Previous (/articles/maximum-distance-in-array/) Next (/articles/minimum-factorization/)

621. Task Scheduler (/problems/task-scheduler/)

June 17, 2017 | 162.1K views

Average Rating: 2.42 (252 votes)

Given a char array representing tasks CPU need to do. It contains capital letters A to Z where different letters represent different tasks. Tasks could be done without original order. Each task could be done in one interval. For each interval, CPU could finish one task or just be idle.

However, there is a non-negative cooling interval **n** that means between two **same tasks**, there must be at least n intervals that CPU are doing different tasks or just be idle.

You need to return the **least** number of intervals the CPU will take to finish all the given tasks.

Example:

```
Input: tasks = ["A","A","A","B","B","B"], n = 2
Output: 8
Explanation: A -> B -> idle -> A -> B -> idle -> A -> B.
```

Constraints:

- The number of tasks is in the range `[1, 10000]`.
- The integer `n` is in the range `[0, 100]`.

Solution

Approach #1 Using Sorting [Accepted]

Before we start off with the solution, we can note that the names of the tasks are irrelevant for obtaining the solution of the given problem. The time taken for the tasks to be finished is only dependent on the number of instances of each task and not on the names of tasks.

The first solution that comes to the mind is to consider the tasks to be executed in the descending order of their number of instances. For every task executed, we can keep a track of the time at which this task was executed in order to consider the impact of cooling time in the future. We can execute all the tasks in the descending order of their number of instances and can keep on updating the number of instances pending for each task as well. After one cycle of the task list is executed, we can again start with the first task(largest count of instances) and keep on continuing the process by inserting idle cycles wherever appropriate by considering the last execution time of the task and the cooling time as well.

But, there is a flaw in the above idea. Consider the case, where say the number of instances of tasks A, B, C, D, E are 6, 1, 1, 1, 1 respectively with n=2(cooling time). If we go by the above method, firstly we give 1 round to each A, B, C, D and E. Now, only 5 instances of A are pending, but each instance will take 3 time units to complete because of cooling time. But a better way to schedule the tasks will be this: A, B, C, A, D, E, ... In this way, by giving turn to the task A as soon as its cooling time is over, we can save a good number of clock cycles.

From the above example, we are clear with one idea. It is that, the tasks with the currently maximum number of outstanding (pending)instances will contribute to a large number of idle cycles in the future, if not executed with appropriate interleavings with the other tasks. Thus, we need to re-execute such a task as soon as its cooling time is finished.

Thus, based on the above ideas, firstly, we obtain a count of the number of instances of each task in map array. Then, we start executing the tasks in the order of descending number of their initial instances. As soon as we execute the first task, we start its cooling timer as well(i). For every task executed, we update the pending number of instances of the current task. We update the current time, $time$, at every instant as well. Now, as soon as the timer, i's value exceeds the cooling time, as discussed above, we again need to consider the task with the largest number of pending instances. Thus, we again sort the $tasks$ array with updated counts of instances and again pick up the tasks in the descending order of their number of instances.

Now, the task picked up first after the sorting, will either be the first task picked up in the last iteration(which will now be picked after its cooling time has been finished) or the task picked will be the one which lies at $(n+1)^{th}$ position in the previous descending $tasks$ array. In either of the cases, the cooling time won't cause any conflicts(it has been considered implicitly). Further, the task most critical currently will always be picked up which was the main requirement.

We stop this process, when the pending instances of all the tasks have been reduced to 0. At this moment, $time$ gives the required result.

Java

Copy

```java
public class Solution {
    public int leastInterval(char[] tasks, int n) {
        int[] map = new int[26];
        for (char c: tasks)
            map[c - 'A']++;
        Arrays.sort(map);
        int time = 0;
        while (map[25] > 0) {
            int i = 0;
            while (i <= n) {
                if (map[25] == 0)
                    break;
                if (i < 26 && map[25 - i] > 0)
                    map[25 - i]--;
                time++;
                i++;
            }
            Arrays.sort(map);
        }
        return time;
    }
}
```

Complexity Analysis

- Time complexity : $O(time)$. Number of iterations will be equal to resultant time $time$.
- Space complexity : $O(1)$. Constant size array map is used.

Approach #2 Using Priority-Queue [Accepted]

Algorithm

Instead of making use of sorting as done in the last approach, we can also make use of a Max-Heap($queue$) to pick the order in which the tasks need to be executed. But we need to ensure that the heapification occurs only after the intervals of cooling time, n, as done in the last approach.

To do so, firstly, we put only those elements from map into the $queue$ which have non-zero number of instances. Then, we start picking up the largest task from the $queue$ for current execution. (Again, at every instant, we update the current $time$ as well.) We pop this element from the $queue$. We also decrement its pending number of instances and if any more instances of the current task are pending, we store them(count) in a temporary $temp$ list, to be added later on back into the $queue$. We keep on doing so, till a cycle of cooling time has been finished. After every such cycle, we add the generated $temp$ list back to the $queue$ for considering the most critical task again.

We keep on doing so till the $queue$(and $temp$) become totally empty. At this instant, the current value of $time$ gives the required result.

Java　　Articles > 621. Task Scheduler　　Copy

```java
public class Solution {
    public int leastInterval(char[] tasks, int n) {
        int[] map = new int[26];
        for (char c: tasks)
            map[c - 'A']++;
        PriorityQueue < Integer > queue = new PriorityQueue < > (26,
Collections.reverseOrder());
        for (int f: map) {
            if (f > 0)
                queue.add(f);
        }
        int time = 0;
        while (!queue.isEmpty()) {
            int i = 0;
            List < Integer > temp = new ArrayList < > ();
            while (i <= n) {
                if (!queue.isEmpty()) {
                    if (queue.peek() > 1)
                        temp.add(queue.poll() - 1);
                    else
                        queue.poll();
                }
                time++;
                if (queue.isEmpty() && temp.size() == 0)
                    break;
                i++;
            }
            for (int l: temp)
```

Complexity Analysis

- Time complexity : $O(n)$. Number of iterations will be equal to resultant time $time$.
- Space complexity : $O(1)$. $queue$ and $temp$ size will not exceed O(26).

Approach #3 Calculating Idle slots [Accepted]

Algorithm

This approach is inpired by @zhanzq (https://leetcode.com/zhanzq)

If we are able to, somehow, determine the number of idle slots($idle\_slots$), we can find out the time required to execute all the tasks as $idle\_slots + TotalNumberOfTasks$. Thus, the idea is to find out the idle time first.

To find the idle time, consider figure 1 below.

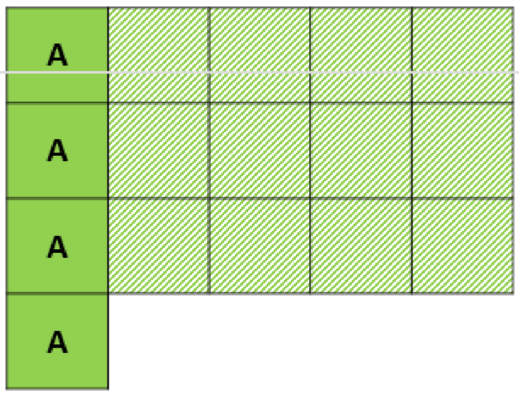

Figure 1

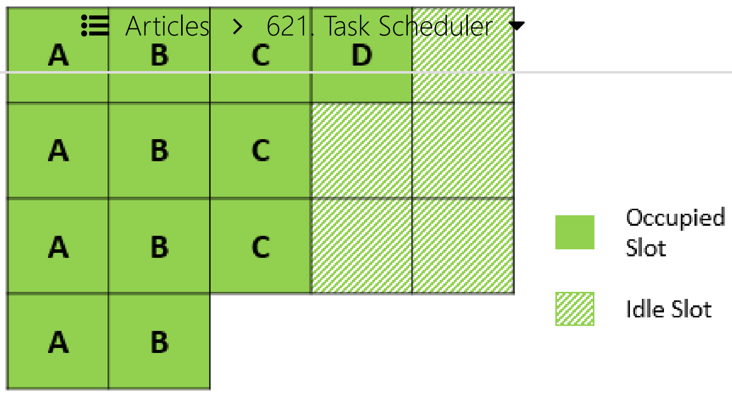

Figure 2

From this figure, we can observe that the maximum number of idle slots will always be given by the product of the cooling time and the number of instances of the task with maximum count less 1(in case only multiple instances of the same task need to be executed, and each, then, is executed after lapse of every cooling time). The factor of 1 is deducted from the task's count with maximum number of instances, as is clear from the figure, is that in the last round of execution of the tasks, the idle slots need not be considered for insertion following the execution of the related task. Now, based on the count of the instances of the other tasks, we can reduce the number of idle slots from this maximum value, to determine the minimum number of idle slots needed.

To do so, consider figure 2 as shown above. From the figure above, assuming the tasks are executed in row-wise order, we can see that in case the number of instances of another task equal the number of instances of the task with maximum number of instances, the number of idle slots saved is equal to its number of instances less 1 as is clear for the case of task B above. But, if the count of the number of instances, say i is lesser than the this maximum value, the number of idle slots saved is equal to the value i itself as is clear for the case of task C. Further, we can observe that for any arbitrary task other than A, B or C with the count of number of instances lesser than C, this task can be easily accomodated into the idle slots or if no more idle slot is available, this task can be appended after every row of tasks without interfering with the cooling time. In the first case, subtracting its number of intances from the number of idle slots leads to obtaining the correct number of available idle slots. In the second case, which will only occur if the number of idle slots pending is already zero, it leads to negative net idle slots, which can later be considered as zero for the purpose of calculations.

Thus, we can easily obtain the number of pending idle slots by subtracting appropriate number of slots from the available ones and at the end, we can obtain the total time required as the sum of pending idle slots and the total number of tasks.

Java Copy

```java
public class Solution {
    public int leastInterval(char[] tasks, int n) {
        int[] map = new int[26];
        for (char c: tasks)
            map[c - 'A']++;
        Arrays.sort(map);
        int max_val = map[25] - 1, idle_slots = max_val * n;
        for (int i = 24; i >= 0 && map[i] > 0; i--) {
            idle_slots -= Math.min(map[i], max_val);
        }
        return idle_slots > 0 ? idle_slots + tasks.length : tasks.length;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. We iterate over $tasks$ array only once. ($O(n)$).Sorting $tasks$ array of length n takes $O\big(26log(26)\big) = O(1)$ time. After this, only one iteration over 26 elements of map is done($O(1)$).
- Space complexity : $O(1)$. map array of constant size(26) is used.

Rate this article:

Previous (/articles/maximum-distance-in-array/)

Next (/articles/minimum-factorization/)

Comments: 106

Sort By

Type comment here... (Markdown is supported)

Preview Post

teterin (/teterin) 380 January 5, 2019 2:56 AM

Description of task is really bad. I still cannot understand the general rule

380 Share Reply

SHOW 5 REPLIES

haoguoxuan (/haoguoxuan) 187 October 8, 2018 1:31 PM

solution works, yet explanations been poorly written, hard to understand.

131 Share Reply

HareshMiriyala (/hareshmiriyala) 126 January 15, 2019 9:56 PM

Poor wording of the problem . I still don't get the question :/

108 Share Reply

aureole-420 (/aureole-420) ★ 130 August 28, 2018 10:27 AM

Was wondering why this article is LC official. All three solutions are greedy but none has a proof.

88 Share Reply

xinx (/xinx) ★ 59 March 18, 2018 3:32 AM

Am I the only one that felt the writing of this article very confusing?
Is it possible to revise the wording and expressions of this article please?

58 Share Reply

SHOW 1 REPLY

HY_huaiyu (/hy_huaiyu) ★ 164 February 9, 2019 10:00 PM

too long-winded explanations :(

54 Share Reply

kk42 (/kk42) ★ 81 October 22, 2018 8:45 AM

An array called map, so misleading...

71 Share Reply

SHOW 3 REPLIES

logical_paradox (/logical_paradox) ★ 244 October 23, 2018 9:50 AM

What is the `O(time)` in approach 1. This doesn't make any sense. Shouldn't complexity be calculated based on the input? So it should be in terms of number of tasks, even in the first case. Could you guys, tell us what is the complexity for approach 1 in relation to the input?

34 Share Reply

SHOW 2 REPLIES

csreddy (/csreddy) ★ 27 April 4, 2018 11:02 AM

No kidding, me too I can't follow the logic described. But I do appreciate the person for putting the effort.

27 Share Reply

Cloudson (/cloudson) ★ 92 March 14, 2018 4:21 PM

Why O(time) is the time complexity of the first approach? Does this include the Array.Sort() function?

16 Share Reply

SHOW 1 REPLY

‹ 1 2 3 4 5 6 ... 10 11 ›