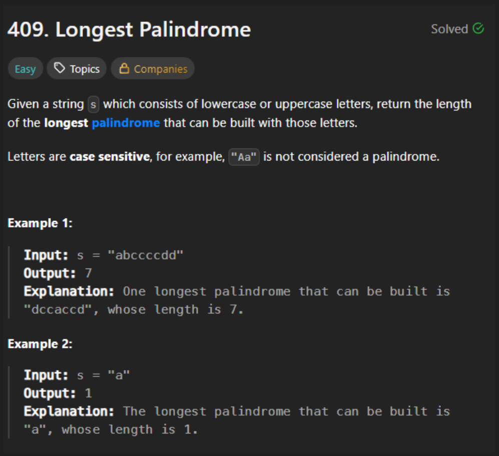
Weekly Contest - 7



409. Longest Palindrome

Solved ✓

Easy · Topics · Companies

Given a string `s` which consists of lowercase or uppercase letters, return the length of the **longest** palindrome that can be built with those letters.

Letters are **case sensitive**, for example, `"Aa"` is not considered a palindrome.

Example 1:

```
Input: s = "abccccdd"
Output: 7
Explanation: One longest palindrome that can be built is
"dccaccd", whose length is 7.
```

Example 2:

```
Input: s = "a"
Output: 1
Explanation: The longest palindrome that can be built is
"a", whose length is 1.
```

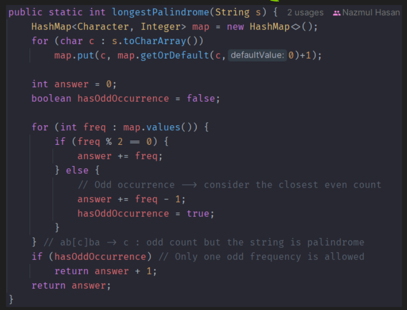
Store the frequency of each character

A valid palindrome has - only one char of odd freq

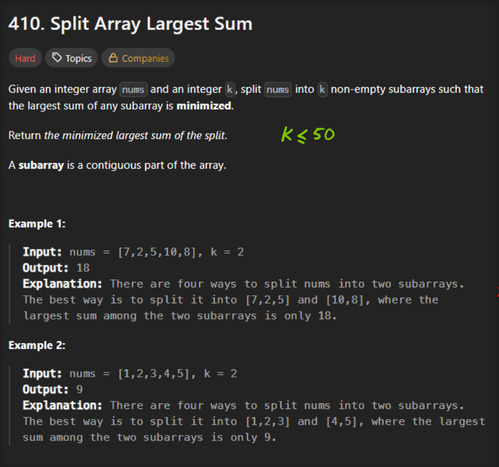
if freq is even -

Add one at the start

Another at the end



```java
public static int longestPalindrome(String s) {  2 usages  Nazmul Hasan
    HashMap<Character, Integer> map = new HashMap<>();
    for (char c : s.toCharArray())
        map.put(c, map.getOrDefault(c, defaultValue: 0)+1);

    int answer = 0;
    boolean hasOddOccurrence = false;

    for (int freq : map.values()) {
        if (freq % 2 == 0) {
            answer += freq;
        } else {
            // Odd occurrence ⟶ consider the closest even count
            answer += freq - 1;
            hasOddOccurrence = true;
        }
    } // ab[c]ba → c : odd count but the string is palindrome
    if (hasOddOccurrence) // Only one odd frequency is allowed
        return answer + 1;
    return answer;
}
```



410. Split Array Largest Sum

Hard · Topics · Companies

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.*

$k \leq 50$

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.
```

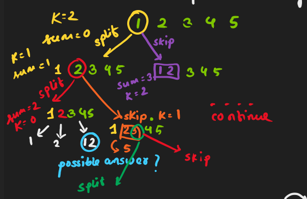
Dynamic Programming Approach

	1	2	3	4	5
leftsum	1	3	6	10	15
Right sum	15	14	12	9	5

Now we can follow the 0/1 Knapsack problem

for K>0, at each index

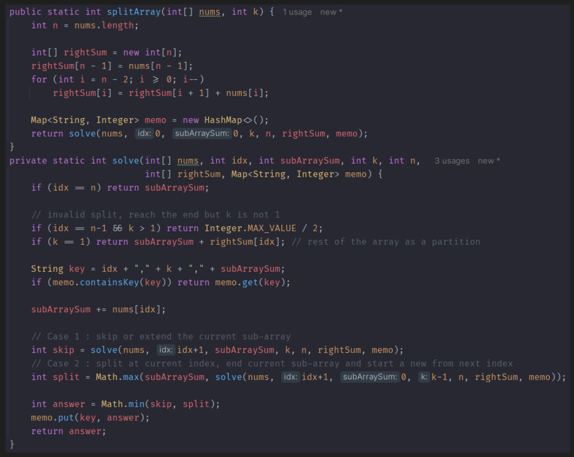
we can split the array or skip it



Case 1 → skip, Extend the subarray, sum += num

2 → split, Start new subarray, sum = 0

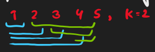
TLE

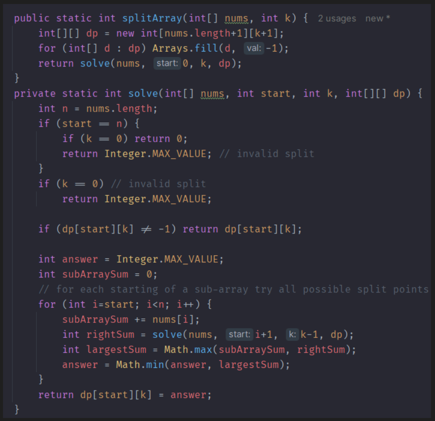


```java
public static int splitArray(int[] nums, int k) {  1 usage  new *
    int n = nums.length;

    int[] rightSum = new int[n];
    rightSum[n - 1] = nums[n - 1];
    for (int i = n - 2; i ≥ 0; i--)
        rightSum[i] = rightSum[i + 1] + nums[i];

    Map<String, Integer> memo = new HashMap<>();
    return solve(nums, idx: 0, subArraySum: 0, k, n, rightSum, memo);
}
private static int solve(int[] nums, int idx, int subArraySum, int k, int n,   3 usages  new *
                         int[] rightSum, Map<String, Integer> memo) {
    if (idx == n) return subArraySum;

    // invalid split, reach the end but k is not 1
    if (idx == n-1 && k > 1) return Integer.MAX_VALUE / 2;
    if (k == 1) return subArraySum + rightSum[idx]; // rest of the array as a partition

    String key = idx + "," + k + "," + subArraySum;
    if (memo.containsKey(key)) return memo.get(key);

    subArraySum += nums[idx];

    // Case 1 : skip or extend the current sub-array
    int skip = solve(nums, idx: idx+1, subArraySum, k, n, rightSum, memo);
    // Case 2 : split at current index, end current sub-array and start a new from next index
    int split = Math.max(subArraySum, solve(nums, idx: idx+1, subArraySum: 0, k: k-1, n, rightSum, memo));

    int answer = Math.min(skip, split);
    memo.put(key, answer);
    return answer;
}
```

A better Approach → DP

For each starting position check every possible split indices





```java
public static int splitArray(int[] nums, int k) {  2 usages  new *
    int[][] dp = new int[nums.length+1][k+1];
    for (int[] d : dp) Arrays.fill(d, val: -1);
    return solve(nums, start: 0, k, dp);
}
private static int solve(int[] nums, int start, int k, int[][] dp) {
    int n = nums.length;
    if (start == n) {
        if (k == 0) return 0;
        return Integer.MAX_VALUE; // invalid split
    }
    if (k == 0) // invalid split
        return Integer.MAX_VALUE;

    if (dp[start][k] ≠ -1) return dp[start][k];

    int answer = Integer.MAX_VALUE;
    int subArraySum = 0;
    // for each starting of a sub-array try all possible split points
    for (int i=start; i<n; i++) {
        subArraySum += nums[i];
        int rightSum = solve(nums, start: i+1, k: k-1, dp);
        int largestSum = Math.max(subArraySum, rightSum);
        answer = Math.min(answer, largestSum);
    }
    return dp[start][k] = answer;
}
```

Greedy + Binary Search

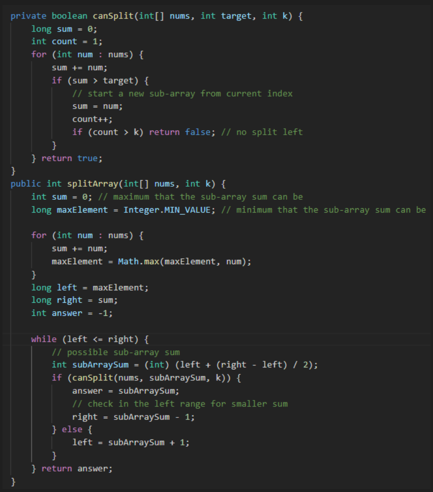
Minimum sum ~~of~~ the subarray ?

→ Max of array — ①

Maximum? < sum of the array — ②

Binary Search in range ① ~ ②

then check if it is a valid split



```java
private boolean canSplit(int[] nums, int target, int k) {
    long sum = 0;
    int count = 1;
    for (int num : nums) {
        sum += num;
        if (sum > target) {
            // start a new sub-array from current index
            sum = num;
            count++;
            if (count > k) return false; // no split left
        }
    } return true;
}
public int splitArray(int[] nums, int k) {
    int sum = 0; // maximum that the sub-array sum can be
    long maxElement = Integer.MIN_VALUE; // minimum that the sub-array sum can be

    for (int num : nums) {
        sum += num;
        maxElement = Math.max(maxElement, num);
    }
    long left = maxElement;
    long right = sum;
    int answer = -1;

    while (left <= right) {
        // possible sub-array sum
        int subArraySum = (int) (left + (right - left) / 2);
        if (canSplit(nums, subArraySum, k)) {
            answer = subArraySum;
            // check in the left range for smaller sum
            right = subArraySum - 1;
        } else {
            left = subArraySum + 1;
        }
    } return answer;
}
```