410. Split Array Largest Sum

Hard | Greedy | Array | Binary Search | Dynamic Programming | Prefix Sum | Leetcode Link

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.

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.

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).

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

- We set a middle value (`mx`) that could potentially be our minimized largest sum.
- 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`).
- 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.
- 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. To find the correct answer, the code uses the `bisect_left` function from Python's `bisect` module, which is a binary search implementation that returns the insertion point (as `left`) to the left side of a sorted array.

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 arrays.

Binary Search

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 possible sum if all elements were in one subarray).

The `check` Function

The `check` function is vital to the binary search. It takes a middle value `mx` and iterates over `nums`, creating subarrays of elements 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 answer or there might be a lower possible `mx`. Therefore, `check` returns a boolean indicating whether the current `mx` can result in `k` or fewer subarrays.

Narrowing the Search Space

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` that allows for the division into `k` or fewer subarrays.

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

Time Complexity

The time complexity of this solution is `O(n log(sum(nums)-max(nums)))` since the binary search requires `O(log(sum(nums)-max(nums)))` iterations, and during each iteration, the entire `nums` array of size `n` is traversed by the `check` function.

In summary, this solution approach efficiently applies binary search by utilizing the `bisect` module and a custom condition to ascertain the optimal way to split the array into `k` parts.

Example Walkthrough

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

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

Step 3: The `check` Function Execution Using the `check` function with `mx=21`, we start summing values from `nums` to form subarrays where the sum does not exceed `21`.

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` form the second subarray with sum `18`.

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

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 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.

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

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

- `[7, 2, 5]` which has a sum of `14`. Adding `10` would exceed `15`, so we stop here.
- `[10]` starts a new subarray, and adding `8` would again exceed `15`, so `10` remains its own subarray.
- `[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`.

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

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

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 lower our `right` bound to `18`.

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 result in `2` subarrays.

With the `left` and `right` bounds converging and achieving the required `k` subarrays, we find that our minimized largest sum across the subarrays is `17`.

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.

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.

Python Solution

```python
from typing import List
from bisect import bisect_left
from math import inf

class Solution:
    def splitArray(self, nums: List[int], k: int) -> int:
        # Helper function to check if a given max array sum is feasible by counting subarray splits
        def is_feasible(max_sum):
            current_sum, splits = 0, 0
            for number in nums:
                current_sum += number
                # If current subarray sum exceeds max_sum, we need a new subarray
                if current_sum > max_sum:
                    current_sum = number
                    splits += 1
            return splits <= k

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

        # 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)
        # such that is_feasible returns True
        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
        return smallest_max_sum
```

Java Solution

```java
class Solution {
    public int splitArray(int[] nums, int maxSplits) {
        // Initialize 'maxElement' to the largest number in the array to set the minimum possible sum,
        // 'sumRange' to the total sum of the array to set the maximum possible sum
        int maxElement = 0, sumRange = 0;

        // Calculate the maxElement and sumRange
        for (int num : nums) {
            maxElement = Math.max(maxElement, num);
            sumRange += num;
        }

        // Binary search between maxElement and sumRange to find the minimum largest sum
        while (maxElement < sumRange) {
            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
            if (isSplitPossible(nums, mid, maxSplits)) {
                sumRange = mid; // If it's possible, try to find a smaller subarray sum
            } else {
                maxElement = mid + 1; // Otherwise, increase the subarray sum and check again
            }
        }

        // maxElement now represents the minimum largest sum for the subarrays
        return maxElement;
    }

    private boolean isSplitPossible(int[] nums, int maxSubarraySum, int maxSplits) {
        int subarraySum = (1 << 30), subarrayCount = 0; // Initialize subarraySum with a large value

        // Iterate over the numbers and try to split the array into subarrays that do not exceed maxSubarraySum
        for (int num : nums) {
            subarraySum += num;

            // When current subarraySum exceeds maxSubarraySum, increment subarrayCount
            // and start a new subarray with the current number
            if (subarraySum > maxSubarraySum) {
                subarrayCount++;
                subarraySum = num;
            }
        }

        // Check if the number of splits required is less than or equal to the allowed maxSplits
        return subarrayCount <= maxSplits;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    int splitArray(vector<int>& nums, int k) {
        // Initialize the bounds for binary search.
        int minMax = 0, sum = 0;
        // Calculate the largest number in nums as the lower bound and the sum for the upper bound.
        for (int num : nums) {
            minMax = max(minMax, num);
            sum += num;
        }

        // Lambda function to check if it's possible to split the array into at most 'k' parts with max sum 'maxSum'.
        auto canSplit = [&](int maxSum) {
            int cumulativeSum = 0, parts = 1;
            for (int num : nums) {
                // Add current number to the cumulative sum.
                cumulativeSum += num;
                // If the cumulative sum exceeds maxSum, we need a new part.
                if (cumulativeSum > maxSum) {
                    cumulativeSum = num; // start a new part
                    ++parts;
                }
            }
            // If the number of parts required is less than or equal to k, return true.
            return parts <= k;
        };

        // Binary search to find the minimum largest sum with which we can split the array into at most k parts.
        while (minMax < sum) {
            int mid = (minMax + sum) / 2;
            // If we can split the array into at most k parts with max sum 'mid', search in the left half.
            if (canSplit(mid)) {
                sum = mid;
            }
            // Otherwise, search in the right half.
            else {
                minMax = mid + 1;
            }
        }

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

Typescript Solution

```typescript
function splitArray(nums: number[], maxSubarrays: number): number {
    let maxElement = 0; // Initialize the max element in the array
    let totalSum = 0; // Initialize the total sum of the array

    // Calculate the max element and total sum of the array
    for (const num of nums) {
        maxElement = Math.max(maxElement, num);
        totalSum += num;
    }

    // Helper function to determine if the current maximum subarray sum 'maxSum' is valid
    const canSplitWithMaxSum = (maxSum: number) : boolean => {
        let subarraySum = 0; // Initialize current subarray sum
        let subarrayCount = 1; // Initialize the subarray count, start with 1

        // Check if the array can be split into subarrays with the sum not exceeding 'maxSum'
        for (const num of nums) {
            subarraySum += num;
            // If adding the current element exceeds 'maxSum', start a new subarray
            if (subarraySum > maxSum) {
                subarraySum = num; // Current number becomes the start of the new subarray
                subarrayCount++; // Increment the number of subarrays
            }
        }
        // Return true if the required number of subarrays is less than or equal to 'maxSubarrays'
        return subarrayCount <= maxSubarrays;
    };

    // Binary search to find the minimum possible maximum subarray sum
    let start = maxElement; // Lower bound for the maximum subarray sum
    let end = totalSum; // Upper bound for the maximum subarray sum

    // Perform binary search
    while (start < end) {
        const mid = start + ((end - start) >> 1); // Calculate mid avoiding possible overflow

        // If it's possible to split with 'mid' as the maximum subarray sum
        if (canSplitWithMaxSum(mid)) {
            end = mid; // We might find a smaller maximum sum, search the left half
        } else {
            start = mid + 1; // Search the right half
        }
    }

    // Once binary search is complete, 'start' will contain the minimum possible maximum sum
    return start;
}
```

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 validate splits.

- **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 elements and `Max` is the maximum element in the list.

- **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.

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

Space Complexity

- The space complexity of the function is `O(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 the algorithm. The binary search itself also doesn't consume additional space that scales with the input size, as it works on the range of ints and not on additional data structures.