Exclusive Time of Functions

Given the running logs of **n** functions that are executed in a nonpreemptive single threaded CPU, find the exclusive time of these functions.

Each function has a unique id, start from **0** to **n-1**. A function may be called recursively or by another function.

A log is a string has this format : `function_id:start_or_end:timestamp`. For example, `"0:start:0"` means function 0 starts from the very beginning of time 0. `"0:end:0"` means function 0 ends to the very end of time 0.

Exclusive time of a function is defined as the time spent within this function, the time spent by calling other functions should not be considered as this function's exclusive time. You should return the exclusive time of each function sorted by their function id.

Example 1:

```
Input:
n = 2
logs =
["0:start:0",
 "1:start:2",
 "1:end:5",
 "0:end:6"]
Output:[3, 4]
Explanation:
Function 0 starts at time 0, then it executes 2 units of time and reaches the end of
time 1.
Now function 0 calls function 1, function 1 starts at time 2, executes 4 units of tim
e and end at time 5.
Function 0 is running again at time 6, and also end at the time 6, thus executes 1 un
it of time.
So function 0 totally execute 2 + 1 = 3 units of time, and function 1 totally execute
 4 units of time.
```

Note:

1. Input logs will be sorted by timestamp, NOT log id.
2. Your output should be sorted by function id, which means the 0th element of your output corresponds to the exclusive time of function 0.
3. Two functions won't start or end at the same time.
4. Functions could be called recursively, and will always end.
5. 1

Solution 1

```
public int[] exclusiveTime(int n, List<String> logs) {
    int[] res = new int[n];
    Stack<Integer> stack = new Stack<>();
    int prevTime = 0;
    for (String log : logs) {
        String[] parts = log.split(":");
        if (!stack.isEmpty()) res[stack.peek()] +=  Integer.parseInt(parts[2]) - pr
evTime;
        prevTime = Integer.parseInt(parts[2]);
        if (parts[1].equals("start")) stack.push(Integer.parseInt(parts[0]));
        else {
            res[stack.pop()]++;
            prevTime++;
        }
    }
    return res;
}
```

written by compton_scatter original link here

Solution 2

We examine two approaches - both will be stack based.

In a more conventional approach, let's look between adjacent events, with duration `time - prev_time`. If we started a function, and we have a function in the background, then it was running during this time. Otherwise, we ended the function that is most recent in our stack.

```
def exclusiveTime(self, N, logs):
    ans = [0] * N
    stack = []
    prev_time = 0

    for log in logs:
        fn, typ, time = log.split(':')
        fn, time = int(fn), int(time)

        if typ == 'start':
            if stack:
                ans[stack[-1]] += time - prev_time
            stack.append(fn)
            prev_time = time
        else:
            ans[stack.pop()] += time - prev_time + 1
            prev_time = time + 1

    return ans
```

In the second approach, we try to record the "penalty" a function takes. For example, if function 0 is running at time [1, 10], and function 1 runs at time [3, 5], then we know function 0 ran for 10 units of time, less a 3 unit penalty. The idea is this: **Whenever a function completes using T time, any functions that were running in the background take a penalty of T.** Here is a slow version to illustrate the idea:

```
def exclusiveTime(self, N, logs):
    ans = [0] * N
    #stack = SuperStack()
    stack = []

    for log in logs:
        fn, typ, time = log.split(':')
        fn, time = int(fn), int(time)

        if typ == 'start':
            stack.append(time)
        else:
            delta = time - stack.pop() + 1
            ans[fn] += delta
            #stack.add_across(delta)
            stack = [t+delta for t in stack] #inefficient

    return ans
```

This code already ACs, but it isn't efficient. However, we can easily upgrade our stack to a "superstack" that supports `self.add_across`: addition over the whole array in constant time.

```
class SuperStack(object):
    def __init__(self):
        self.A = []
    def append(self, x):
        self.A.append([x, 0])
    def pop(self):
        x, y = self.A.pop()
        if self.A:
            self.A[-1][1] += y
        return x + y
    def add_across(self, y):
        if self.A:
            self.A[-1][1] += y
```

written by awice original link here

Solution 3

0----1----2----3----4----5----6
|----|----|----|----|----|----|----|

fun0 - 0 to 1, 1 to 2 - 2 time units
fun1 - 2 to 3, 3 to 4, 4 to 5 - 3 time units
fun0 - 5 to 6 - 1 time units

Wondering how fun1 took 4 time units?

Update -

figured it, instead of seeing x as clock strike (or coordinate on line), see x as an item

[0][1][2][3][4][5][6]

Now it makes sense,
fun0 - took first 2 boxes
fun1 - took next 4 boxes
fun0 - took last 1 box

written by parvez.h.shaikh original link here