Report for Assignment 2

Note:

In Possible Solution sections, I also include information in my solution. Plz be well-checked.

Problem 1. Battle Game

1.1 Possible Solutions & My Solution

Clearly, it is difficult to implement the algorithm given the input order, since the encounter of players are determined by their order of positions in the floors instead of the input order. It is natural to sort the players based on their positions in different floors. Since sorting is used, we need to record the initial input order (as an attribute) so that we can re-sorting to output the result.

```
data_list = []
n=int(input())
for i in range(n):
    num = i
    floor , hp , d = input().split()
    player = (int(floor), int(hp), d, num)
    data_list.append(player)
new_list = sorted(data_list,key=lambda s: s[0])
```

Now, we have a sorted list of players based on their floor number. We want to put the players into the battle. Note that, t***he only situation that two players will eventually engage in a battle is the player with higher floor heading downwards while the player with lower floor heading upwards.***

Now, we use a stack to record the temporally surviving players. We put the player one by one into the stack. To ensure all the members in the stack survive, ***we must have no situation mentioned above***. Now, there is a "challenger" coming to the stack. If it goes up, or all the elements (essentially the top elements) in stack goes down, or the stack is empty the new challenger has no effect on the equilibrium, we add it into the stack

```
if s.size==0 or s.peek()[2]=="D" or player[2] == "U":
    s.push(player)
```

For the remained case (nonempty stack & top player (defender) goes up & challenger goes down), there must be a battle, and the battle determines three cases (We pop out the defender to fight against the challenger):

```
while (not s.peek() is None) and s.peek()[2] == "U":
                defender = s.pop()
                defender_hp = defender[1]
                player_hp = player[1]
```

1. The defender wins (higher HP), the challenger will disappear (indicating by 0 HP), and the defender HP-1

   ```
   elif player_hp<defender_hp:
       player = (player[0],0 ,player[2], player[3])
       defender = (defender[0],defender_hp-1 , defender[2],
   defender[3])
       s.push(defender)
       break
   ```

2. It is a tie, both defender and the challenger will be eliminated

   ```
   if player_hp==defender_hp:
      player = (player[0],0 , player[2], player[3])
      break
   ```

3. The challenger wins, it has HP-1, and the defender disappear. The challenger will continue to challenge the equilibrium of the stack until it loses or no one else to challenge (that is, the next defender is heading downwards)

   ```
   elif player_hp>defender_hp:
      player = (player[0],player_hp-1 , player[2], player[3])
      continue
   ```

Once there is no one else to challenge and the challenger does not die, we push it into the stack:

```
if player[1]>0:
    s.push(player)
```

After inserting all the players into the stack, the stack must be in equilibrium, thus all the players are survivors.

We then sort the survivors by their input order and make outputs.

1.2 Advantages

1. The use of sorting of the input list could make we insert the players into stack sequentially, since the nearest players will interact first than others. The time complexity is $O(n \log n)$ using quick sort
2. Given the sorted array, we insert it one by one into the stack. Since each player has maximum 100HP, each time we put a player into the stack, there have at most 100 steps to achieve the equilibrium. We only need to traversal once every element in the sorted array, the time complexity is $O(100n) = O(n)$ when n is large.
3. therefore, even if there may be extremely large steps and many battles, since we only consider the battle results of adjacent players, we no longer need to think about the dynamics of the process, the overall time complexity is only $O(n \log n)$

Problem 2. Buried Treasure

To describe the problem in a mathematics language, indeed, this is equivalent to, given a list of numbers $[d_1, \cdots, d_n]$, we want to find the subarray such that the product of its length and the minimum value is maximized. i.e., Maximize the product of the subarray length and minimum.

2.1 Possible Solutions & My solution

The Brute-Force approach is to given every d_i, we find the maximum area, which needs $O(n^2)$ time complexity, too slow.

Analogous to the example from PPT, in fact, we can design algorithm with much faster complexity using stack to track the minimum value of subarray.

Now, we use a stack to record the indices of the elements from the list. Given a new element with index d_n from the list, we can check the top value of the stack and the corresponding value in the list (d_t). If $d_t < d_n$, it means that currently, adding the new element into the stack, the minimum value will not changed, thus we can safely push it into the stack without any concern, say:

```
for i in range(n):
        if s.size == 0 or array[s.peek()]<=array[i]:
            s.push(i)
```

However, if $d_t \geq d_n$, it means that we have encountered the new minimum, and the minimum in the past is no longer the real minimum. This means we cannot extend our stack safely by directly push the new element. In fact, all the parts that are larger than d_n is now restricted, thus need to be popped out. More specifically, as long as the the segments has width larger than d_n, it will now be popped out. Since s record the index of minimum value, we can pop s until the value is smaller than d_n, the part which is still useful for d_n's restriction. For other part, we can calculate the corresponding area. Indeed,

```
            while (s.size>0 and array[s.peek()]>array[i]):
                mini_height_index = s.pop()
                mini_height = array[mini_height_index]
```

The minimum is the height of the rectangle. The width of the rectangle is given by

```
if s.size == 0:
    width = i
else:
    width = i-s.peek()-1
```

We can then calculate the area and update the maximum area one by one, say

```
if product_area>max_product:
    max_product=product_area
```

Since d_n is the new minimum, we push index of d_n into the stack.

After doing this steps for all the elements in the list, we have the stack. If it is empty, then, the maximum product is indeed what we required. However, if it is not empty (meaning that from the corresponding index to the end of list, the minimum is unchanged), we need to pop those rectangles out to update our max product:

```
    while s.size>0:
        height = array[s.pop()]
        if s.size==0:
            width = n
        else:
            width = n-s.peek()-1
        product_area = width*height
        if product_area>max_product:
                    max_product=product_area
```

The resulting max product is the result for one query. We can use loops to get results for T queries.

1.3 Advantages

Avoid the use of $O(n^2)$ algorithm, we only need to traverse all the data once. And as long as they are considered, the maximum product will be updated correspondingly. The time complexity is only $O(n)$, which is much faster than the Brute force.