<https://leetcode.com/problems/split-array-largest-sum/description/>

Given an integer array nums and an integer k, split nums into k non-empty subarrays such that the largest sum of any subarray is **minimized**.

Return the minimized largest sum of the split.

A **subarray** is a contiguous part of the array.

**Example 1:**

**Input:** nums = [7,2,5,10,8], k = 2

**Output:** 18

**Explanation:** There are four ways to split nums into two subarrays.The best way is to split it into [7,2,5] and [10,8], where the largest sum among the two subarrays is only 18.

**Example 2:**

**Input:** nums = [1,2,3,4,5], k = 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.

**Constraints:**

1 <= nums.length <= 1000

0 <= nums[i] <= 10^6

1 <= k <= min(50, nums.length)

**Attempt 1: 2024-12-06**

**Solution 1: Binary Search + Greedy (30 min)**

class Solution {

    public int splitArray(int[] nums, int k) {

        int max = 0;

        int sum = 0;

        for(int num : nums) {

            max = Math.max(max, num);

            sum += num;

        }

        // The answer is between maximum value of input array numbers

        // and sum of those numbers. Use binary search to approach the

        // correct answer. We have lo = max number of array; hi = sum of

        // all numbers in the array; Every time we do mid = (lo + hi) / 2;

        int lo = max;

        int hi = sum;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            // Use greedy to narrow down left and right boundaries in binary search.

            if(canSplitArrayWithMidOrNot(nums, k, mid)) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return lo;

    }

    private boolean canSplitArrayWithMidOrNot(int[] nums, int k, int minSubarrayMaxSum) {

        int count = 1;

        long sum = 0;

        for(int num : nums) {

            sum += num;

            if(sum > minSubarrayMaxSum) {

                count++;

                sum = num;

                // If the number of subarrays exceeds k, return false

                if(count > k) {

                    return false;

                }

            }

        }

        return true;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/split-array-largest-sum/solutions/89817/clear-explanation-8ms-binary-search-java/>

The answer is between maximum value of input array numbers and sum of those numbers.

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;

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;

}

}

**Refer to chatGPT**

**Two Key Approaches:**

**1. Binary Search with Greedy Check (Efficient Solution)**

**Idea**:

The largest sum in the optimal split must lie between:

max(nums): The largest single element (minimum possible largest sum).

sum(nums): The sum of all elements (maximum possible largest sum).

Use **binary search** to find the smallest largest sum, and use a **greedy check** to validate the current guess.

class Solution {

public int splitArray(int[] nums, int m) {

int max = 0;

long sum = 0;

// Determine the range for binary search

for (int num : nums) {

max = Math.max(max, num);

sum += num;

}

// Binary search bounds

long left = max, right = sum;

while (left < right) {

long mid = left + (right - left) / 2;

// Check if it's possible to split with largest sum <= mid

if (canSplit(nums, m, mid)) {

right = mid; // Try for a smaller largest sum

} else {

left = mid + 1; // Increase the largest sum

}

}

return (int) left; // The smallest possible largest sum

}

private boolean canSplit(int[] nums, int m, long maxSum) {

int count = 1;

long currentSum = 0;

for (int num : nums) {

if (currentSum + num > maxSum) {

// Create a new subarray

count++;

currentSum = num;

// If the number of subarrays exceeds m, return false

if (count > m) {

return false;

}

} else {

currentSum += num;

}

}

return true;

}

}

**Explanation**:

Binary Search Range:

Start with the range [max(nums), sum(nums)].

Narrow the range by checking if the current mid value is a valid maximum sum for splitting.

Greedy Check (**canSplit)**:

Traverse the array and keep adding to the current subarray sum.

If adding a number exceeds maxSum, start a new subarray and increment the count.

Terminate:

When the binary search completes, left represents the smallest valid largest sum.

**Complexity**:

Time: O(nlog⁡(sum(nums)−max(nums))):

Binary search runs O(log⁡(range)) times.

Each canSplit check runs O(n)

Space: O(1), as no additional space is used.

**使用标准 Find Lower Boundary 的模版如下：**

If you want to use the binary search template provided, you need to adjust the logic slightly because the left <= right condition allows the loop to terminate when left > right, and the smallest valid largest sum will be stored in left at the end of the loop. The key difference is that in this template, you adjust right to mid - 1 and keep track of the smallest possible valid solution.

Here's how your modified binary search solution should look:

class Solution {

public int splitArray(int[] nums, int m) {

int max = 0;

long sum = 0;

// Determine the range for binary search

for (int num : nums) {

max = Math.max(max, num);

sum += num;

}

// Binary search bounds

long left = max, right = sum;

while (left <= right) {

long mid = left + (right - left) / 2;

// Check if it's possible to split with largest sum <= mid

if (canSplit(nums, m, mid)) {

right = mid - 1; // Try for a smaller largest sum

} else {

left = mid + 1; // Increase the largest sum

}

}

return (int) left; // The smallest possible largest sum

}

private boolean canSplit(int[] nums, int m, long maxSum) {

int count = 1;

long currentSum = 0;

for (int num : nums) {

if (currentSum + num > maxSum) {

// Start a new subarray

count++;

currentSum = num;

// If the number of subarrays exceeds m, return false

if (count > m) {

return false;

}

} else {

currentSum += num;

}

}

return true;

}

}

**Key Adjustments:**

Loop Condition:

The loop runs while left <= right instead of left < right.

Binary Search Updates:

If canSplit(nums, m, mid) is true, adjust right = mid - 1 instead of right = mid.

If canSplit(nums, m, mid) is false, adjust left = mid + 1.

Return Statement:

After the loop, left holds the smallest valid largest sum because right will be one step below left when the loop exits.

**Why It Works:**

In this template, the binary search ensures the range [left, right] shrinks until it converges. At termination:

left represents the smallest value for which canSplit returns true.

The adjustments to right ensure that any invalid midpoint (canSplit = false) is excluded from consideration.

This is a direct adaptation of your template to this problem.

**Refer to**

[L1011.Capacity To Ship Packages Within D Days (Ref.L410)](note://WEB87ca8f9dd9a14fb4535856a4e0f8f4b1)

[L704.Binary Search](note://484532A5D2CB4B98A28218674438D5D8)