Assignment 5

Course: COMPSCI 2C03

Date: 11/09/2025

Author: Sulagna Nandi

MacID: nandis

Student Number: 400575118

Professor: Dr. Vincent Maccio

Problem 1.1

```
Step 1: Insert 78
78
  *  *

Step 2: Insert 6
6
  *  78
     *  *

Step 3: Insert 88
6
  *  78
     *  88

Step 4: Insert 19
6
  19  78
  *   88
  *

Step 5: Insert 77
6
  19  78
  77  88
```

```
  *

Step 6: Insert 79
6
  19  78
  77  88
  79  *

Step 7: Insert 94
6
  19  78
  77  88
  79  94

Step 8: Insert 7
6
  7   78
  19  88
  77  94
  79  *

Step 9: Insert 1
1
  6   78
  7   88
```

```
  19  94
  77  79

Step 10: Insert 36
1
  6   36
  7   78
  19  94
  77  79

\newpage
\section*{Problem 1.2}
\begin{verbatim}
Step 1: Insert 78
78
  *  *

Step 2: Insert 6
78
  6  *
     *  *

Step 3: Insert 88
88
  78  6
```

```
  *   *

Step 4: Insert 19
88
  78  6
  *   19

Step 5: Insert 77
88
  78  77
  6   19

Step 6: Insert 79
88
  79  77
  78  19
  6   *

Step 7: Insert 94
94
  88  79
  78  77
  6   19

Step 8: Insert 7
```

```
94
  88  79
  78  77
  6   19
  *   7

Step 9: Insert 1
94
  88  79
  78  77
  6   19
  1   7

Step 10: Insert 36
94
  88  79
  78  77
  6   36
  1   19

Step 11: Insert 934
934
  94  79
  88  77
  78  36
```

```
6   19
1   7
```

Problem 1.3

```
Step 1: Insert 78
78


Step 2: Insert 6
78
    6    *


Step 3: Insert 88
78
    6    88


Step 4: Insert 19
78
    6    88
        19    *


Step 5: Insert 77
78
    6    88
        19    *
            77    *


Step 6: Insert 79
78
```

```
    6    88
        19    *
           77    *
              79    *

Step 7: Insert 94
78
    6    88
        19    *
           77    *
              79    *
        *     94

Step 8: Insert 7
78
    6    88
        19    *
           7    77
              *    79
        *     94

Step 9: Insert 1
78
    6    88
        1    19
```

```
            7    77
                *    79
        *     94

Step 10: Insert 36
78
    6    88
        1    19
            7    77
                36    79
        *     94

Step 11: Insert 934
78
    6    88
        1    19
            7    77
                36    79
        *     94
                *    934
```

Problem 2.1

The algorithm below uses binary search to find the first start date that's greater than a (which uses O(log(|E|)) time in the worst case) and then linearly finds all the events with a start date greater than a and less than b (which uses O(|$result$|) time). That is why, in the worst case, the time complexity would be O(log(|E|) + |$result$|).

Algorithm 1 Range(E, a, b)

```
L = BinarySearchFirstGE(E, a)
result = [ ]
for i = L to |E| − 1 do
  if E[i].s > b then
    break
  end if
  append E[i] to result
end for
return result
```

Algorithm 2 BinarySearchFirstGE(E, a)

```
left = 0
right = |E| − 1
ans = |E|
while left ⩽ right do
  mid = ⌊(left + right)/2⌋
  if E[mid].s ⩾ a then
    ans = mid
    right = mid − 1
  else
    left = mid + 1
  end if
end while
return ans
```

Problem 2.2

In the tree T(E), each node stores an event with its start time, end time, and the largest end time in its entire subtree. The start time is what we use to navigate the tree: events with smaller start times are in the left child, and events with larger start times are in the right child. The end time tells us when the event finishes, and the maximum end time helps us skip over parts of the tree that cannot possibly have events active at a given time. To find all events still active at time a, we start at the root and check if the current event is active. If it is, we add it to our results. We then look at the left child only if its maximum end time is at least a, and we look at the right child if the current event starts before a. This way, we explore only the parts of the tree that might contain active events and ignore everything else. Since the tree is balanced, the search is very efficient. The time it takes in the worst case is proportional to the height of the tree plus the number of events that are actually active, which can be written as O($\log(|E|) + |result|$), where —result— is the number of active events found. Each event is stored only once in the tree, so the memory used is at most O($|E|$).

Algorithm 3 StillActive($T(E), a$)

```
result = [ ]
searchNode(T(E).root, a, result)
return result
```

Algorithm 4 searchNode($node, a, result$)

if $node$ is null **then**
  return
end if
if $node.left \neq null$ and $node.left.maxEnd \geqslant a$ **then**
  searchNode($node.left, a, result$)
end if
if $node.s < a$ and $node.e \geqslant a$ **then**
  append $(node.s, node.e)$ to $result$
end if
if $node.s < a$ **then**
  searchNode($node.right, a, result$)
end if

Problem 2.3

This algorithm builds the tree T(E) from a list of events that is already sorted by start time. The goal is to create a balanced tree where each node stores an event's start time, end time, and the largest end time in its whole subtree. The algorithm is designed to work recursively. It picks the middle event in the current list and makes it the root of the subtree. Then it builds the left subtree from all events before the middle and the right subtree from all events after the middle. Once both subtrees are built, it calculates the node's maximum end time by taking the largest value among its own end time and the maximum end times of its left and right children. Since it looks at each event only once and does a constant amount of work per node, the entire tree can be built in linear time, which is O(—E—). That's what makes this an efficient way to prepare the tree so that queries for active events can later be answered quickly.

Algorithm 5 BuildTree(E)

if E is empty **then**
  return null
end if
$mid = \lfloor |E|/2 \rfloor$
$node$ = new Node($E[mid].s, E[mid].e$)
$node.left$ = BuildTree($E[0 : mid]$)
$node.right$ = BuildTree($E[mid + 1 : |E|]$)
$node.maxEnd$ = max($node.e$,
$node.left.maxEnd$ if left exists,
$node.right.maxEnd$ if right exists)
return $node$