Homework 4 - Solution

R-7.5

A connected component in a network is implemented here using a list-based implementation, say linked-list. The head of the linked list points to the first element, and the first elements point to the second element, second to the third element and so on. So to list this set, we just need to iterate over the linked list by starting with the head pointer. In this way, if the size of the set is n, then listing it takes $O(n)$ time.

C-7.1

Items in a set could themselves be nodes or maintained as some kind of lookup table or map for finding the node associated with an item in constant time. This would result in the constant time $O(1)$ for $find()$ and $makeSet()$ operations.

The only difference between the extended array based implementation and the list based implementation is that with the $union()$ operation. Note that in a list based implementation, each time we update the head pointer for some node, the size of the new set at least doubles. This property is due to the fact that we always link the smaller set into the larger one, with ties broken arbitrarily. Thus, any node can have its head pointer changed at most $log\ n$ times; hence, each such node can be charged at most $O(log\ n)$ cyber-dollars, since we assume that the partition starts with n singleton sets. Thus, the total amount of cyber dollars charged to all the nodes in this implementation of a union-find structure is $O(n\ log\ n)$.

So for an extended array based implementation, we will extend the size of the larger array to double its size, so we can accommodate the items of the smaller set. As per amortized running time, this will take $O(n)$. We will need to carry out $log\ n$ union operations, to turn all the n singleton sets into a connected component. So, the total running time for $union()$ operation will be $O(n\ log\ n)$.

Thus, the extendable array based union-find implementation will take $O(n\ log\ n\ +\ m)$ time for processing sequence of m union-find operations on an initial collection of n singleton sets.

A-7.5

This problem is similar to the maze generation application or percolation theory of the union - find data structures.

We initialize a grid of $n\ X\ n$ cells by calling the *makeSet()* operation creating multiple singleton sets of each cell.

We create a set of cells in the top row and check for a black stone. On finding the first black stone, we check the adjacent cells for other black stones and call the *union()* method to combine

them. We continue to combine adjacent cells containing black stone until we reach a cell in the bottommost row. This denotes a win situation for the player 1.

Next, we create a set of cells in the leftmost column and check for a white stone. On finding the first white stone, we check the adjacent cells for other white stones and call the *union()* method to combine them. We continue to combine adjacent cells containing white stone until we reach a cell in the rightmost row. This denotes a win situation for the player 2.

After the winner is available, we can sum through the values for the path taken by the winning player to calculate bonus points for the gold elements. This will take *O(n)* time.

We will use a tree based implementation of union-find structures to implement this algorithm. In a sequence with *m* union and find operations performed using union-by-size and path compression, starting with a collection of *n* single-element sets, the total time to perform the operations in sequence is $O((n + m)\alpha(n))$.

C-8.5

Start with two pointers: head and tail, head pointing to the first element of an array *S* and tail to the last element. We start a loop until head < tail and check for the head element, if it is pointing to blue element, we start increasing it till we find the red one.

After that, if the tail points to the red one, we start decreasing it till, it points to blue one and we swap the elements. Running time for this will be *O(n)* as in worst case, we need to traverse the entire array.

For three colors (e.g., red, green, and blue), we can use above method to sort the array by any of one color (say, red) by considering all the elements as red and non-red. Then, we apply the same method for ordering the array with remaining colors (green and blue). Running time for this will also be *O(n)* as we are ordering *S* two times using above method.

A-8.2

We can sort the *n*-element sequence *S* based on *ID* of the candidate. We may use merge-sort to achieve this in *O(n log n)* time.

Then, we traverse the above sorted sequence and count total number of votes of current *ID's* and store it. We continue going on next *ID* in the sequence and count corresponding *ID's* votes. Then, we compare it with previous *ID's* count and set max with the larger count's value. This step takes $O(n)$ time for performing count operation for each *ID* and for updating the value of max.

Thus the total time for this approach will be *O(n log n)*.

A-8.3

We know that total $k < n$ candidates are running. So, we can use AVL tree to store *k* candidates as nodes with their *ID* and total count of votes (initializing it with 0). We traverse the given *n*-element sequence *S* of votes, we search the node having the same specific *ID* and increment the corresponding count by 1.

As AVL is a balanced binary search tree, each time for searching and updating the node as well the structure itself, takes $O(log\ k)$ time as there are total *k* nodes in the tree. We need to perform this step *n* times. So, total running time is $O(n\ log\ k)$.

C-9.10

We will use a divide and conquer approach, which is used in binary search. In this approach we compare the middle elements of arrays or lists *A* and *B*, let us call this *midA* and *midB* respectively. We then try to divide the search space of finding *n*th smallest key (median of *A* union *B*) either by dividing *A* or *B* based on the value of *n* and the values of *midA* and *midB*. In this way, we define a new sub problem with half the size of one of the arrays, thus the running time of this algorithm will be *O*(log *n*). The implementation will be the same in case of an array or a list, and the pseudo code can be given as:

Algorithm findNthSmallest(int[] *A*, int[] *B*, int *n*)
Input: The sorted arrays *A* and *B* of size n, and the value of *n*
Output: The *n*th smallest key in the union of the keys from *A* and *B*

```
if (A.length = 0) then
        return B[n]
if (B.length = 0) then
        return A[n]
int midA ← A.length / 2
int midB ← B.length / 2
if (midA + midB < n) then
        if (A[midA] > B[midB]) then
                return findNthSmallest (A, B[midB+1, B.length], n-midB-1)
        else
                return findNthSmallest (A [midA+1, B.length], B, n-midB-1)
else
        if (A [midA] > B[midB]) then
                return findNthSmallest (A [0,midA], B, n)
        else
                return findNthSmallest (A, B[0,midB], n)
```

A-9.4

We are given an array A containing n votes listed in no particular order. The goal is to find student numbers of every candidate that received more than $n/2$ votes in $O(n)$ time.

We can use a linear-time selection algorithm to search for item, x, of rank $n/2$. Note that if a candidate has received over $n/2$ votes then his or her student number must be included in the items at rank $n/2$. So given x, we scan A once more and count how many times x appears in A.