

Task Scheduler

Greedy scheduling with cooldown — Max-Heap + queue combo ■

● MEDIUM

■ Hook

Your CPU has tasks to execute but each task type needs a cooldown of n intervals before it can repeat. Minimize total execution time. This is a **greedy scheduling problem**. The greedy insight: always schedule the most frequent available task next. A max-heap gives us the most frequent task instantly. A cooldown queue tracks when tasks become available again.

■ Core Intuition — Two Structures Working Together

Use a **max-heap** (ordered by frequency) for available tasks, and a **queue** for tasks on cooldown with their 'available at' time. At each time tick: pop the most frequent available task, execute it, put it in cooldown queue with time = current + $n + 1$. Move any cooled-down tasks from queue back to heap.

■ Math Formula Alternative ($O(n)$)

Formula: $\max((\text{maxFreq} - 1) * (n + 1) + \text{countOfMaxFreq}, \text{tasks.length})$ maxFreq = highest task frequency. countOfMaxFreq = how many tasks share that frequency. The formula gives minimum intervals without simulation — $O(n)$ time. Know both approaches!

■ Visual — tasks=[A,A,A,B,B,B], n=2

```
Frequencies: A=3, B=3   Cooldown n=2
Max-Heap: [A(3), B(3)]   Queue: []   time=0
t=0: Execute A(3) → freq drops to A(2). Cooldown until t=3.
      Heap=[B(3)]   Queue=[(A,ready@3)]
t=1: Execute B(3) → freq drops to B(2). Cooldown until t=4.
      Heap=[]   Queue=[(A,@3),(B,@4)]
t=2: Heap empty! IDLE. Queue still cooling.
      Heap=[]   Queue=[(A,@3),(B,@4)]
t=3: Queue front (A,@3): ready! Move A(2) back to heap.
      Execute A(2)→A(1). Cooldown until t=6.
      Heap=[B—not yet]   Queue=[(B,@4),(A,@6)]
t=4: B(2) ready! Execute B(2)→B(1). Cooldown until t=7.
t=5: IDLE (nothing available)
t=6: A(1) ready. Execute → done.
t=7: B(1) ready. Execute → done.
TOTAL = 8 intervals   (formula: (3-1)*(2+1)+2 = 8 ✓)
```

■ Code — Simulation Approach + Formula Approach

```
import java.util.*;
public class TaskScheduler {
    // Simulation: Max-Heap + Cooldown Queue O(n * intervals)
    public int leastInterval(char[] tasks, int n) {
        int[] freq = new int[26];
        for (char t : tasks) freq[t - 'A']++;
        // Max-heap of frequencies
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());

        for (int f : freq) if (f > 0) maxHeap.offer(f);
        // Queue of [remaining_freq, available_at_time]
        Queue<int[]> cooldown = new LinkedList<>();
        int time = 0;
        while (!maxHeap.isEmpty() || !cooldown.isEmpty()) {
            time++;
            if (!maxHeap.isEmpty()) {
                int f = maxHeap.poll() - 1; // execute task
                if (f > 0) cooldown.offer(new int[]{f, time + n});
            }
            // Release cooled-down tasks
            if (!cooldown.isEmpty() && cooldown.peek()[1] == time)
                maxHeap.offer(cooldown.poll()[0]);
        }
        return time;
    }
    // Formula: O(n) — Math insight
    public int leastIntervalFormula(char[] tasks, int n) {
        int[] freq = new int[26];
        for (char t : tasks) freq[t - 'A']++;
        Arrays.sort(freq);
        int maxFreq = freq[25];
        int countMax = 0;
        for (int f : freq) if (f == maxFreq) countMax++;
        return Math.max(tasks.length,
            (maxFreq - 1) * (n + 1) + countMax);
    }
}
```

■ Time & Space Complexity

■ TIME COMPLEXITY

$O(n * \text{time}) / O(n)$

■ SPACE COMPLEXITY

$O(n)$

Simulation: $O(n * \text{total_intervals})$ — can be $O(n^2)$ worst case. Formula approach: $O(n \log n)$ for sorting or $O(26) = O(1)$ for fixed alphabet. Space: $O(26) = O(1)$ for fixed task types.

■ Common Mistakes

■ Confusing n with $n+1$ in the formula

The formula uses $(n+1)$ as the 'cycle length' — you need n idle slots BETWEEN executions, plus 1 for the task itself. So a cycle is $n+1$ intervals long.

■ Not handling IDLE slots in simulation

When the heap is empty but cooldown queue is not, you MUST still increment time (IDLE). Skipping this step gives wrong answers when all tasks are cooling simultaneously.

■ Practice Problems

#	Problem	Diff	Pattern
1	Task Scheduler (LC 621)	Medium	Max-Heap + Cooldown Queue
2	Rearrange String K Distance Apart (LC 358)	Hard	Max-Heap + Queue
3	Reorganize String (LC 767)	Medium	Max-Heap + prev tracking
4	Maximum Frequency Stack (LC 895)	Hard	Freq + Stack
5	Process Tasks Using Servers (LC 1882)	Hard	Two Min-Heaps

■ Quick Revision

- ✓ Greedy: always execute the most frequent available task
- ✓ Max-heap stores available task frequencies
- ✓ Cooldown queue stores (remaining_freq, available_at_time)
- ✓ IDLE when heap is empty but queue is not — still increment time
- ✓ Formula: $\max(\text{tasks.length}, (\text{maxFreq}-1)*(n+1)+\text{countMax})$
- ✓ Time: $O(n*\text{intervals})$ simulation | $O(n)$ formula

■ MENTAL MODEL — Remember This Forever

Think of it as a music DJ with a rule: you can't play the same song within n songs. The DJ always plays the most-requested song that's currently 'off cooldown'. When no song is available, play a filler track (IDLE). The most-requested songs drive the structure — they appear in every cycle, and the minimum time is determined by how many cycles the most popular song needs.