Task Scheduler

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 1:

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

Note:

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

Solution 1

I think this should be correct, math plays an important role here.

```
public class Solution {
    public int leastInterval(char[] tasks, int n) {

        if(tasks.length == 0) return 0;
        if(n == 0) return tasks.length;

        int[] c = new int[26];
        for(char t : tasks){
            c[t - 'A']++;
        }
        Arrays.sort(c);
        int i = 25;
        while(i >= 0 && c[i] == c[25]) i--;

// (c[25] - 1) * (n + 1) + 25 - i  is frame size
// when inserting chars, the frame might be "burst", then tasks.length takes precede
nce
// when 25 - i > n, the frame is already full at construction, the following is stil
l valid.
        return Math.max(tasks.length, (c[25] - 1) * (n + 1) + 25 - i);
    }
}
```

First consider the most frequent characters, we can determine their positions first and use them as a frame to insert the remaining less frequent characters. Here is a proof by construction:

Let F be the set of most frequent chars with frequency k.
Then we can create k chunks, each chunk is identical and is a string consists of chars in F in a specific fixed order.
Let the heads of these chunks to be H_i, then H_2 should be at least n chars away from H_1, and so on so forth; then we insert the less frequent chars into the gaps between these chunks sequentially one by one ordered by frequency in a decreasing order and try to fill the k-1 gaps full each time. In another word, Append the less frequent characters to the end of each chunk of the first k-1 chunks sequentially and round and round, then join the chunks.

Examples:

AAAABBBEEFFGG 3

here X represents a space gap:

```
Frame: "AXXXAXXXAXXXA"
insert 'B': "ABXXABXXABXXA" <--- 'B' has higher frequency than the other characters,
insert it first.
insert 'E': "ABEXABEXABXXA"
insert 'F': "ABEFABEXABFXA" <--- each time try to fill the k-1 gaps as full or evenl
y as possible.
insert 'G': "ABEFABEGABFGA"
```

AACCCBEEE 2

```
3 identical chunks "CE", "CE CE CE" <-- this is a frame
insert 'A' among the gaps of chunks since it has higher frequency than 'B' ---> "CEA
CEACE"
insert 'B' ---> "CEABCEACE" <----- result is tasks.length;
```

AACCCDDEEE 3

```
3 identical chunks "CE", "CE CE CE" <--- this is a frame.
Begin to insert 'A'->"CEA CEA CE"
Begin to insert 'B'->"CEABCEABCE" <---- result is tasks.length;
```

ACCCEEE 2

```
3 identical chunks "CE", "CE CE CE" <-- this is a frame
Begin to insert 'A' --> "CEACE CE" <-- result is (c[25] - 1) * (n + 1) + 25 -i = 2
* 3 + 2 = 8
```

written by fatalme original link here

Solution 2

The idea is:

0. To work on the same task again, CPU has to wait for time `n`, therefore we can think of as if there is an `cycle`, of `time n+1`, regardless whether you schedule some other task in the cycle or not.
1. To avoid leave the CPU with limited choice of tasks and having to sit there cooling down frequently at the end, it is critical the keep the diversity of the task pool for as long as possible.
2. In order to do that, we should try to schedule the CPU to `always try round robin between the most popular tasks at any time`.

priority_queue< `task`, `count` >

```
class Solution {
public:
    int leastInterval(vector<char>& tasks, int n) {
        unordered_map<char, int> counts;
        for (char t : tasks) {
            counts[t]++;
        }
        priority_queue<pair<int, char>> pq;
        for (pair<char, int> count : counts) {
            pq.push(make_pair(count.second, count.first));
        }
        int alltime = 0;
        int cycle = n + 1;
        while (!pq.empty()) {
            int time = 0;
            vector<pair<int, char>> tmp;
            for (int i = 0; i < cycle; i++) {
                if (!pq.empty()) {
                    tmp.push_back(pq.top());
                    pq.pop();
                    time++;
                }
            }
            for (auto t : tmp) {
                if (--t.first) {
                    pq.push(t);
                }
            }
            alltime += !pq.empty() ? cycle : time;
        }
        return alltime;
    }
};
```

priority_queue< `count` >

As @milu point out, we don't really need to store <task - count> pair in the priority_queue, we don't need to know the task name, store counts works good enough:

```cpp
class Solution {
public:
    int leastInterval(vector<char>& tasks, int n) {
        unordered_map<char, int> counts;
        for (char t : tasks) {
            counts[t]++;
        }
        priority_queue<int> pq;
        for (pair<char, int> count : counts) {
            pq.push(count.second);
        }
        int alltime = 0;
        int cycle = n + 1;
        while (!pq.empty()) {
            int time = 0;
            vector<int> tmp;
            for (int i = 0; i < cycle; i++) {
                if (!pq.empty()) {
                    tmp.push_back(pq.top());
                    pq.pop();
                    time++;
                }
            }
            for (int cnt : tmp) {
                if (--cnt) {
                    pq.push(cnt);
                }
            }
            alltime += !pq.empty() ? cycle : time;
        }
        return alltime;
    }
};
```

Java Version

```java
public class Solution {
    public int leastInterval(char[] tasks, int n) {
        Map<Character, Integer> counts = new HashMap<Character, Integer>();
        for (char t : tasks) {
            counts.put(t, counts.getOrDefault(t, 0) + 1);
        }

        PriorityQueue<Integer> pq = new PriorityQueue<Integer>((a, b) -> b - a);
        pq.addAll(counts.values());

        int alltime = 0;
        int cycle = n + 1;
        while (!pq.isEmpty()) {
            int worktime = 0;
            List<Integer> tmp = new ArrayList<Integer>();
            for (int i = 0; i < cycle; i++) {
                if (!pq.isEmpty()) {
                    tmp.add(pq.poll());
                    worktime++;
                }
            }
            for (int cnt : tmp) {
                if (--cnt > 0) {
                    pq.offer(cnt);
                }
            }
            alltime += !pq.isEmpty() ? cycle : worktime;
        }

        return alltime;
    }
}
```

written by alexander original link here

Solution 3

The idea used here is similar to -https://leetcode.com/problems/rearrange-string-k-distance-apart
We need to arrange the characters in string such that each same character is K distance apart, where distance in this problems is time b/w two similar task execution.

Idea is to add them to a priority Q and sort based on the highest frequency.
And pick the task in each round of 'n' with highest frequency. As you pick the task, decrease the frequency, and put them back after the round.

```
public int leastInterval(char[] tasks, int n) {
     Map<Character, Integer> map = new HashMap<>();
    for (int i = 0; i < tasks.length; i++) {
        map.put(tasks[i], map.getOrDefault(tasks[i], 0) + 1); // map key is TaskName
, and value is number of times to be executed.
    }
    PriorityQueue<Map.Entry<Character, Integer>> q = new PriorityQueue<>( //frequenc
y sort
            (a,b) -> a.getValue() != b.getValue() ? b.getValue() - a.getValue() : a
.getKey() - b.getKey());

    q.addAll(map.entrySet());

    int count = 0;
    while (!q.isEmpty()) {
        int k = n + 1;
        List<Map.Entry> tempList = new ArrayList<>();
        while (k > 0 && !q.isEmpty()) {
            Map.Entry<Character, Integer> top = q.poll(); // most frequency task
            top.setValue(top.getValue() - 1); // decrease frequency, meaning it got
executed
            tempList.add(top); // collect task to add back to queue
            k--;
            count++; //successfully executed task
        }

        for (Map.Entry<Character, Integer> e : tempList) {
            if (e.getValue() > 0) q.add(e); // add valid tasks
        }

        if (q.isEmpty()) break;
        count = count + k; // if k > 0, then it means we need to be idle
    }
    return count;
}
```

written by jaqenhgar original link here