

MAT 243: Lecture Hall Problem Explanation

Dylan Lathrum

August 8, 2019

This is simply copy-pasted from the code so that it is easier to read the comments without an IDE or text editor with syntax highlighting. This is all of the relevant code for solving the problem.

```
process(events) {  
    // This is where the fun begins.  
    // This process() function is what controls everything. It takes one input in  
    // the form of a set containing all the events  
    // And outputs another set containing only the events that provide the highest  
    // attendance without any events overlapping.
```

The function takes one input in the form of the variable `events`, which is a set of every event and its details, and outputs another set that contains only the events that satisfy the question requirements.

The process can be split up into three parts:

1. Finding every possible combination of events
2. Pruning the combinations of instances where events overlap
3. Finding which of the remaining combinations have the highest number of attendees

```
// ~~~~~  
// STEP ONE  
// ~~~~~
```

Start by declaring a new array (or set) that will store our combinations. Note that there is an empty array (set) predefined, this is the same as what we would do in class...

`{}` -- A set that contains the empty set.

Note this combination will never be used, because any event that has attendees will always take precedence, but it's included to keep it similar to what we do in class.

```
var combinations = [[]];  
  
// For every event...  
for (var event of events) {  
    // Grab a copy of the combinations we've already made...  
    var copy = [...combinations];  
    // ...And for each of those already-existing combinations...  
    for (var element of copy) {
```

```

        // Add the next combination to our list of combinations
        combinations.push(element.concat(event));
    }
}

```

That last line reads as "To our list of COMBINATIONS, push another item (set) that contains our ELEMENT concatenated with the EVENT we're currently iterating over"

Now we have a new array (set) that contains every possible combination of events, including the null set and every event in the list. This is stored in the variable called `combinations`

```

//~~~~~
// STEP TWO
//~~~~~

// For every combination...
for(var i = 0; i < combinations.length; i++) {

```

We need to compare each event to each other. To do this, we will iterate over every item twice, so that every event is compared to every other even (Fun Fact: This means the complexity is $O(x^2)$, because for whatever length of events `x`, we have to compare each item individually)

```

    for(var j = 0; j < combinations[i].length; j++) {
        for(var k = 0; k < combinations[i].length; k++) {
            // If we are comparing an item/event to itself, skip it and continue to
            // the next comparison.
            // It does not matter if an event intersects with itself (because it
            // always will)
            if (j === k) {continue;}

```

This is where the logic lives! We need to compare two events, defined by a start time and an end time, to see if they overlap. Logically, we can find this by checking:

- If Event A Starts *AFTER* Event B Ends...

`EventAStart > EventBEnd` (I'll call this condition 'p' to simplify the logic)

- ... OR Event A Ends *AFTER* Event B Starts

`EventAEnd < EventBStart` (I'll call this condition 'q' to simplify the logic)

Which combines in the form we use in class to be...

`!(p ∨ q)`

...to which we can apply De Morgan's Law to make...

$\neg p \wedge \neg q$

Which when turned back to our extended form reads:

$(\text{EventAStart} < \text{EventBEnd}) \text{ AND } (\text{EventAEnd} > \text{EventBStart})$

Of course, the original conditional statement can be used, but I find that this is a lot easier to read as it directly implies that we're looking for overlap, not the inverse of no overlap.

```
// If there IS overlap in the two events
if ((combinations[i][j].start < combinations[i][k].end) &&
    (combinations[i][j].end > combinations[i][k].start)) {
    combinations[i].valid = false; //Mark the combination as invalid. We
will not consider this combination in the next step
}
// Otherwise just move on to the next comparison
}
}
}
```

For this demonstration, we will save the combinations to a variable that can be read by the website so we can display it in the middle of the page. This step is usually unnecessary.

```
this.setState({combinations});

//~~~~~
// STEP THREE
//~~~~~
```

We need to find the combination with the most attendees. To keep track, we will have two variables to identify the current "winning" combination

```
// We save the index of the best combination...
var highestIndex = 0;
// ...as well as it's actual value (the sum of attendees)
var highestSum = 0;

// For every combination...
for(i = 0; i < combinations.length; i++) {
```

We need to check if the combination is invalid. If it is, we can just continue on to the next combination without spending any time calculating it.

```
if (combinations[i].valid===false) {continue;}

// Create a variable for counting the sum of attendees for the combination
```

```

var sum = 0;
// For every event in the combination
for(j = 0; j < combinations[i].length; j++) {
    // Add the number of attendees to the sum
    sum += combinations[i][j].attendees;
}
// If the number of attendees for this combination is higher than our previous
// highest...
if (sum > highestSum) {
    // Set the highest sum to this combination's sum
    highestSum = sum;
    // And set the highest index to the current index
    highestIndex = i;
}
}

```

We have found the best combination! The solution is the combination at index `highestIndex`. This is where we return our result.

```

this.setState({results: combinations[highestIndex]})
}

```

The full code can also be found online at <https://github.com/Dylancyclone/MAT243-LectureHallProblem/blob/master/src/App.js#L48>