Solution Review: Problem Challenge 2

We'll cover the following

Maximum CPU Load (hard)

We are given a list of Jobs. Each job has a Start time, an End time, and a CPU load when it is running. Our goal is to find the **maximum CPU load** at any time if all the **jobs are running on the same machine**.

Example 1:

```
Jobs: [[1,4,3], [2,5,4], [7,9,6]]
Output: 7
Explanation: Since [1,4,3] and [2,5,4] overlap, their maximum CPU load (3+4=7) will be when bot
h the
jobs are running at the same time i.e., during the time interval (2,4).
```

Example 2:

```
Jobs: [[6,7,10], [2,4,11], [8,12,15]]
Output: 15
Explanation: None of the jobs overlap, therefore we will take the maximum load of any job whic
h is 15.
```

Example 3:

```
Jobs: [[1,4,2], [2,4,1], [3,6,5]]
Output: 8
Explanation: Maximum CPU load will be 8 as all jobs overlap during the time interval [3,4].
```

Solution

The problem follows the Merge Intervals pattern and can easily be converted to Minimum Meeting Rooms. Similar to 'Minimum Meeting Rooms' where we were trying to find the maximum number of meetings happening at any time, for 'Maximum CPU Load' we need to find the maximum number of jobs running at any time. We will need to keep a running count of the maximum CPU load at any time to find the overall maximum load.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```java
import java.util.*;

class Job {
  int start;
  int end;
  int cpuLoad;

  public Job(int start, int end, int cpuLoad) {
    this.start = start;
    this.end = end;
    this.cpuLoad = cpuLoad;
  }
};

class MaximumCPULoad {

  public static int findMaxCPULoad(List<Job> jobs) {
    // sort the jobs by start time
    Collections.sort(jobs, (a, b) -> Integer.compare(a.start, b.start));

    int maxCPULoad = 0;
    int currentCPULoad = 0;
    PriorityQueue<Job> minHeap = new PriorityQueue<>(jobs.size(), (a, b) -> Integer.compare(a.end, b.end)
    for (Job job : jobs) {
      // remove all jobs that have ended
      while (!minHeap.isEmpty() && job.start > minHeap.peek().end)
        currentCPULoad -= minHeap.poll().cpuLoad;

```

Run Save Reset

Time complexity

The time complexity of the above algorithm is $O(N * logN)$, where 'N' is the total number of jobs. This is due to the sorting that we did in the beginning. Also, while iterating the jobs, we might need to poll/offer jobs to the priority queue. Each of these operations can take $O(logN)$. Overall our algorithm will take $O(NlogN)$.

Space complexity

The space complexity of the above algorithm will be $O(N)$, which is required for sorting. Also, in the worst case, we have to insert all the jobs into the priority queue (when all jobs overlap) which will also take $O(N)$ space. The overall space complexity of our algorithm is $O(N)$.

← Back
Problem Challenge 2

Next →
Problem Challenge 3

✓ Mark as Completed

Report an Issue Ask a Question