Worksheet 2

Student Name: Mayank Vashishta **UID:** 22CS12920

Branch:CSE **Section/Group:**605-B

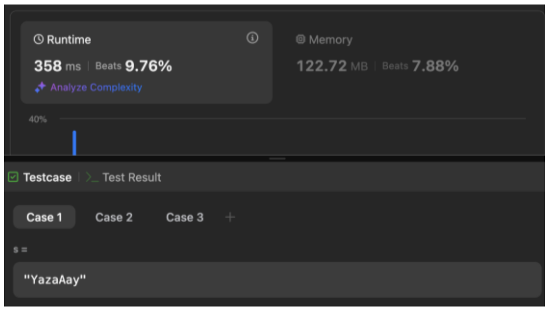
Semester: 5 **Date of Performance:** 5 /02/25

Subject Name: AP **Subject Code: 22CSP-351**

1. A string s is nice if, for every letter of the alphabet that s contains, it appears both in uppercase and lowercase. For example, "abABB" is nice because 'A' and 'a' appear, and 'B' and 'b' appear. However, "abA" is not because 'b' appears, but 'B' does not.

   Given a string s, return the longest substring of s that is nice. If there are multiple, return the substring of the earliest occurrence. If there are none, return an empty string.

```
class Solution {

public:

   bool isNice(const string& str) {

   unordered_set<char> charSet(str.begin(), str.end());

   for (char ch : str) {

      if (charSet.count(tolower(ch)) == 0 || charSet.count(toupper(ch)) == 0) {

         return false;

      }

   }

   return true;

}
string longestNiceSubstring(string s) {
    int maxLength = 0;
  string result = "";

  for (int i = 0; i < s.length(); ++i) {
    for (int j = i + 1; j <= s.length(); ++j) {
      string substring = s.substr(i, j - i);
      if (isNice(substring) && substring.length() > maxLength) {
        maxLength = substring.length();
        result = substring;
      }
    }
  }
  return result;
  }
};
```



358 ms | Beats 9.76%

122.72 MB | Beats 7.88%

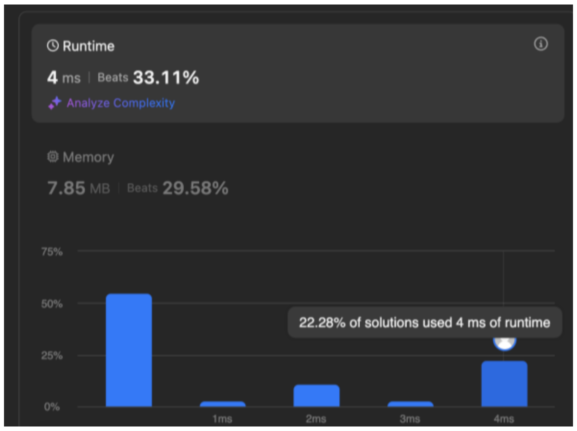
s =

"YazaAay"

2. Reverse bits of a given 32 bits unsigned integer. Note: Note that in some languages, such as Java,

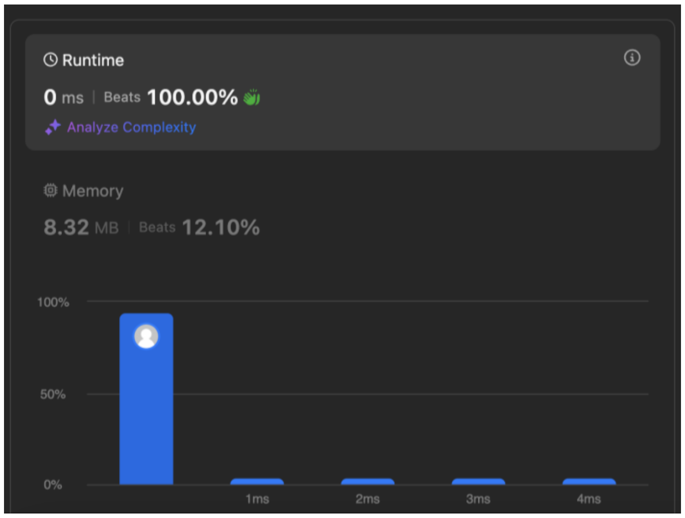
there is no unsigned integer type. In this case, both input and output will be given as a signed integer type. They should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned. In Java, the compiler represents the signed integers using 2's complement notation. Therefore, in Example 2 above, the input represents the signed integer -3 and the output represents the signed integer -1073741825.

```
uint32_t reverseBits(uint32_t n) {

    uint32_t result = 0;

    for (int i = 0; i < 32; ++i) {

        result = (result << 1) | (n & 1);  // Shift result to the left and add the last bit of n

        n >>= 1;                           // Shift n to the right to process the next bit

    }

    return result;

}
```


Runtime
4 ms | Beats 33.11%
Analyze Complexity
Memory
7.85 MB | Beats 29.58%
22.28% of solutions used 4 ms of runtime

3. Given a positive integer n, write a function that returns the number of set bits in its binary representation (also known as the Hamming weight).

```
int hammingWeight(uint32_t n) {

    int count = 0;

    while (n != 0) {

        count += (n & 1);  // Check the last bit of n
        n >>= 1;          // Right shift n by 1 to check the next bit
    }
    return count;
    }
```


Runtime

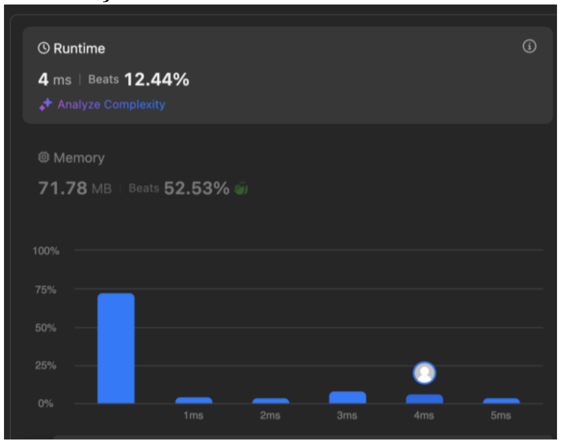
0 ms | Beats 100.00%

Memory

8.32 MB | Beats 12.10%

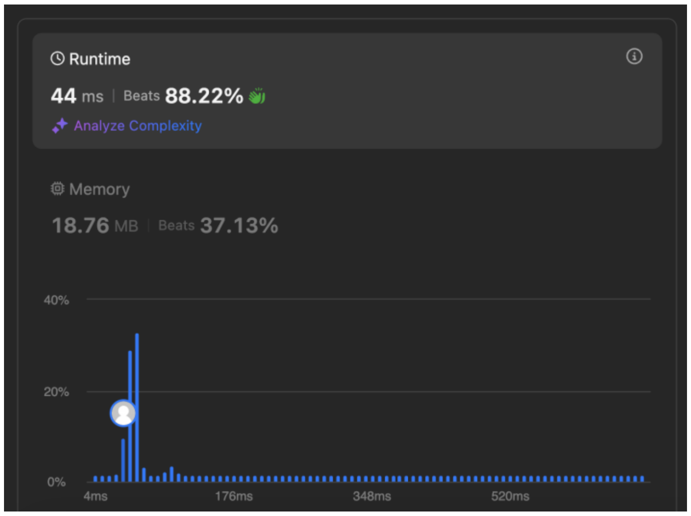
4. Given an integer array nums, find the subarray with the largest sum, and return its sum.

```
int maxSubArray(vector<int>& nums) {
    int maxSum = nums[0];
    int currentSum = nums[0];

    for (int i = 1; i < nums.size(); ++i) {
        currentSum = max(nums[i], currentSum + nums[i]);  // Extend the subarray or start a new one
        maxSum = max(maxSum, currentSum);              // Update the max sum
    }

    return maxSum;
    }
```



5. Write an efficient algorithm that searches for a value target in an m x n integer matrix matrix. This matrix has the following properties: Integers in each row are sorted in ascending from left to right. Integers in each column are sorted in ascending from top to bottom.

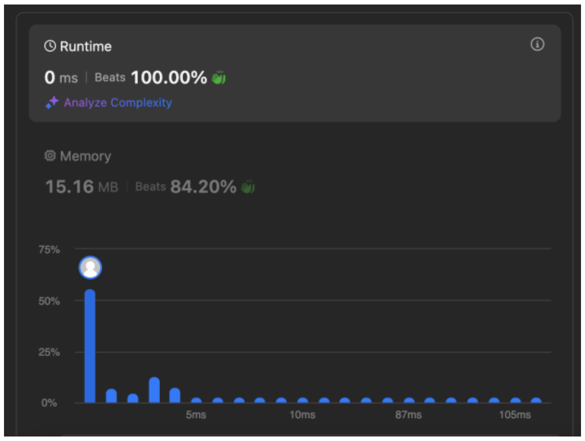
```
bool searchMatrix(vector<vector<int>>& matrix, int target) {

    if (matrix.empty() || matrix[0].empty()) return false;



    int rows = matrix.size();

    int cols = matrix[0].size();

    int row = 0, col = cols - 1;  // Start from the top-right corner



    while (row < rows && col >= 0) {

        if (matrix[row][col] == target) {

            return true;  // Target found

        } else if (matrix[row][col] > target) {

            col--;  // Move left

        } else {

            row++;  // Move down
        }
    }

    return false;  // Target not found
}
```


Runtime
44 ms | Beats 88.22%
Analyze Complexity

Memory
18.76 MB | Beats 37.13%

6. Your task is to calculate ab mod 1337 where a is a positive integer and b is an extremely large positive integer given in the form of an array.

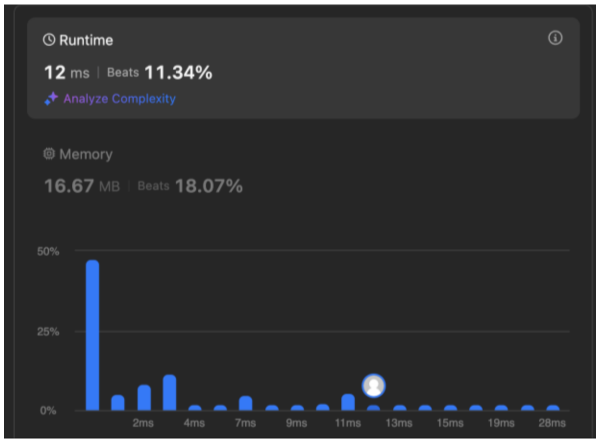
```
int modPow(int a, int b, int mod) {
    int result = 1;
    a %= mod;
    while (b > 0) {
        if (b % 2 == 1) {
            result = (result * a) % mod;
        }
        a = (a * a) % mod;
        b /= 2;
    }
    return result;
}

// Function to calculate a^b % 1337 where b is given as a vector of digits
int superPow(int a, vector<int>& b) {
    if (b.empty()) return 1;

    int lastDigit = b.back();
    b.pop_back();

    int part1 = modPow(superPow(a, b), 10, MOD);  // a^(remaining digits * 10) % MOD
    int part2 = modPow(a, lastDigit, MOD);        // a^lastDigit % MOD

    return (part1 * part2) % MOD;
}
```


Runtime
0 ms | Beats 100.00%
Analyze Complexity
Memory
15.16 MB | Beats 84.20%

7. An array nums of length n is beautiful if: nums is a permutation of the integers in the range [1, n].

For every 0 <= i < j < n, there is no index k with i < k < j where 2 * nums[k] == nums[i] + nums[j]. Given the integer n, return any beautiful array nums of length n. There will be at least one valid answer for the given n.

```
vector<int> beautifulArray(int n) {

   if (n == 1) return {1};

   vector<int> odd = beautifulArray((n + 1) / 2);  // Construct for odd indices

   vector<int> even = beautifulArray(n / 2);       // Construct for even indices

   vector<int> result;

   for (int x : odd) result.push_back(2 * x - 1);  // Map odd part: 2*x - 1

   for (int x : even) result.push_back(2 * x);    // Map even part: 2*x

      return result;

}
```

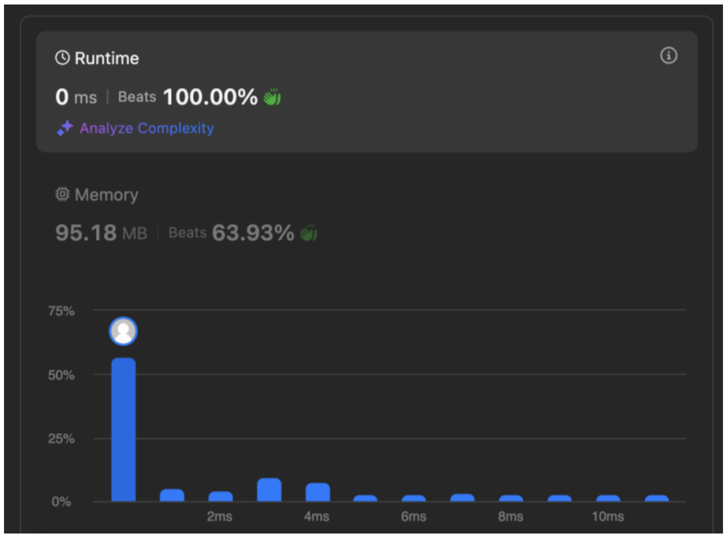

Runtime
12 ms | Beats 11.34%
Analyze Complexity
Memory
16.67 MB | Beats 18.07%

8. Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

```
double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {

   if (nums1.size() > nums2.size()) {

      return findMedianSortedArrays(nums2, nums1);  // Ensure nums1 is the smaller array

   }
```

```
int m = nums1.size();

int n = nums2.size();

int left = 0, right = m;


while (left <= right) {

    int partitionA = (left + right) / 2;

    int partitionB = (m + n + 1) / 2 - partitionA;


    int maxLeftA = (partitionA == 0) ? INT_MIN : nums1[partitionA - 1];

    int minRightA = (partitionA == m) ? INT_MAX : nums1[partitionA];

    int maxLeftB = (partitionB == 0) ? INT_MIN : nums2[partitionB - 1];

    int minRightB = (partitionB == n) ? INT_MAX : nums2[partitionB];


    if (maxLeftA <= minRightB && maxLeftB <= minRightA) {

        if ((m + n) % 2 == 0) {

            return (max(maxLeftA, maxLeftB) + min(minRightA, minRightB)) / 2.0;

        } else {

            return max(maxLeftA, maxLeftB);

        }

    } else if (maxLeftA > minRightB) {

        right = partitionA - 1;

    } else {
```

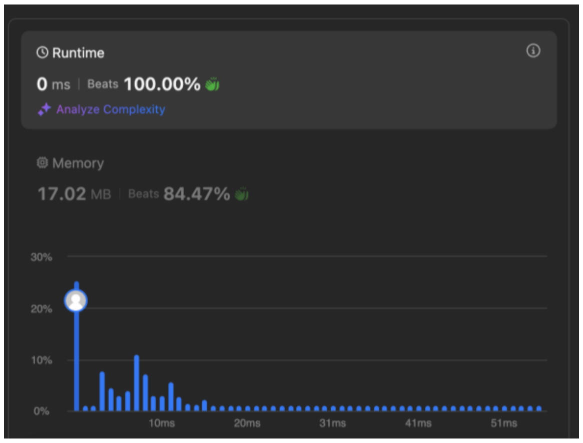
```
                left = partitionA + 1;

            }

        }



        throw invalid_argument("Input arrays are not sorted.");

    }
```


Runtime
0 ms | Beats 100.00%
Analyze Complexity
Memory
95.18 MB | Beats 63.93%

9. Given an n x n matrix where each of the rows and columns is sorted in ascending order, return the kth smallest element in the matrix. Note that it is the kth smallest element in the sorted order, not the kth distinct element.

   You must find a solution with a memory complexity better than O(n2).

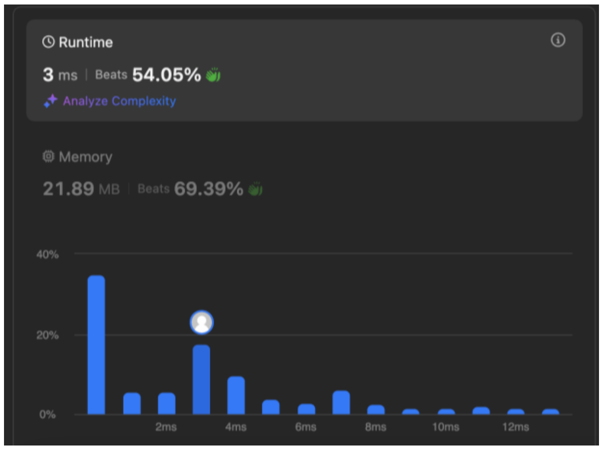
```
int countLessEqual(vector<vector<int>>& matrix, int mid) {

    int n = matrix.size();

    int count = 0, row = n - 1, col = 0;



    while (row >= 0 && col < n) {

        if (matrix[row][col] <= mid) {
                count += row + 1;  // All elements in the current column up to this row are <= mid
                col++;
            } else {
                row--;
            }
        }
        return count;
    }

    int kthSmallest(vector<vector<int>>& matrix, int k) {
        int n = matrix.size();
        int left = matrix[0][0], right = matrix[n - 1][n - 1];

        while (left < right) {
            int mid = left + (right - left) / 2;
            int count = countLessEqual(matrix, mid);

            if (count < k) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        return left;
    }
```


Runtime
0 ms | Beats 100.00%
Memory
17.02 MB | Beats 84.47%

10. Given an integer array nums, reorder it such that nums[0] < nums[1] > nums[2] < nums[3]....

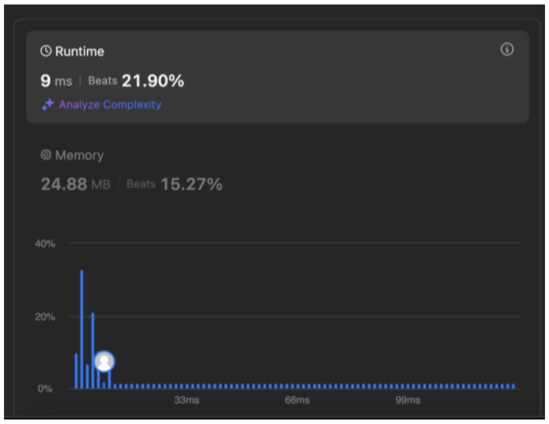
You may assume the input array always has a valid answer.

```
 void wiggleSort(vector<int>& nums) {
    for (int i = 0; i < nums.size() - 1; ++i) {
        if ((i % 2 == 0 && nums[i] > nums[i + 1]) || (i % 2 == 1 && nums[i] < nums[i + 1])) {
            swap(nums[i], nums[i + 1]);
        }
    }
}
```



11. Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

```
vector<vector<int>> mergeIntervals(vector<vector<int>>& intervals) {

    if (intervals.empty()) return {};


    // Step 1: Sort intervals based on the starting time

    sort(intervals.begin(), intervals.end());


    vector<vector<int>> merged;

    merged.push_back(intervals[0]);


    // Step 2: Merge intervals

    for (int i = 1; i < intervals.size(); ++i) {

        if (intervals[i][0] <= merged.back()[1]) {

            // Overlapping intervals, merge them

            merged.back()[1] = max(merged.back()[1], intervals[i][1]);
        } else {
            // Non-overlapping interval, add it to the result
            merged.push_back(intervals[i]);
        }
    }

    return merged;
}
```

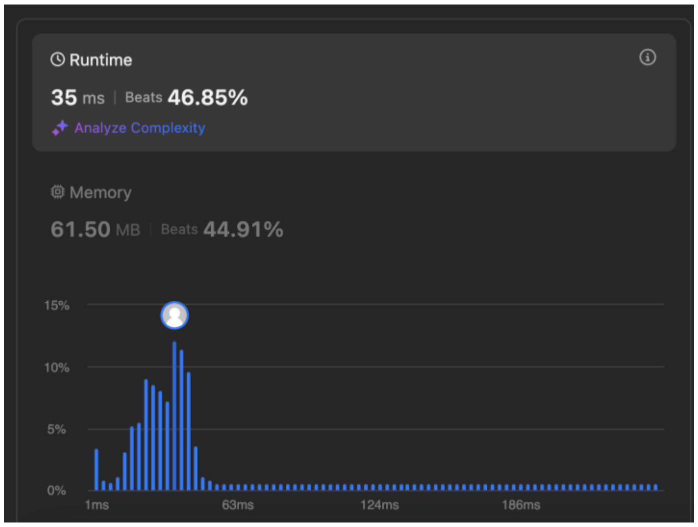

Runtime
9 ms | Beats 21.90%
Analyze Complexity
Memory
24.88 MB | Beats 15.27%

12. A peak element is an element that is strictly greater than its neighbors. Given a 0–indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks. You may imagine that nums[−1] = nums[n] = −∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array. You must write an algorithm that runs in O(log n) time.

```
int findPeakElement(vector<int>& nums) {
    int left = 0, right = nums.size() - 1;

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] > nums[mid + 1]) {
            // Peak is in the left half (including mid)
            right = mid;
        } else {
            // Peak is in the right half (excluding mid)
            left = mid + 1;
        }
    }

    return left;  // left == right, which is a peak index
}
```

13. Given an integer array nums and an integer k, return the kth largest element in the array. Note that it is the kth largest element in the sorted order, not the kth distinct element. Can you solve it without sorting?

```
int findKthLargest(vector<int>& nums, int k) {
    // Min-heap to store the top k elements
    priority_queue<int, vector<int>, greater<int>> minHeap;

    for (int num : nums) {
        minHeap.push(num);
        // If the heap size exceeds k, remove the smallest element
        if (minHeap.size() > k) {
            minHeap.pop();
        }
    }

    // The top element of the heap is the kth largest
    return minHeap.top();
}
int findKthLargest(vector<int>& nums, int k) {
    // Min-heap to store the top k elements
    priority_queue<int, vector<int>, greater<int>> minHeap;

    for (int num : nums) {
        minHeap.push(num);
        // If the heap size exceeds k, remove the smallest element
        if (minHeap.size() > k) {
            minHeap.pop();
        }
    }

    // The top element of the heap is the kth largest
    return minHeap.top();
}
```


Runtime

35 ms | Beats 46.85%

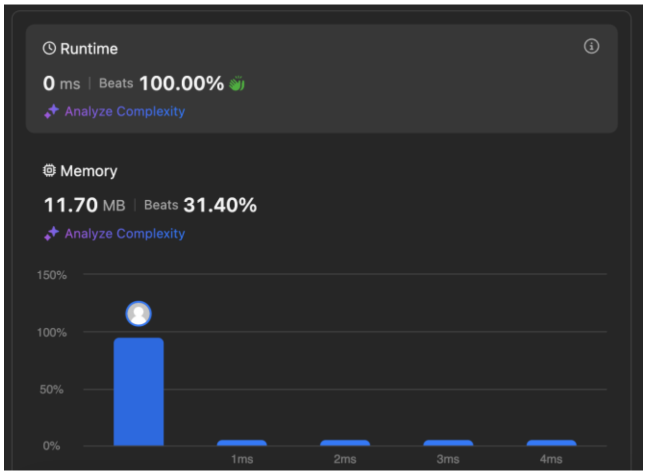
Analyze Complexity

Memory

61.50 MB | Beats 44.91%

14. Given an array nums with n objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue. We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively. You must solve this problem without using the library's sort function.

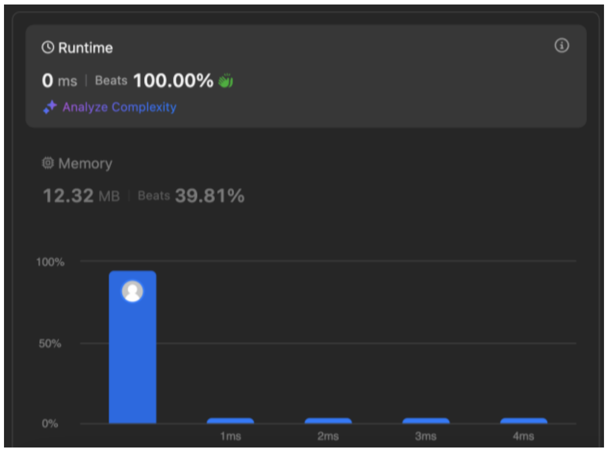
```
void sortColors(vector<int>& nums) {

   int low = 0, mid = 0, high = nums.size() - 1;



   while (mid <= high) {

      if (nums[mid] == 0) {

         // Swap nums[low] and nums[mid], increment both

         swap(nums[low], nums[mid]);

         low++;

         mid++;

      } else if (nums[mid] == 1) {

         // 1 is in the correct place, just move the mid pointer

         mid++;

      } else {
            // Swap nums[mid] and nums[high], decrement high

            swap(nums[mid], nums[high]);

            high--;

        }

    }

}
```


Runtime
0 ms | Beats 100.00%
Analyze Complexity
Memory
11.70 MB | Beats 31.40%
Analyze Complexity

15. You are given two integer arrays nums1 and nums2, sorted in non-decreasing order, and two integers m and n, representing the number of elements in nums1 and nums2 respectively.

    Merge nums1 and nums2 into a single array sorted in non-decreasing order. The final sorted array should not be returned by the function, but instead be stored inside the array nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.

```
void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {

    for(int i=m,j=0;j<n;i++,j++){

    nums1[i]= nums2[j];

    }

    sort(nums1.begin(),nums1.end());
}
```


Runtime
0 ms | Beats 100.00%
Memory
12.32 MB | Beats 39.81%