How to understand the solution to Task Scheduler problem on LeetCode?

Asked 1 year, 8 months ago Modified 1 year, 8 months ago Viewed 2k times

4

LeetCode Task Scheduler problem is the following:

Given a characters array tasks, representing the tasks a CPU needs to do, where each letter represents a different task. Tasks could be done in any order. Each task is done in one unit of time. For each unit of time, the CPU could complete either one task or just be idle.

Return the least number of units of times that the CPU will take to finish all the given tasks.

Example 1:

Input: tasks = ["A","A","A","B","B","B"], n = 2

Output: 8

Explanation:

A -> B -> idle -> A -> B -> idle -> A -> B There are at least 2 units of time between any two same tasks.

Example 2:

Input: tasks = ["A","A","A","A","A","A","B","C","D","E","F","G"], n = 2

Output: 16

-> A -> F -> G -> A -> idle ->

> int:

```
        for i in range(1, 26):
            idlesState -= min(freq[i], idleBlocks)

        return len(tasks) + max(0, idlesState)
```

Basically, it works like this:

Given the tasks `["A","A","A","A","A","A","B","C","D","E","F","G"]`

1. Sort the tasks by frequency descendingly

```
{ A: 6, B: 1, C: 1, D: 1, E: 1, F: 1, G: 1 }
```

2. We first place the most frequent character. All the spots between the same characters are first idle.

```
A _ _ A _ _ A _ _ A _ _ A _ _ A
```

3. We try to fill the remaining characters in the idleSpots using the sorted task array. (most frequent filled first)

```
A B C A D E A F G A _ _ A _ _ A
```

4. If the idleSpots < 0, we return the total number of tasks, else we return the total number of tasks + idleSpots.

I'm having issues proving this statement:

```
If the idleSpots < 0, we return the total number of tasks.
```

veen the most frequent character, why
tary idle tasks?

etween all the As are filled.

5

80

asked Oct 29, 2020 at 18:12

desmond.belphegor

43 3

2

When `idleSpot` (or, as in the code, `idleState`) is zero or negative, it means we have enough tasks other than `A` to fill the required **minimum** cooldown periods between `A` tasks.

Well, suppose `BB` are the overflowing tasks. Instead of appending them to the end of scheduling queue, we can just insert them, one right after each `A`. So instead of

```
A X X A X X A B _ _ B _ _
```

we will have

```
A B X X A B X X A.
```

Note that the cooldown periods between all existing tasks of the same type are at least as large as before. The cooldown periods between `B`s are as long as the cooldown periods between `A`s, so they are no shorter than the minimum as well.

If we have addition overflowing tasks, such as `CC`, then we can do the same thing, inserting them one right after each `A`. So we would have

```
A C B X X A C B X X A.
```

Note that the cooldown periods between all existing tasks of the same type are at least as large as before. The cooldown periods between `C`s are as long as the cooldown periods between `A`s, so they are no shorter than the minimum as well.

And so on.

Share Cite Improve this answer Follow

answered Oct 31, 2020 at 1:35

John L.
34.2k 4 29 80

Thanks a lot! It's really clear! – desmond.belphegor Nov 9, 2020 at 19:59