Solution: Schedule Tasks on Minimum Machines

Let's solve the Schedule Tasks on Minimum Machines problem using the Two Heaps pattern.

We'll cover the following

Statement

Given a set of n number of tasks, implement a task scheduler method, **tasks()**, to run in $O(n \ \log n)$ time that finds the minimum number of machines required to complete these n tasks.

Constraints:

- $1 <=$ `tasks.length` $<= 10^4$
- $0 \leq task_i.start < task_i.end \leq 10^6$

Solution

This problem is a good candidate for the two heaps pattern since we require efficient access to these two quantities that are computed after sorting:

- The task with the earliest start time that has not yet been scheduled.
- The machine whose currently scheduled workload ends first.

For this reason, we maintain the following two heaps:

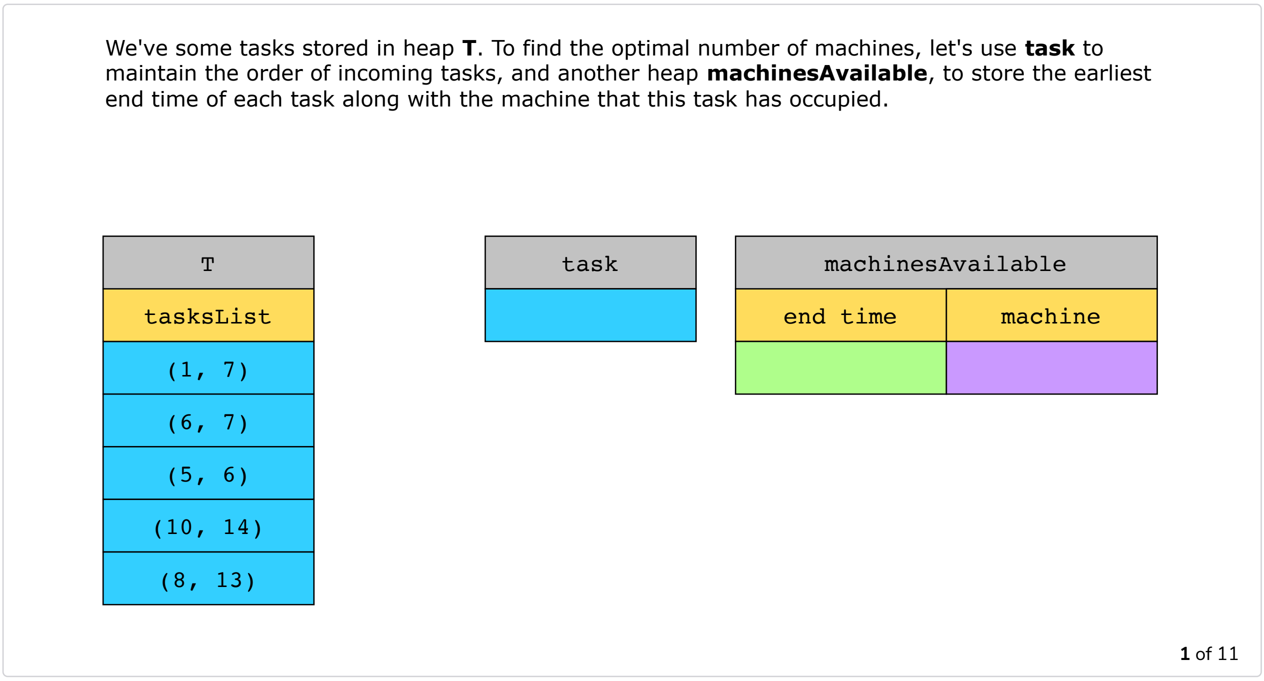
- The first heap stores the tasks to be scheduled such that the root of the heap will be the task with the earliest starting time.
- The second heap stores the list of machines such that the root of the heap will be the machine that will be free before all the others—that is, the machine whose last task ends the earliest.

Next, loop through the list of tasks to schedule them on available machines. For each task, perform the following:

- Check whether there are any machines in use. If there are, compare the end time of the machine whose workload ends the earliest with the start time of the current task. If the machine is free at the start time of the current task, we can use it. Therefore, we update the scheduled end time of this machine.
- If there are no machines in use or if no machines are available when the current task starts, we need a new machine. So, we schedule the task on a new machine and add it to the list of machines. Increment the count of machines used.

After processing all the tasks, we'll know the minimum number of machines required.

The slides below illustrate how we would like the algorithm to run:


We've some tasks stored in heap **T**. To find the optimal number of machines, let's use **task** to maintain the order of incoming tasks, and another heap **machinesAvailable**, to store the earliest end time of each task along with the machine that this task has occupied.

We can see the code of this solution below:

Java

```java
import java.util.*;

class ScheduleTask {
  public static int tasks(List<List<Integer>> tasksList) {
      // to count the total number of machines for "optimalMachines" tasks
      int optimalMachines = 0;

      PriorityQueue<int[]> machinesAvailable = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));
      PriorityQueue<int[]> tasksQueue = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));
      for (List<Integer> task: tasksList) {
          tasksQueue.offer(new int[] {
              task.get(0), task.get(1)
          });
      }

      while (!tasksQueue.isEmpty()) { // looping through the tasks list
          // remove from "tasksQueue" the task i with earliest start time
          int[] task = tasksQueue.poll();
          int[] machineInUse;
          if (!machinesAvailable.isEmpty() && task[0] >= machinesAvailable.peek()[0]) {
              // top element is deleted from "machinesAvailable"
              machineInUse = machinesAvailable.poll();
              // schedule task on the current machine
              machineInUse = new int[] {
                  task[1], machineInUse[1]
              };
          } else {
              // if there's a conflicting task, increment the
```

Schedule Tasks on Minimum Machines

Time complexity

The time complexity of this solution is $O(n \log n)$, where n is the total number of tasks in the heap. We are maintaining two heaps, so we have to include their contribution toward the time complexity. Since the cost of adding an element to a heap is $O(\log(sizeofheap))$, in the worst case, the cost of adding to the

$$\log(1) + \log(2) + \log(3) + ... + \log(n)$$

Stirling's approximation tells us that this sum is $O(n \times \log(n))$. Therefore, the total time complexity becomes $O(n \times\ log(n) + n \times \log(n))$, that is, $O(n \times \log\,(n))$.

Space complexity

Since the problem is solved using two heaps each of size n, the space complexity of this solution is $O(n) + O(n)$, which is $O(2n)$. Since we ignore the constants in computing complexities of a solution, the total space complexity of this solution becomes $O(n)$.

Alternative approach

Another approach is to use a min-heap data structure to keep track of the ending time of the tasks. We keep the earliest ending time at the root of the min-heap and compare the rest of the ending times of the tasks with the root of the min-heap. If a machine is getting free before the next task begins, we use that machine otherwise, we push the task to the min-heap. In the end, the levels of min-heap will be the minimum number of machines required to complete the tasks. This approach allows us to solve the problem using one min-heap instead of maintaining two heaps like this solution discussed above.

← Back
Schedule Tasks on Mi...

Next →
K-way Merge: Introdu...

✓ Mark as Completed