CSC3100 Assignment2 Report

Problem 1:

Create a Player class to store the information of each player.

Besides the three information descriptions provided, addKey is added to record the input sequences of these players so that at the end we can output the hp of each survivor in the exact input sequence.

```
    static class Player{
           int addKey;
           int floor;
           int hp;
           int direction; // UP(1)/DOWN(-1)
           public Player(int addKey,int floor, int hp, int direction) {
               this.addKey = addKey;//input sequence
               this.floor = floor;
               this.hp = hp;
               this.direction = direction;
           }
       }
```

After inputting all of the players into the players list, we need to sort the list based on the floor.

In Java, we can use the sort in the class Collection by overriding the compare method, just as below.

```
    //sort based on floor
           Collections.sort(players, new Comparator<Player>() {
               @Override
               public int compare(Player o1, Player o2) {
                   if(o1.floor < o2.floor){
                       return -1;
                   }else if(o1.floor > o2.floor){
                       return 1;
                   }else{
                       return 0;
                   }
               }
           });
```

After the operation, we get the list of players from the lower floor to the higher floor.

It's meaningless to deal with the floor number. All we need is the sorted players to determine the sequence of battle.

Create a stack to store the players whose direction is going up.

`LinkedList<Player> stack = new LinkedList<>();` using LinkedList as stack since it has the method of pushing and popping.

From the bottom, iterate the players list.

```
    for(Player p: players){
               //if we meet a player whose direction is going up, we push
   the play into the stack
               if(p.direction == UP){
                   stack.push(p);
               }else{
                   //if we meet a player whose direction is going down
                   //1.pop the player in the peek of the stack, battle
                   while(!stack.isEmpty()){
```

```
                    //if the stack is not empty, it means that the pop
player has at least one opponent.
                        Player p1 = stack.pop();
                        //battle rule
                        if(p.hp > p1.hp){
                            p1.hp = 0;
                            p.hp -= 1;
                        }else if(p.hp < p1.hp){
                            p.hp = 0;
                            p1.hp -= 1;
                            stack.push(p1);
                            break;
                        }else{
                            //if have the same hp, set hp of both of them to
0
                            //break
                            p1.hp = p.hp = 0;
                            break;
                        }
                    }
                    //2. if the stack is empty, it means that the current
player does not have an opponent anymore, surviving in the game.
                    if(stack.isEmpty() && p.hp != 0){
                       survivors.add(p);
                    }
                }
            }
            //After the battle game, those who are in the stack survive as
well
            while(!stack.isEmpty()){
                survivors.add(stack.pop());
            }
```

Finally, we sort the survivor's list based on the input sequence(addKey), then output their hp.

Problem 2:

```
private static long MaxArea(int length, int[] depth) {
        //create a stack to store the potential left boundary for the
area
        LinkedList<Integer> stack = new LinkedList<>();
        long maxArea = 0;
        int width,height;
        for(int i = 0; i < length; i++){
            if(stack.isEmpty() || depth[stack.peek()] < depth[i]){
                stack.push(i);
            }else{
//If the i-index is smaller than the top element of the stack, it
indicates that the column where the i-element is located will serve as
the left boundary of the post-order area.
   //At this point, use the top element of the stack as the right
boundary to process the previous area.
                while(!stack.isEmpty() && depth[stack.peek()] >
depth[i]){
                  //depth[stack.peek()] > depth[i] make sure that
                  //we leave those depth smaller than depth of i-index
                  //to be the left boundary of index i or larger than i
           //Pop up the elements in the stack sequentially as the left
boundary
//The pop-up elements decrease in order, so the height is the depth of
the column where each pop-up element is located
                    int pop = stack.pop();
                    height = depth[pop];
          //The distance from the width of the i-element to the column
where each peek element is located
                    if(stack.isEmpty()){
                        //If there are no elements at the top of the
stack, the width is 0 to i
                        width = i;
                    }else{
                        width = i - stack.peek() - 1;
```

```
                    }
                    maxArea = Math.max(maxArea,(long)height * width);
                }
    //The column where the i element is located will serve as the left
boundary of the subsequent area
                stack.push(i);
            }
        }
        //The remaining elements in the stack
        while(!stack.isEmpty()){
            height = depth[stack.pop()];
            if(stack.isEmpty()){
                width = length;
            }else {
                width = length - stack.peek() - 1;
            }
            maxArea = Math.max(maxArea,(long)height * width);
        }

        return maxArea;
    }
```