#### 410. Split Array Largest Sum **Dynamic Programming** Greedy **Array Binary Search** Hard

**Problem Description** 

The goal of this problem is to find a way to divide an array nums into k non-empty contiguous subarrays in such a manner that the

largest sum among those subarrays is as small as possible. In other words, we are trying to balance out the sums of these k subarrays so that no single subarray has a disproportionately large sum compared to the others.

**Prefix Sum** 

When we split the array, every subarray has its own sum. The "largest sum" refers to the sum of subarray with highest total. If we distribute the integers in nums skillfully among the k subarrays, we can minimize this value. The challenge lies in determining the

optimal way to distribute these numbers to achieve our goal. Here's a simple example to illustrate the problem: Let's say nums = [10, 5, 30, 20] and k = 2. A possible way to split this array into two subarrays that minimize the largest sum would be [10, 5] and [30, 20], where the largest sum among subarrays is 50.

using binary search:

Intuition

To solve this problem, we can use binary search. The first step is to determine the search space. We know that the minimum possible value of the largest sum could be the largest number in nums (since each subarray must contain at least one number, and we cannot go lower than the maximum value). The maximum possible value would be the sum of all elements in nums (if we didn't need to split the array at all).

 We create subarrays from the nums array such that each subarray has a sum that does not exceed our chosen middle value (mx). • We then count how many subarrays we form. If we can form at most k subarrays, the middle value (mx) might be too high and we could possibly divide the array in a better way, which means we should search the lower half (reduce mx).

Once we have our search space, we can use binary search to find the minimum possible largest sum. Here's the intuition behind

 Conversely, if we end up with more than k subarrays, that means our middle value (mx) is too low and we have not minimized the sum efficiently.

To find the correct answer, the code uses the bisect\_left function from Python's bisect module, which is a binary search

- We continue the binary search, adjusting our range and mx accordingly, until we find the optimal value. The solution employs a helper function check that uses the current mx to see if we can split the array within the desired constraints.
- Solution Approach
- The solution uses a binary search to iteratively narrow down the search space for the minimum possible largest sum of the k subarrays. This approach is a clever application of binary search on an answer range instead of the traditional usage on sorted

We set a middle value (mx) that could potentially be our minimized largest sum.

implementation that returns the insertion point (as left) to the left side of a sorted array.

# possible sum if all elements were in one subarray).

The check Function

Binary Search

arrays.

whose sums do not exceed mx. If adding an element x causes the current subarray sum to exceed mx, a new subarray starts with x as the first element, incrementing the subarray count cnt. The check function ensures we do not exceed the number of allowed subarrays (k). If cnt surpasses k, it implies that mx is too low

and cannot accommodate the division of nums into k parts. Conversely, if cnt is less than or equal to k, mx could potentially be our

The bisect\_left function from the bisect module uses the check function as a key to perform the binary search. It looks for the

place in the range from left to right where switching check from False to True would occur, which corresponds to the smallest mx

Thus, the binary search continues to narrow the search range by adjusting the left and right pointers based on the result from the

check function. The left pointer is increased if the check function returns True, and the right pointer is decreased if it returns False.

When left and right converge, the left pointer marks the smallest mx that successfully divides nums into k non-empty, contiguous

answer or there might be a lower possible mx. Therefore, check returns a boolean indicating whether the current mx can result in k or

The check function is vital to the binary search. It takes a middle value mx and iterates over nums, creating subarrays of elements

Binary search begins with a left and right pointer, where left is initialized as the maximum value in nums (because this is the

minimum possible sum for the largest subarray), and right is initialized as the sum of all elements in nums (since this is the maximum

### Narrowing the Search Space

that allows for the division into k or fewer subarrays.

ascertain the optimal way to split the array into k parts.

subarrays while minimizing the largest sum of any subarray. This is our desired answer.

fewer subarrays.

**Time Complexity** The time complexity of this solution is 0(n log(sum(nums)-max(nums))) since the binary search requires 0(log(sum(nums)-

In summary, this solution approach efficiently applies binary search by utilizing the bisect module and a custom condition to

Let us consider an example where nums = [7, 2, 5, 10, 8] and k = 2. Our goal is to divide the array nums into k contiguous

max(nums))) iterations, and during each iteration, the entire nums array of size n is traversed by the check function.

Step 2: Binary Search Execution Now we execute the binary search with our left at 10 and our right at 32.

We've successfully created 2 subarrays without exceeding the middle value 21 which matches our k value.

• [10] starts a new subarray, and adding 8 would again exceed 15, so 10 remains its own subarray.

• [10, 8] can be the next subarray with a sum of 18 which matches our new middle value mx.

**Iteration 2:** The new mx becomes (10 + 21) / 2 = 15.5, we'll use 15 for simplicity's sake.

• [7, 2, 5] which has a sum of 14. Adding 10 would exceed 15, so we stop here.

Reapplying the check function with mx=15, we get the following subarrays:

**Iteration 1: Middle Value Calculation** We take the middle value mx = (left + right) / 2, which is (10 + 32) / 2 = 21.

subarrays such that the largest sum among these subarrays is minimized. Step 1: Initialize Binary Search Parameters We set left equal to the maximum value in nums, which is 10 in this case, and right equal to the sum of all elements in nums, which sums up to 32.

Step 3: The check Function Execution Using the check function with mx=21, we start summing values from nums to form subarrays

#### We start with 7, and then add 2 for a subtotal of 9. We can also add 5 and reach 14 which still is less than 21, but adding 10 would exceed our middle value mx. Therefore, we form a subarray [7, 2, 5] with sum 14 and start a new subarray with 10. Now 10 and 8

where the sum does not exceed 21.

form the second subarray with sum 18.

With mx=18, we test the check function again:

subarray.

lower our right bound to 18.

result in 2 subarrays.

the subarrays is 17.

Python Solution

1 from typing import List

from math import inf

class Solution:

8

9

10

11

12

13

14

15

16

23

24

25

26

27

28

29

30

9

10

11

12

13

14

15

16

17

18

19

20

21

23

24

25

26

27

28

29

30

11

12

13

14

16

17

18

20

23

24

25

26

27

28

29

30

31

32

34

35

36

37

38

39

40

41

42

43

44

45

46

47

49

48 };

8

9

10

11

12

13

14

15

16

18

19

20

**}**;

from bisect import bisect\_left

def is\_feasible(max\_sum):

for number in nums:

return splits <= k</pre>

return smallest\_max\_sum

current\_sum, splits = 0, 0

current\_sum += number

splits += 1

# such that is\_feasible returns True

int maxElement = 0, sumRange = 0;

while (maxElement < sumRange) {</pre>

int splitArray(vector<int>& nums, int k) {

minMax = max(minMax, num);

auto canSplit = [&](int maxSum) {

cumulativeSum += num;

int mid = (minMax + sum) / 2;

// Otherwise, search in the right half.

function splitArray(nums: number[], maxSubarrays: number): number {

const canSplitWithMaxSum = (maxSum: number) : boolean => {

let subarraySum = 0; // Initialize current subarray sum

for (int num : nums) {

++parts;

return parts <= k;</pre>

if (canSplit(mid)) {

minMax = mid + 1;

sum = mid;

while (minMax < sum) {</pre>

else {

return minMax;

totalSum += num;

for (const num of nums) {

subarraySum += num;

if (subarraySum > maxSum) {

elements and Max is the maximum element in the list.

range of ints and not on additional data structures.

Typescript Solution

int cumulativeSum = 0, parts = 1;

if (cumulativeSum > maxSum) {

// Add current number to the cumulative sum.

cumulativeSum = num; // start a new part

// If the cumulative sum exceeds maxSum, we need a new part.

// If the number of parts required is less than or equal to k, return true.

// 'minMax' is now the minimum largest sum to split the array into at most k parts.

// Helper function to determine if the current maximum subarray sum 'maxSum' is valid

// Check if the array can be split into subarrays with the sum not exceeding 'maxSum'

// If adding the current element exceeds 'maxSum', start a new subarray

let subarrayCount = 1; // Initialize the subarray count, start with 1

int minMax = 0, sum = 0;

for (int num : nums) {

sum += num;

// Initialize the bounds for binary search.

for (int num : nums) {

} else {

return maxElement;

sumRange += num;

// Calculate the maxElement and sumRange

maxElement = Math.max(maxElement, num);

if (isSplitPossible(nums, mid, maxSplits)) {

if current\_sum > max\_sum:

current\_sum = number

Example Walkthrough

possible to have the largest sum as 21 or maybe even smaller. Therefore, we bring down our right bound to 21. Now left is 10 and right is 21. We repeat the binary search process.

Step 4: Adjusting Binary Search Bounds Since we can create 2 subarrays that do not exceed the sum of 21 (k=2), we know that it is

• [8] is forced to be a subarray on its own because adding it to any previous subarray would exceed 15. We now have 3 subarrays which exceed our k value. Therefore, the mx of 15 is too low. We need to increase our left bound to 16. **Iteration 3:** We now have left as 16 and right as 21. New mx is 18.

• [7, 2, 5] has a sum of 14. We can add 10, but the sum would be 24, exceeding 18. So again, [7, 2, 5] becomes the first

Now we have 2 subarrays, which is equal to our desired k value. As this is valid and potentially still not optimal (it could be lower), we

With the left and right bounds converging and achieving the required k subarrays, we find that our minimized largest sum across

Step 5: Convergence of Binary Search Now left is 16 and right is 18. Our new mx is 17. Applying the check with mx=17 would also

In this example, subarrays [7, 2, 5] and [10, 8] represent the division of nums with a minimized largest sum, which is 17. This binary search process ensures that we explore all possible configurations efficiently by focusing only on valid ranges defined by the check function's results.

17 # Set initial bounds for binary search 18 19 # left is the maximum value (since we cannot have a smaller sum than a single element) 20 # right is the sum of all the numbers (if we put all nums in one subarray) 21 left, right = max(nums), sum(nums) 22

# Helper function to check if a given max array sum is feasible by counting subarray splits

# If current subarray sum exceeds max\_sum, we need a new subarray

smallest\_max\_sum = left + bisect\_left(range(left, right + 1), True, key=is\_feasible)

# Return the smallest feasible maximum sum for splitting nums into k or fewer subarrays

// Initialize 'maxElement' to the largest number in the array to set the minimum possible sum,

int mid = (maxElement + sumRange) >> 1; // Find the mid value between maxElement and sumRange

// Check if the current mid can be the maximum subarray sum with at most maxSplits splits

maxElement = mid + 1; // Otherwise, increase the subarray sum and check again

int subarraySum = (1 << 30), subarrayCount = 0; // Initialize subarraySum with a large value</pre>

sumRange = mid; // If it's possible, try to find a smaller subarray sum

// 'sumRange' to the total sum of the array to set the maximum possible sum

// Binary search between maxElement and sumRange to find the minimum largest sum

// maxElement now represents the minimum largest sum for the subarrays

private boolean isSplitPossible(int[] nums, int maxSubarraySum, int maxSplits) {

# Perform a binary search to find the smallest feasible maximum sum

# We use bisect\_left to find the first value in range(left, right+1)

This walkthrough illustrates the power of binary search in optimizing a search space to find the minimum possible largest sum for k contiguous subarrays in a given nums array.

def splitArray(self, nums: List[int], k: int) -> int:

Java Solution 1 class Solution { public int splitArray(int[] nums, int maxSplits) {

```
31
32
           // Iterate over the numbers and try to split the array into subarrays that do not exceed maxSubarraySum
           for (int num : nums) {
33
34
               subarraySum += num;
35
36
               // When current subarraySum exceeds maxSubarraySum, increment subarrayCount
37
               // and start a new subarray with the current number
               if (subarraySum > maxSubarraySum) {
38
39
                   subarrayCount++;
40
                   subarraySum = num;
41
42
43
44
           // Check if the number of splits required is less than or equal to the allowed maxSplits
45
           return subarrayCount <= maxSplits;</pre>
48
C++ Solution
1 #include <vector>
2 #include <algorithm>
   using namespace std;
   class Solution {
   public:
```

// Calculate the largest number in nums as the lower bound and the sum for the upper bound.

// Lambda function to check if it's possible to split the array into at most 'k' parts with max sum 'maxSum'.

// Binary search to find the minimum largest sum with which we can split the array into at most k parts.

// If we can split the array into at most k parts with max sum 'mid', search in the left half.

```
let maxElement = 0; // Initialize the max element in the array
      let totalSum = 0; // Initialize the total sum of the array
5
      // Calculate the max element and total sum of the array
      for (const num of nums) {
          maxElement = Math.max(maxElement, num);
```

```
21
                     subarraySum = num; // Current number becomes the start of the new subarray
 22
                     subarrayCount++; // Increment the number of subarrays
 23
 24
 25
             // Return true if the required number of subarrays is less than or equal to 'maxSubarrays'
 26
             return subarrayCount <= maxSubarrays;</pre>
 27
         };
 28
 29
         // Binary search to find the minimum possible maximum subarray sum
 30
         let start = maxElement; // Lower bound for the maximum subarray sum
 31
         let end = totalSum; // Upper bound for the maximum subarray sum
 32
 33
         // Perform binary search
 34
         while (start < end) {</pre>
             const mid = start + ((end - start) >> 1); // Calculate mid avoiding possible overflow
 35
 36
 37
             // If it's possible to split with 'mid' as the maximum subarray sum
 38
             if (canSplitWithMaxSum(mid)) {
                 end = mid; // We might find a smaller maximum sum, search the left half
 39
 40
             } else {
 41
                 start = mid + 1; // Search the right half
 42
 43
 44
 45
         // Once binary search is complete, 'start' will contain the minimum possible maximum sum
 46
         return start;
 47 }
 48
Time and Space Complexity
Time Complexity
The time complexity of the splitArray function is determined by the binary search it performs and the check function it uses to
```

## • Check Function: For each iteration of binary search, the check function is called, which iterates over all elements of the nums list to verify if the current maximum sum split (mx) is valid. The iteration over all n elements results in a time complexity of O(n) for

the check function.

validate splits.

binary search. **Space Complexity** 

the algorithm. The binary search itself also doesn't consume additional space that scales with the input size, as it works on the

Combining these two gives us an overall time complexity of 0(n \* log(Sum-Max)), as the check function is called for each step of the

• Binary Search: The binary search is performed on a range from left (which is the maximum element in nums) to right which is

the sum of all elements in nums. The number of iterations in binary search is O(log(Sum-Max)) where Sum is the sum of all

• The space complexity of the function is 0(1) not including the input. This is because the check function uses only a constant amount of extra space (s and cnt) and the space used by the inputs or outputs is not counted towards the space complexity of