Solution: Exclusive Execution Time of Functions

Let's solve the Exclusive Execution Time of Functions problem using the Stacks pattern.

We'll cover the following

Statement

We are given an integer number, `n`, representing the number of functions running in a single-threaded CPU, and an execution log, which is essentially a list of strings. Each string has the format `{function id}:{"start" | "end"}:{timestamp}`, indicating that the function with `function id` either started or stopped execution at the time identified by the `timestamp` value. Each function has a unique ID between 0 and $n - 1$. Compute the exclusive time of the functions in the program.

> **Note:** The exclusive time is the sum of the execution times for all the calls to a specific function.

Constraints:

- $1 \leq$ `n` ≤ 100
- $1 \leq$ `logs.length` ≤ 500
- $0 \leq$ `function id` $<$ `n`
- $0 \leq$ `timestamp` $\leq 10^3$
- No two start events and two end events will happen at the same `timestamp`.
- Each function has an `end` log entry for each `start` log entry.

Solution

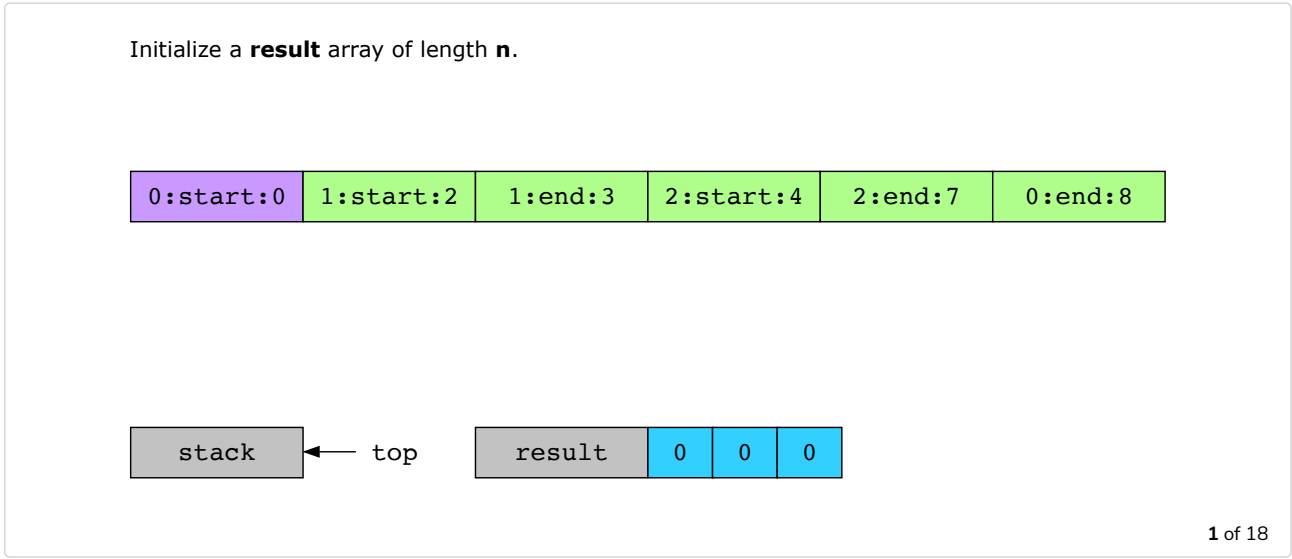
To find out the exclusive execution time of functions, we will use a stack. Just as a single-threaded CPU uses a stack to manage function execution, preemption, and resumption, we can use a stack to perform our calculation. Every time we see a new start event, we'll push the information regarding the previously running function onto the stack. When we see an end event, we'll pop the currently running function from the stack. That way, all end events will be matched with the corresponding start event, and the execution time is correctly computed.

The stack will contain the starting time of all functions in the program. Here's how the algorithm would work:

1. First, we'll read a line of text from the log and tokenize it to separate the function ID, the event type, and the timestamp.
2. If the event type is "start", push the current log details to the stack.
3. Otherwise, we pop the log details from the stack and add the execution time of the current function in the actual exclusive time.

4. If the stack is not empty, the current function is a nested function call. Therefore, we subtract the execution time of this called function from the calling function. We decrease the time in the calling function, in advance.
5. We store the execution time of each function at the index equal to the function ID in the `result` array.
6. When the same function is called recursively, we add the function's execution time to the current value at the specific index.

The following slides show how this solution works:



Let's look at the solution code below:

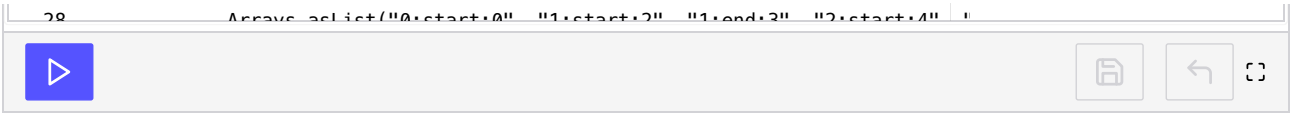
Java

main.java

Event.java

```
class TimeOfFunction {

    public static List<Integer> exclusiveTime(int n, List<String> events) {
        Deque<Event> stack = new ArrayDeque<>();
        List<Integer> result = new ArrayList<Integer> (Collections.nCopies(n,
        for (String content: events) {
            // Extract the log details from the content(string)
            Event event = new Event(content);
            if (event.getIsStart()) {
                // Push the event details to the stack
                stack.push(event);
            } else {
                // Pop the log details from the stack
                Event top = stack.pop();
                // Add the execution time of the current function in the actu
                result.set(top.getId(), result.get(top.getId()) + (event.getT
                // If the stack is not empty, subtract the current child func
                }
            }
        }
        return result;
    }
    public static void main(String args[]) {
        List<List<String>> events = Arrays.asList(
```



Exclusive Execution Time of Functions

Time complexity

Let's say there are m events in total. Each log line contains either "start" or "end", which is always a constant number of characters. The line also has a function ID and a start/end time. Since we're processing the string a single character at a time, we need to know how many digits the function ID has. It has $\log_{10} n$ digits. Then, there is the start/end time, which has $\log_{10} t$ digits, where t is the time at which the last event occurs. A particular event log entry takes $\log_{10}(n + t)$ time. Therefore, the overall running time is $O(m \log_{10}(n + t))$ or simply $O(m \log(n + t))$.

Space complexity

The space complexity is $O(m)$.

Back

Exclusive Execution Ti...

Next

Flatten Nested List Ite...

Mark as Completed