A2_Report_122090337

Problem_1

1 Initial Analysis

To solve such a complex task, the first thing that we should do is to divide the task into some subtasks, and then write the functions. The subtasks in this problem should be:

- create a class to store the data of Player
- input the data of Player
- sort the Player in the order of floor (or it will be hard to decide who should fight)
- **decide who will fight (IMPORTANT)**
- write a function for fight and win condition
- fight and store the data of winners/reserved ones
- sort the Player in the order of input order (required in the question)
- output and print

Some of the steps can be implemented in the main functions together. Among all the steps, deciding who will fight is the most significant step, because steps like sorting can be implemented by `sort` in the standard library. We need to come up with a smart way to decide the fight.

2 Possible Solutions for the problem

A good solution is starting from the top floor, and check each players' direction to decide fight or not. Then check players one by one from top to bottom. Because sometimes the winner of a fight may still encounter another opponent later (e.g. the player keeps going down), so the player may need to be added back to the checklist.

By the analysis above, we see really directly that this process is exactly what stack can do. That's why we use stack in this question. There is no need to hesitate to choose.

3 How to Solve the Problem

I will elaborate what is the meaning of my code.

```
class Player implements Comparable<Player> {
    int floor;
    int HP;
    String direction;
    int order;

    public Player(int floor, int HP, String direction, int order) {
        this.floor = floor;
        this.HP = HP;
        this.direction = direction;
        this.order = order;
    }

    @Override
    public int compareTo(Player other) {
        return this.floor - other.floor;
    }
}
```

Class Player is used to store the basic data of the players. `order` means the order of input, which will be used later in the output. `@Override` part is used to make the array of players sorted based on the floor.

```
public class BattleGame {

    public static Stack<Player> Floors = new Stack<Player>();
    public static Stack<Player> fightWaitlist = new Stack<Player>();

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int numberOfPlayers = sc.nextInt();
        Player[] floorLevels = new Player[numberOfPlayers];

        for (int i = 0; i < numberOfPlayers; i++) {
```

```
            int floor = sc.nextInt();
            int HP = sc.nextInt();
            String direction = sc.next();
            Player player = new Player(floor, HP, direction, i);
            floorLevels[i] = player;
        }
```

Let's talk about main function. What we do here is some basic preparation for fight. We use this part to input the data

```
        Arrays.sort(floorLevels);

        for (int i = 0; i < numberOfPlayers; i++) {
            Floors.push(floorLevels[i]);
        }
```

This part is to sort all the Object player in floor order, and then push them into a stack called Floor.

```
while (! Floors.isEmpty()) {
    Player initialPlayer = Floors.pop();
    if (initialPlayer.direction.equals("D")) {
        fightWaitlist.push(initialPlayer);
    } else if (initialPlayer.direction.equals("U")) {
        if (! fightWaitlist.isEmpty() && fightWaitlist.peek().direction.equals("D")) {
            Player winner = fight(initialPlayer, fightWaitlist.pop());
            if (winner.HP != 0) {
                Floors.push(winner);
            }
        } else {
            fightWaitlist.push(initialPlayer);
        }
    }
}
```

This is the most important part. The thinking process is that, we start from the top of the Floor (which is a stack full of player in floor order, top is the max floor). Pop the top player, if its direction is down, then it is in the waitlist for fight, or say `fightWaitlist`.

If the direction is up, there are two scenarios:

- One: The fightWaitlist is not empty (or say, there is possible opponent), as well as the top player in the `fightWailist` goes down (or say, these two players will meet each other). Then they will have a fight. After going through the fight function we have a winner, then push the winner back to `Floor` (since the winner may have a fight again)
- Two: the `fightWaitlist` is empty, or the top play in the waitlist goes up, then obviously there is no fight, because there isn't a meet of two players. We push the player to the waitlist for possible fight later.

A key here: We only we two stacks here, one for original players, one for fight waitlist. We don't need another "winnerStack" "fightStack" or something else. The process saves time.

```
    int size = fightWaitlist.size();
    Player[] finalFloors = new Player[size];
    for (int i = 0; i < size; i++) {
        finalFloors[i] = fightWaitlist.pop();
    }
```

Finally, those who can stay in the fight waitlist is the final winner, because there are only two possible result for a player: die or staying in the waitlist. Then push the winner into an array for sorting.

```
    Arrays.sort(finalFloors, Comparator.comparingInt(Player -> Player.order));

    for (int i = 0; i < size; i++) {
        System.out.println(finalFloors[i].HP);
    }
}
```

Sort the array based on the input order, then output.

The fight function after it is just a translation from english to java code. Three scenarios: player_1 wins, player_2 wins, ties, are all included.

Problem_2

1 Initial Analysis

This is a typical dynamic programming problem. The goal is to find the biggest square in the given area.

How to get the area of a square? Use $width \times depth$ (depth is actually the height). So the task is to find the biggest match of width and depth.

2 Possible Solutions for the problem

Brute Force

Directly traverse each column, take its height as the height of the rectangle, and then extend it forward and backward to obtain the maximum possible length of the rectangle. Then calculate the area of each rectangle, and finally take the maximum value.

Time complexity: Two layers of loops, therefore the time complexity is $O(N)$

DP

Why the brute force is bad? Because the data is wasted: we already know some possible width for rectangles, but we discard it. Dynamic programming tells us: it is good to store useful data for using later.

How to store? Use stack! Choosing stack is because we always need the most "recent" data of width, which is actually the concept of pop().

3 How to Solve the Problem

```
public class SearchTreasure {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int T = sc.nextInt();

        for (int t = 0; t < T; t++) {
            int Widths = sc.nextInt();
            sc.nextLine();

            int[] Depths = new int[Widths];
            for (int i = 0; i < Widths; i++) {
                Depths[i] = sc.nextInt();
            }
            sc.nextLine();

            long areaTreasure = TrenchSearch(Widths, Depths);
            System.out.println(areaTreasure);
        }
        sc.close();
    }
```

This is just for input the data.

```
public static long TrenchSearch(int Widths, int[] Depths) {

    long areaTreasure = 0;
    int LeftBoundary;
    Stack<Integer> depthStack = new Stack<Integer>();

    for (int curColumn = 0; curColumn < Widths; curColumn++) {

        while(! depthStack.isEmpty() && Depths[depthStack.peek()]>Depths[curColumn]) {
            int curDepth = Depths[depthStack.pop()];

            if (depthStack.isEmpty()) {
                LeftBoundary = 0;
            } else {
                LeftBoundary = depthStack.peek() + 1;
            }
            areaTreasure = Math.max(areaTreasure, (long)(curColumn - LeftBoundary) * curDepth);
        }

        depthStack.push(curColumn);
    }

    while(! depthStack.isEmpty()) {
        int curDepth = Depths[depthStack.pop()];
```

```
            if (depthStack.isEmpty()) {
                LeftBoundary = 0;
            } else {
                LeftBoundary = depthStack.peek() + 1;
            }
            areaTreasure = Math.max(areaTreasure,(long)(Widths - LeftBoundary) * curDepth);
        }

        return areaTreasure;
        }
```

This is the most important part. Stack can be used. As long as the top element of the stack is larger than the current one, the maximum area with the top element as the highest can be counted. Due to the monotonic nature of the elements within the stack, it can be determined that the right boundary of the rectangle is the previous bit of the current element, while the left boundary is the last bit of the top element of the stack after the pop operation. If the element in the stack is empty, it indicates that 0 is the left boundary. The following must be larger than it and the preceding must be smaller. If it is empty, it indicates that the left boundary is 0. If the monotonic stack is not empty after traversal, continue to count the area values in the same way, and finally return the maximum value.