System.in;
        inputStream.mark(0);
```

```
        try {
            // Get the file from input
            BufferedReader input = new BufferedReader(new InputStreamReader(inputStream));

            // Set up image and flag arrays for 72x75 image girl.img.txt (72 columns, 75 rows)
            image = new int[numRows + 1][numCols + 1];

            // Initialize a string to read each line
            String string;

            // Read through each line; for each + set image at them coords to 1 and flag to true
            for(int i = 0; input.ready(); i++){
                // Read line-by-line
                string = input.readLine();
                for(int j = 0; j < numCols; j++){
                    // If it is a plus, set to 1 and true
                    if(string.charAt(j) == '+'){
                        image[i][j] = 1;
                    }
                }
            }

            // Close the BufferedReader
            input.close();
        }
        // Catch FileNotFound / IO exception
        catch (Exception e){
        e.printStackTrace();
        }

        /*
        * PART 1 PRINTING - the input binary image
        */
        System.out.println("--------PART 1--------\n");
        for(int i = 0; i < numRows; i++){
            // Print each line
            for(int j = 0; j < numCols; j++){
                System.out.print(image[i][j]);
            }
            // New line for each line being finished
            System.out.println();
        }
        // Print a newline at the end of part 1
        System.out.println("\n\n\n");


        // Create the UnionFind and connect components
        uandf imageSet = new uandf(numRows * numCols + 1);
        // Loop through the number of rows
        for(int i = 0; i < numRows; i++){
            // Loop through the number of columns
            for(int j = 0; j < numCols; j++){
                // If the image is 1, make a set and union with the left and top if they are 1 (can ignore other directions since we go fro
                if (image[i][j] == 1){
                    // Want to get the current index (row # * number of columns) + column in row + 1 (1-based indexing)
                    imageSet.makeSet((i * numCols + j) + 1);
                    // If the element to the left is 1 (means it was a +)
                    if (j > 0 && image[i][j - 1] == 1){
                        // Union with the left
                        imageSet.unionSets(i * numCols + (j - 1) + 1, i * numCols + j + 1);
                    }
                    // If the element above is 1
                    if (i > 0 && image[i - 1][j] == 1){
                        // Union with the top
                        imageSet.unionSets((i - 1) * numCols + j + 1, i * numCols + j + 1);
                    }
                }
            }
        }

        // Initialize the sets and chars arrays for part 3 based on final sets
        sets = new int[imageSet.finalSets() + 1];
        chars = new int[sets.length];

        /*
        * PART 2 PRINTING - connected component image with a-z representing the sets
        */
        System.out.println("--------PART 2--------\n");
        int k;
        // Loop through the number of rows
        for(int i = 0; i < numRows; i++){
```

```
            // Loop through the number of columns
            for(int j = 0; j < numCols; j++){
                // Assign a character to k at the current index of the image
                k = imageSet.findSet(i * numCols + j + 1) + 96;

                // If k is 96, print space
                if (k == 96){
                    System.out.print(' ');
                }
                // If k is not 96, print the character and increment the set
                else{
                    System.out.print((char)k);
                    sets[k - 97]++;
                }
            }
            // Print a newline after each line has been processed
            System.out.println();
        }
        // Print a newline after finishing part 2
        System.out.println();

        // Sort the lists for part 3
        int[] sortedSets = new int[sets.length];

        // Copying sets into sorted sets
        for(int i = 0; i < sets.length; i++){
            sortedSets[i] = sets[i];
        }
        // Copying sorted sets into chars
        for(int i = 0; i < sortedSets.length; i++){
            chars[i] = sortedSets[i];
        }
        // Sorting sorted sets
        Arrays.sort(sortedSets);

        /*
        * PART 3 PRINTING - list sorted by component size. each line contains the size and label of the component.
        */
        System.out.println("--------PART 3--------\n");
        System.out.println("Label" + "\t " + "Size");
        for(int i = 0; i < sortedSets.length; i++){
            for(int j = 0; j < chars.length; j++){
                if(sortedSets[i] == chars[j]){
                    // Print out the character and it's size
                    System.out.println((char)(j + 97) + "\t " + sortedSets[i]);
                    chars[j] = -1;
                    break;
                }
            }
        }
        System.out.println("\n\n\n");

        /*
        * PART 4 PRINTING - the input binary image with small components removed (size < 2)
        */
        System.out.println("--------PART 4--------\n");
        for (int i = 0; i < numRows; i++) {
            for (int j = 0; j < numCols; j++) {
                // Assign the character to k
                k = imageSet.findSet(i * numCols + j + 1) + 96;

                // If size is less than 2 (1 or less), remove it
                if (k == 96 || sets[k - 97] < 2){
                    System.out.print(' ');
                }

                // Otherwise print out the character
                else
                    System.out.print((char) k);
            }
            System.out.println();
        }
        System.out.println("\n\n\n");

        /*
        * PART 5 PRINTING - the input binary image with components smaller than 12 removed
        */
        System.out.println("--------PART 5--------\n");
        for (int i = 0; i < numRows; i++) {
            for (int j = 0; j < numCols; j++) {
                // Assign the character to k
```

```
                k = imageSet.findSet(i * numCols + j + 1) + 96;

                // If size is less than 12 (11 or less), remove it
                if (k == 96 || sets[k - 97] < 12){
                    System.out.print(' ');
                }

                // Otherwise print out the character
                else
                    System.out.print((char)k);
            }
            System.out.println();
        }
        System.out.println();
    }
}
```