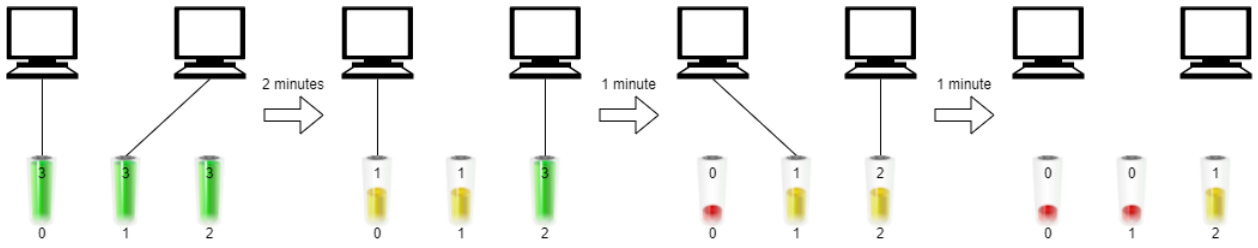
Day 33 / 100 :

Topic - Array

1 Problem statement: Maximum running time of N computers (Hard)

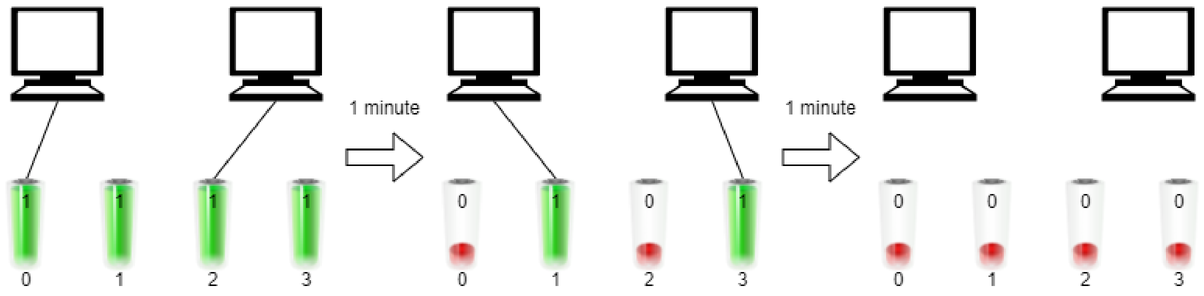
You have n computers. You are given the integer n and a 0-indexed integer array batteries where the ith battery can run a computer for batteries[i] minutes. You are interested in running all n computers simultaneously using the given batteries.
Initially, you can insert at most one battery into each computer. After that and at any integer time moment, you can remove a battery from a computer and insert another battery any number of times. The inserted battery can be a totally new battery or a battery from another computer. You may assume that the removing and inserting processes take no time.
Note that the batteries cannot be recharged.
Return the maximum number of minutes you can run all the n computers simultaneously.

Example 1:



Input: n = 2, batteries = [3,3,3]
Output: 4
Explanation:
Initially, insert battery 0 into the first computer and battery 1 into the second computer.
After two minutes, remove battery 1 from the second computer and insert battery 2 instead. Note that battery 1 can still run for one minute.
At the end of the third minute, battery 0 is drained, and you need to remove it from the first computer and insert battery 1 instead.
By the end of the fourth minute, battery 1 is also drained, and the first computer is no longer running.
We can run the two computers simultaneously for at most 4 minutes, so we return 4.

Example 2:



Input: n = 2, batteries = [1,1,1,1]
Output: 2
Explanation:
Initially, insert battery 0 into the first computer and battery 2 into the second computer.
After one minute, battery 0 and battery 2 are drained, so you need to remove them and insert battery 1 into the first computer and battery 3 into the second computer.
After another minute, battery 1 and battery 3 are also drained, so the first and second computers are no longer running.
We can run the two computers simultaneously for at most 2 minutes, so we return 2.

Solutions :

Approach 1 - Binary Search

Intuition:
In the question, it is asked to find the maximum number of minutes you can run all the n computers simultaneously.
For a Binary Search type question, we need to observe if the given question asks us to find minimum of maximums or maximums of minimum. If we can identify this in a question, it generally can be solved using Binary Search .
The question stated to us is the maximum time all computers run simultaneously.

For example,
Input: n = 2, batteries = [3,3,3]
Assume : Time in minutes
n computers can run for 0 minutes. (If not possible to run, the lowest value will be zero)
n computers can run for 1 time. (Since our batteries have more than 1 (time) x 2 (computers) charge, we can keep them alive for 1 minute for sure)
n computers can run for 2 minutes.

(This can be computed using basic thinking skills)

We observe that our answer varies in a range. If the number of computers = 1, we need summation of all batteries, to keep it alive for the longest time.
Therefore, low = 0 (minTime kept alive), high = summation of batteries arrays

```
bool isPossible(vector<int>& batteries, long long time, int computers){
        long long totTime = time*computers;

        for(long long bTime : batteries)
            totTime -= min(time, bTime);

        return (totTime <= 0);//if entire charge for all computers is drained, then
we used all batteries, so it's a possible solution
    }
    long long maxRunTime(int n, vector<int>& batteries) {
        long long low = 0, high = 0;
        int si = batteries.size();

        for(int i = 0; i < si; i++){
            high += batteries[i];
        }

        long long ans = 0;
        while(low <= high){

            long long mid = low + (high-low)/2;

            if(isPossible(batteries, mid, n)){//asking for max minutes => increase
the time to check if that's possible
                ans = mid;
                low = mid+1;
            }
            else{
                 high = mid-1;//if this particular time isn't possible, reduce the
range to mid-1
            }
        }

        return ans;
    }
```

2 Problem statement: Minimum size subarray sum (Medium)

Given an array of positive integers nums and a positive integer target, return the minimal length of a subarray whose sum is greater than or equal to target. If there is no such subarray, return 0 instead.

Example 1:

Input: target = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: The subarray [4,3] has the minimal length under the problem constraint.
Example 2:

Input: target = 4, nums = [1,4,4]
Output: 1
Example 3:

Input: target = 11, nums = [1,1,1,1,1,1,1,1]
Output: 0

Solutions :

Approach 1 - Using sorting O(n Log n)

```
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int n = nums.size(), len = INT_MAX;
        vector<int> sums(n + 1, 0);
        for (int i = 1; i <= n; i++) {
            sums[i] = sums[i - 1] + nums[i - 1];
        }
        for (int i = n; i >= 0 && sums[i] >= s; i--) {
            int j = upper_bound(sums.begin(), sums.end(), sums[i] -
s) - sums.begin();
            len = min(len, i - j + 1);
        }
        return len == INT_MAX ? 0 : len;
    }
};
```

Solutions :

Approach 2 - Using sorting O(n Log n)

The O(n) solution is to use two pointers: l and r. First we move r until we get a sum >= s, then we move l to the right until sum < s. In this process, store the minimum length between l and r. Since each element in nums will be visited by l and r for at most once. This algorithm is of O(n) time.

```
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int l = 0, r = 0, n = nums.size(), sum = 0, len = INT_MAX;
        while (r < n) {
            sum += nums[r++];
            while (sum >= s) {
                len = min(len, r - l);
                sum -= nums[l++];
            }
        }
        return len == INT_MAX ? 0 : len;
    }
};
```