







Strings:

The .compareTo() method in Java is used to compare two strings lexicographically (alphabetically).

str1.compareTo(str2) returns:

Negative value if str1 comes before str2 alphabetically.

Zero if str1 and str2 are equal.

Positive value if str1 comes after str2 alphabetically.

Certainly! Let's dive into a **detailed explanation** of the **Divide and Conquer** approach used to solve the **Majority Element** problem. We'll go through each line of code, explain the logic behind it, and illustrate it with an example.

**Problem Recap:**

You are given an array of size n, and you need to find the majority element. The majority element is the one that appears **more than n / 2 times** in the array. It's guaranteed that such an element always exists.

**Example:**

* Input: nums = [2, 2, 1, 1, 1, 2, 2]
* Output: 2

**Code Breakdown**

public class MajorityElement {

public static int majorityElement(int[] nums) {

return majorityElementRec(nums, 0, nums.length - 1);

}

**Step 1: majorityElement Function**

* This function is called when the user provides the array nums. It invokes the recursive function majorityElementRec to begin the divide and conquer process.
* **Input:** nums = [2, 2, 1, 1, 1, 2, 2]
* **Action:** We call majorityElementRec(nums, 0, 6) (the entire array from index 0 to 6).

private static int majorityElementRec(int[] nums, int left, int right) {

if (left == right) {

return nums[left];

}

**Step 2: Base Case - When the Subarray Contains One Element**

* This is the **base case** for the recursion. If the subarray contains only one element (i.e., left == right), then that element is by definition the majority element in that subarray.
* **Input:** For example, when left = 2 and right = 2, we only have one element nums[2] which is 1.
* **Action:** Return nums[2] which is 1.

int mid = left + (right - left) / 2;

**Step 3: Divide - Finding the Middle Point**

* This line calculates the middle index mid of the subarray, splitting it into two halves.
* **Input:** For left = 0 and right = 6, we calculate mid = 0 + (6 - 0) / 2 = 3. So, mid = 3.

int leftMajor = majorityElementRec(nums, left, mid);

int rightMajor = majorityElementRec(nums, mid + 1, right);

**Step 4: Recursion - Finding Majority in Left and Right Halves**

* We now **recursively** call majorityElementRec for the left half (left to mid) and the right half (mid + 1 to right).
* The **left half** is [2, 2, 1, 1], and the **right half** is [1, 1, 2, 2].

Let's see how the recursion unfolds:

* + For the left half [2, 2, 1, 1], it keeps splitting until we reach base cases, eventually returning 2 as the majority.
  + For the right half [1, 1, 2, 2], it also keeps splitting until we reach base cases, eventually returning 2 as the majority.
* **Action:** At this point, we have leftMajor = 2 and rightMajor = 2.

if (leftMajor == rightMajor) {

return leftMajor;

}

**Step 5: Combine - If Left and Right Majority Are the Same**

* If both the left and right halves have the same majority element, we simply return that element as the majority for the entire array.
* **Input:** In our case, both leftMajor and rightMajor are 2. Therefore, we return 2 as the majority element for this subarray.
* **Action:** We return 2 because both halves agree.

int leftCount = countOccurrences(nums, left, right, leftMajor);

int rightCount = countOccurrences(nums, left, right, rightMajor);

**Step 6: Count Occurrences of Left and Right Majorities**

* If the majority elements from the left and right halves are different, we need to **count their occurrences** in the full current subarray to determine which one is the actual majority.
* **Action:** We call countOccurrences for both leftMajor and rightMajor.

return leftCount > rightCount ? leftMajor : rightMajor;

}

**Step 7: Return the Element with the Higher Count**

* After counting, we return the element that appears more times in the current subarray.
* **Input:** If leftMajor = 2 and rightMajor = 1, we count 2 appearing 4 times and 1 appearing 3 times. So, 2 is the majority element and we return 2.

private static int countOccurrences(int[] nums, int left, int right, int num) {

int count = 0;

for (int i = left; i <= right; i++) {

if (nums[i] == num) {

count++;

}

}

return count;

}

**Step 8: Counting Occurrences of a Given Number**

* This helper function counts how many times the number num appears in the subarray from left to right.
* **Input:** If nums = [2, 2, 1, 1, 1, 2, 2], left = 0, right = 6, and num = 2, we loop through the array and count 2 appearing 4 times.
* **Action:** Return the count.

**Final Example Walkthrough**

Let's break it down using the array nums = [2, 2, 1, 1, 1, 2, 2].

**Initial Call:**

* majorityElementRec(nums, 0, 6)
  + Mid: 3
  + Left Subarray: [2, 2, 1, 1]
  + Right Subarray: [1, 1, 2, 2]

**Left Subarray ([2, 2, 1, 1]):**

* Recursive calls break this down into smaller subarrays until we reach base cases.
  + Base case returns 2 for the subarray [2, 2] and 1 for the subarray [1, 1].
* Combine step: leftMajor = 2 and rightMajor = 1. Since they're different, we count their occurrences in [2, 2, 1, 1].
  + Count of 2 is 2, count of 1 is 2. Since they are equal, return 2 as the majority.

**Right Subarray ([1, 1, 2, 2]):**

* Similarly, it breaks down into smaller subarrays, with 1 being the majority in the left half and 2 being the majority in the right half.
* Combine step: leftMajor = 1 and rightMajor = 2. We count their occurrences in [1, 1, 2, 2].
  + Count of 1 is 2, count of 2 is 2. Since they are equal, return 2 as the majority.

**Final Combine Step:**

* Now, the entire array has leftMajor = 2 and rightMajor = 2.
* We return 2 because both halves agree.

**Output:**

The majority element is 2.

**Time Complexity:**

* **Recursion depth:** The recursion splits the array in half at each level. This gives a depth of O(log n).
* **Work per level:** At each level, we call countOccurrences, which takes O(n) time.
* **Overall complexity:** The time complexity is O(n log n) due to splitting and counting at each level of recursion.

**Space Complexity:**

* **Recursive stack space:** The recursion depth is O(log n) due to the divide step.
* **Auxiliary space for countOccurrences:** It doesn't use extra space, but the counting at each level requires iterating over the entire array.
* **Overall space complexity:** O(log n) for recursion stack.

**Conclusion:**

This **Divide and Conquer** solution efficiently finds the majority element by breaking down the problem into smaller subproblems and combining the results. The idea is that the majority element in any subarray will either be the same as the majority element in its left or right half, or we need to count occurrences of both candidates and return the one that appears more frequently.

Sure! Let's walk through the **Divide and Conquer** approach for finding the **majority element** in the array [3, 2, 3] step by step.

### ****Input:**** nums = [3, 2, 3]

### ****Initial Call:****

The function majorityElementRec(nums, 0, 2) is called. Here, 0 is the left index and 2 is the right index of the array.

### Step 1: ****First Recursion (Full Array**** [3, 2, 3]****)****

majorityElementRec(nums, 0, 2);

* **Base Case Check:** left != right (0 != 2), so the recursion continues.
* **Mid Calculation:** mid = 0 + (2 - 0) / 2 = 1
* The array is now split into:
  + Left Subarray: [3, 2]
  + Right Subarray: [3]

#### Step 2: ****Recursion on Left Subarray**** [3, 2]

We recursively call majorityElementRec(nums, 0, 1) for the left subarray [3, 2].

majorityElementRec(nums, 0, 1);

* **Base Case Check:** left != right (0 != 1), so the recursion continues.
* **Mid Calculation:** mid = 0 + (1 - 0) / 2 = 0
* The array is now split into:
  + Left Subarray: [3]
  + Right Subarray: [2]

##### Step 3: **Recursion on Left Subarray** [3]

We call majorityElementRec(nums, 0, 0) for the left subarray [3].

majorityElementRec(nums, 0, 0);

* **Base Case Check:** left == right (0 == 0), so we return the element nums[0], which is 3.

##### Step 4: **Recursion on Right Subarray** [2]

We call majorityElementRec(nums, 1, 1) for the right subarray [2].

majorityElementRec(nums, 1, 1);

* **Base Case Check:** left == right (1 == 1), so we return the element nums[1], which is 2.

#### Step 5: ****Combining Results for Left Subarray**** [3, 2]

Now we combine the results for the left subarray [3, 2]. We have:

* leftMajor = 3 (from [3])
* rightMajor = 2 (from [2])

Now we count their occurrences in the subarray [3, 2].

* **Count of 3 in [3, 2]:** 1
* **Count of 2 in [3, 2]:** 1

Since both counts are equal, we return **3** as the majority element for this subarray (because we just need to return one of them, and 3 is returned due to the divide and conquer nature).

#### Step 6: ****Recursion on Right Subarray**** [3]

We now call majorityElementRec(nums, 2, 2) for the right subarray [3].

majorityElementRec(nums, 2, 2);

* **Base Case Check:** left == right (2 == 2), so we return the element nums[2], which is 3.

#### Step 7: ****Combining Results for the Entire Array**** [3, 2, 3]

Now, we combine the results for the entire array [3, 2, 3]. We have:

* leftMajor = 3 (from [3, 2])
* rightMajor = 3 (from [3])

Now we count their occurrences in the full array [3, 2, 3].

* **Count of 3 in [3, 2, 3]:** 2
* **Count of 2 in [3, 2, 3]:** 1

Since 3 appears more times than 2, we return **3** as the majority element for the entire array.

### ****Output:****

The majority element is 3.

### ****Recap of the Process:****

1. **Divide Step:** The array [3, 2, 3] is split recursively into smaller subarrays.
2. **Base Case:** Once the subarrays are of size 1, the element itself is returned.
3. **Merge Step:** The majority elements from the left and right subarrays are compared, and their occurrences are counted.
4. **Result:** The majority element is returned.

### ****Time Complexity:****

* **Divide step:** The array is divided into subarrays, which takes O(log n) time.
* **Counting occurrences:** At each level, we count the occurrences of the majority elements, which takes O(n) time.
* **Overall time complexity:** O(n log n)

### ****Space Complexity:****

* **Recursive stack space:** The recursion depth is O(log n).
* **Overall space complexity:** O(log n) due to the recursion stack.

String : how to chec it contains vowels or not:

if ("aeiouAEIOU".indexOf(s.charAt(i)) != -1) {

**The Condition (!= -1)**

* != -1 checks if s.charAt(i) exists in the "aeiouAEIOU" string.

TO compare int a, int b WHETHER (AB IS GREATER OR BA)

* public static boolean compare(int a, int b) {
* String s1 = Integer.toString(a);
* String s2 = Integer.toString(b);
* // Compare based on concatenation order: a + b vs b + a
* return (s1 + s2).compareTo(s2 + s1) > 0;
* }