Split Array Largest Sum

1. Problem Statement

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.

Constraints:

- `1 <= nums.length <= 1000`
- `0 <= nums[i] <=` 10^6
- `1 <= k <= min(50, nums.length)`

Example 1:

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

Output: `18`

Explanation: The best split is `[7,2,5]` and `[10,8]`, with a maximum subarray sum of `18`.

Example 2:

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

Output: `9`

Explanation: The best split is `[1,2,3]` and `[4,5]`, with a maximum subarray sum of `9`.

---

2. Intuition

To minimize the largest sum of the subarrays, we need to balance the sum across `k` subarrays. A **binary search** on the possible largest sum is an efficient approach because it allows us to check whether a given maximum sum is feasible by dividing `nums` into at most `k` subarrays.

---

3. Key Observations

1. The **minimum** possible largest sum is `max(nums)` because at least one subarray must contain the largest element.
2. The **maximum** possible largest sum is `sum(nums)`, which happens when all elements are in a single subarray.
3. The problem can be reduced to a **decision problem**:

- "Can we split `nums` into `k` subarrays such that the largest sum does not exceed `mid`?"
- If **yes**, try a smaller `mid`.
- If **no**, increase `mid`.

---

4. Approach

We use **binary search** on the possible largest sum, and for each candidate sum (`mid`), we use a **greedy check** to verify if we can partition `nums` into at most `k` subarrays without exceeding `mid`.

Steps:

1. **Initialize binary search range:**
   - `left = max(nums)`, since we must include the largest element in a subarray.
   - `right = sum(nums)`, since the whole array can be one subarray.
2. **Binary Search:**
   - Compute `mid = (left + right) // 2`.
   - Check if `nums` can be split into at most `k` subarrays with `mid` as the largest sum.
   - If valid, try for a smaller `mid` (`right = mid`).
   - If not, increase `mid` (`left = mid + 1`).
3. **Greedy Validation:**
   - Traverse `nums`, maintaining `current_sum`.
   - If adding an element exceeds `mid`, start a new subarray.
   - If the number of subarrays exceeds `k`, return `False`.
   - Otherwise, return `True`.
4. **Return `left` as the answer**, since it holds the minimized largest sum.

---

5. Edge Cases

- **Single Element Array:** `nums = [10], k = 1` → Output `10`.
- **All Elements Same:** `nums = [5,5,5,5], k = 2` → Output `10`.
- **Maximum Constraints:** Large `nums` and `k` values to test efficiency.
- **Already Balanced:** `nums = [1,1,1,1], k = 4` → Output `1`.
- **Minimum `k = 1`:** Entire array is one subarray → Output `sum(nums)`.
- **Maximum `k = len(nums)`:** Each element is its own subarray → Output `max(nums)`.

---

6. Complexity Analysis

Time Complexity:

- Binary search runs in `O(log(sum(nums) - max(nums)))`.
- Greedy validation runs in `O(n)`.
- Total complexity: **`O(n log(sum(nums) - max(nums)))`**.

Space Complexity:

- **`O(1)`**, as we use only a few extra variables.

---

7. Alternative Approaches

1. Brute Force (Exponential Search)

- Try all possible ways to split `nums` into `k` subarrays.
- Time complexity: **`O(2^n)`** (too slow).

2. Dynamic Programming

- Use `dp[i][j]` to store the minimum largest sum for `i` elements split into `j` subarrays.
- Time complexity: **`O(n^2 * k)`** (better but still slow for large `n`).

---

8. Code Implementation

```
from typing import List

class Solution:
    def splitArray(self, nums: List[int], k: int) -> int:
        def is_valid(mid):
            subarrays = 1
            current_sum = 0
            for num in nums:
                if current_sum + num > mid:
                    subarrays += 1
                    current_sum = num
                    if subarrays > k:
                        return False
                else:
                    current_sum += num
            return True

        left, right = max(nums), sum(nums)
        while left < right:
            mid = (left + right) // 2
            if is_valid(mid):
                right = mid
            else:
                left = mid + 1

        return left
```

---

9. Test Cases

```
solution = Solution()

# Test Case 1
```

```
print(solution.splitArray([7,2,5,10,8], 2))  # Output: 18

# Test Case 2
print(solution.splitArray([1,2,3,4,5], 2))   # Output: 9

# Test Case 3
print(solution.splitArray([1,4,4], 3))      # Output: 4

# Edge Case: Large `k`
print(solution.splitArray([1,2,3,4,5], 5))  # Output: 5
```

---

10. Final Thoughts

- **Binary Search + Greedy** provides an optimal and efficient solution.
- The **key idea** is to minimize the largest sum by adjusting the possible range using binary search.
- The **greedy validation** ensures we do not exceed `k` subarrays.
- This method is **efficient** compared to brute force and dynamic programming.