Split Array Largest Sum

Given an array which consists of non-negative integers and an integer *m*, you can split the array into *m* non-empty continuous subarrays. Write an algorithm to minimize the largest sum among these *m* subarrays.

Note:
Given *m* satisfies the following constraint: 1 ≤ m ≤ length(nums) ≤ 14,000.

Examples:

```
Input:
nums = [1,2,3,4,5]
m = 2

Output:
9

Explanation:
There are four ways to split nums into two subarrays.
The best way is to split it into [1,2,3] and [4,5],
where the largest sum among the two subarrays is only 9.
```

Solution 1

1. The answer is between maximum value of input array numbers and sum of those numbers.
2. Use binary search to approach the correct answer. We have `l = max number of array; r = sum of all numbers in the array;` Every time we do `mid = (l + r) / 2;`
3. Use greedy to narrow down left and right boundaries in binary search.
   3.1 Cut the array from left.
   3.2 Try our best to make sure that the sum of numbers between each two cuts (inclusive) is large enough but still less than `mid`.
   3.3 We'll end up with two results: either we can divide the array into more than m subarrays or we cannot.
   If we can, it means that the `mid` value we pick is too small because we've already tried our best to make sure each part holds as many non-negative numbers as we can but we still have numbers left. So, it is impossible to cut the array into m parts and make sure each parts is no larger than `mid`. We should increase m. This leads to `l = mid + 1;`
   If we can't, it is either we successfully divide the array into m parts and the sum of each part is less than `mid`, or we used up all numbers before we reach m. Both of them mean that we should lower `mid` because we need to find the minimum one. This leads to `r = mid - 1;`

```
public class Solution {
    public int splitArray(int[] nums, int m) {
        int max = 0; long sum = 0;
        for (int num : nums) {
            max = Math.max(num, max);
            sum += num;
        }
        if (m == 1) return (int)sum;
        //binary search
        long l = max; long r = sum;
        while (l <= r) {
            long mid = (l + r)/ 2;
            if (valid(mid, nums, m)) {
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }
        return (int)l;
    }
    public boolean valid(long target, int[] nums, int m) {
        int count = 1;
        long total = 0;
        for(int num : nums) {
            total += num;
            if (total > target) {
                total = num;
                count++;
                if (count > m) {
                    return false;
                }
            }
        }
        return true;
    }
}
```

- list item

written by dax1ng original link here

Solution 2

1. Given a result, it is easy to test whether it is valid or not.
2. The max of the result is the sum of the input nums.
3. The min of the result is the max num of the input nums.
   Given the 3 conditions above we can do a binary search. (need to deal with overflow)

```java
public class Solution {
    public int splitArray(int[] nums, int m) {
        long sum = 0;
        int max = 0;
        for(int num: nums){
            max = Math.max(max, num);
            sum += num;
        }
        return (int)binary(nums, m, sum, max);
    }

    private long binary(int[] nums, int m, long high, long low){
        long mid = 0;
        while(low < high){
            mid = (high + low)/2;
            if(valid(nums, m, mid)){
                //System.out.println(mid);
                high = mid;
            }else{
                low = mid + 1;
            }
        }
        return high;
    }

    private boolean valid(int[] nums, int m, long max){
        int cur = 0;
        int count = 1;
        for(int num: nums){
            cur += num;
            if(cur > max){
                cur = num;
                count++;
                if(count > m){
                    return false;
                }
            }
        }
        return true;
    }
}
```

written by zrythpzhl original link here

Solution 3

Obviously, the final result is in the interval [left, right] (where left is the maximal number in the array, right is sum of all numbers).
So, what we need to do is to find out the first element in [left, right], which exactly we cannot split the array into m subarrays whose sum is no greater than that element. Then its previous one is the final result. The progress is much similar to lower_bound in C++.

```
class Solution {
public:
    using ll = long long;

    bool canSplit(vector<int>& nums, int m, ll sum) {
        int c = 1;
        ll s = 0;
        for (auto& num : nums) {
            s += num;
            if (s > sum) {
                s = num;
                ++c;
            }
        }
        return c <= m;
    }

    int splitArray(vector<int>& nums, int m) {
        ll left = 0, right = 0;
        for (auto& num : nums) {
            left = max(left, (ll)num);
            right += num;
        }
        while (left <= right) {
            ll mid = left + (right-left)/2;
            if (canSplit(nums, m, mid))
                right = mid-1;
            else
                left = mid+1;
        }
        return left;
    }
};
```

written by vesion original link here