<https://leetcode.com/problems/number-of-excellent-pairs/description/>

You are given a **0-indexed** positive integer array nums and a positive integer k.

A pair of numbers (num1, num2) is called **excellent** if the following conditions are satisfied:

Both the numbers num1 and num2 exist in the array nums.

The sum of the number of set bits in num1 OR num2 and num1 AND num2 is greater than or equal to k, where OR is the bitwise **OR** operation and AND is the bitwise **AND** operation.

Return **the number of distinct excellent pairs**.

Two pairs (a, b) and (c, d) are considered distinct if either a != c or b != d. For example, (1, 2) and (2, 1) are distinct.

**Note** that a pair (num1, num2) such that num1 == num2 can also be excellent if you have at least **one** occurrence of num1 in the array.

**Example 1:**

**Input:** nums = [1,2,3,1], k = 3

**Output:** 5

**Explanation:** The excellent pairs are the following:

- (3, 3). (3 AND 3) and (3 OR 3) are both equal to (11) in binary.

The total number of set bits is 2 + 2 = 4, which is greater than or equal to k = 3.

- (2, 3) and (3, 2). (2 AND 3) is equal to (10) in binary, and (2 OR 3) is equal to (11) in binary.

The total number of set bits is 1 + 2 = 3.

- (1, 3) and (3, 1). (1 AND 3) is equal to (01) in binary, and (1 OR 3) is equal to (11) in binary.

The total number of set bits is 1 + 2 = 3.So the number of excellent pairs is 5.

**Example 2:**

**Input:** nums = [5,1,1], k = 10

**Output:** 0

**Explanation:** There are no excellent pairs for this array.

**Constraints:**

1 <= nums.length <= 10^5

1 <= nums[i] <= 10^9

1 <= k <= 60

**Attempt 1: 2024-01-25**

**Solution 1: Math + Bit Manipulation (120 min)**

class Solution {

    public long countExcellentPairs(int[] nums, int k) {

        // Use a set to eliminate duplicate values from 'nums'

        Set<Integer> set = new HashSet<>();

        for(int num : nums) {

            set.add(num);

        }

        // 10^9 < 2^30, no need 32 bits

        // Array to store how many numbers have a certain bit count

        int[] setBitsCounts = new int[30];

        for(int num : set) {

            // Count the 1-bits(set bits) for current number

            int setBits = findNumSetBits(num);

            // Increase the count for this number of 1-bits(set bits)

            setBitsCounts[setBits]++;

        }

        long result = 0;

        for(int i = 1; i < 30; i++) {

            for(int j = Math.max(i, k - i); j < 30; j++) {

                // If i != j, the number of pairs is cnt[i] \* cnt[j] \* 2;

                // numbers for i and j are different, so they form two pairs.

                // If i == j, the number of pairs is cnt[i] \* cnt[i].

                // This is different from above, so that pairs with the same

                // number are counted once.

                result += setBitsCounts[i] \* setBitsCounts[j] \* (i == j ? 1 : 2);

            }

        }

        return result;

    }

    private int findNumSetBits(int num) {

        int count = 0;

        while(num != 0) {

            // Use bitwise AND to check the rightmost bit

            if((num & 1) == 1) {

                count++;

            }

            // Right shift the number to check the next bit

            // Note: The >>> operator is the unsigned right bit-shift operator

            // The difference between >> and >>> would only show up when shifting

            // negative numbers. The >> operator shifts a 1 bit into the most

            // significant bit if it was a 1, and the >>> shifts in a 0 regardless.

            num >>>= 1;

        }

        return count;

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/number-of-excellent-pairs/solutions/2324634/formula/>

The important point to realize is that the sum of OR and AND is just the sum of bits of two numbers.

We dedup the input array, and count numbers containing n bits (where n is [1..29]);

Then, we pick any combination of bits i and j, such that i + j >= k.

**If i != j, the number of pairs is cnt[i] \* cnt[j] \* 2; numbers for i and j are different, so they form two pairs.**

**If i == j, the number of pairs is cnt[i] \* cnt[i]. This is different from above, so that pairs with the same number are counted once.**

If i == j the number of pairs is n \* (n + 1) / 2. Then, we multiply it by 2, and we get n \* (n + 1).Now, we need to remove n elements as they only form one pair. Thus, we get n \* n.

**Java**

public long countExcellentPairs(int[] nums, int k) {

long res = 0, cnt[] = new long[30];

for (int n : Arrays.stream(nums).distinct().toArray()) {

++cnt[Integer.bitCount(n)];

}

for (int i = 1; i < 30; ++i)

for (int j = Math.max(i, k - i); j < 30; ++j)

res += cnt[i] \* cnt[j] \* (i == j ? 1 : 2);

return res;

}

**Q1: Can someone explain why 30 is taken and not 32? Did nums[i] < 10^9 help you come up with that?**

**Refer to**

<https://leetcode.com/problems/number-of-excellent-pairs/solutions/2324984/java-c-python-inclusion-exclusion-principle/comments/1505171>

**A1:** Because 10^9 approximates to 2^30

**Q2: What is Integer.bitCount() method in java? And how to customize it without using library version ?**

**Refer to**

<https://www.studytonight.com/java-wrapper-class/java-integer-bitcount-method>

**A2:** Java bitCount() method belongs to the Integer class. This method is used to return a one-bit number into two's complement binary form of the specified integer value and counts a number of set bits in a binary sequence.

This method takes an integer as a parameter and returns an integer too. For example, if the given input is **1000111110** than this method should return **6**, as there are **6** set **bits(1)** in this input.

If you want to create a customized version of **Integer.bitCount** in Java without using the library method, you can implement your own logic to count the number of set bits (1s) in the binary representation of an integer. Here's an example:

public class CustomBitCount {

public static void main(String[] args) {

int num = 42; // Replace this with your integer value

int count = customBitCount(num);

System.out.println("Number: " + num);

System.out.println("Custom Bit Count: " + count);

}

private static int customBitCount(int num) {

int count = 0;

while (num != 0) {

// Use bitwise AND to check the rightmost bit

if ((num & 1) == 1) {

count++;

}

// Right shift the number to check the next bit

num >>>= 1;

}

return count;

}

}

**Q3: Do we need to handle singles specially?**

**A3:** No need

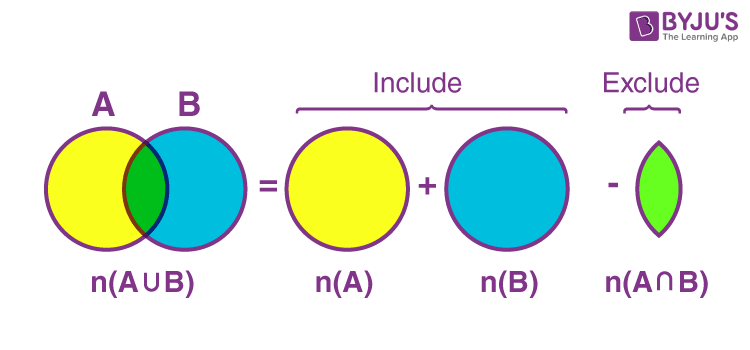
**Refer to**

<https://leetcode.com/problems/number-of-excellent-pairs/solutions/2324984/java-c-python-inclusion-exclusion-principle/>

**Intuition**

**The Inclusion-Exclusion Principle**

**bits(num1 OR num2) + bits(num1 AND num2) = bits(num1) + bits(num2)**



**Explanation**

For all different a in nums,

counts its number of bits.

Enumearte the number of bits k1 and k2,

if k1 + k2 >= k,

we accumulate count[k1] \* count[k2].

**Complexity**

Time O(nlogn)

Space O(n)

**Java**

public long countExcellentPairs(int[] A, int k) {

long cnt[] = new long[30], res = 0;

Set<Integer> set = new HashSet<>();

for (int a : A)

set.add(a);

for (int a : set)

cnt[Integer.bitCount(a)]++;

for (int i = 1; i < 30; ++i)

for (int j = 1; j < 30; ++j)

if (i + j >= k)

res += cnt[i] \* cnt[j];

return res;

}

**Refer to**

<https://algo.monster/liteproblems/2354>

**Problem Description**

In this problem, we're presented with an array nums consisting of 0-indexed positive integers and a positive integer k. Our goal is to find the number of distinct excellent pairs in the array, where a pair (num1, num2) is considered excellent if it satisfies two conditions:

Both num1 and num2 exist in nums.

The sum of the number of set bits (bits with value 1) in num1 OR num2 and num1 AND num2 must be greater than or equal to k.

We count the number of set bits using bitwise OR and AND operations, and we are looking for pairs that collectively have a large enough number of set bits to meet or exceed the threshold k.

Also, it's important to note that a pair where num1 is equal to num2 can also be considered excellent if the number of its set bits is sufficient and at least one occurrence of the number exists in the array. Moreover, we want the count of distinct excellent pairs, so the order matters—(a, b) is considered different from (b, a).

**Intuition**

To find the number of excellent pairs efficiently, we need to think about the problem in terms of set bits because the conditions for an excellent pair depend on the sum of set bits in num1 OR num2 and num1 AND num2.

One intuitive approach is to use the properties of bitwise operations. For any two numbers, num1 OR num2 will have the highest possible set bit count of the two numbers because OR operation results in a 1 for each bit that is 1 in either num1 or num2. On the other hand, num1 AND num2 will have set bits only in positions where both num1 and num2 have set bits.

**Since only the set bits matter, and we're looking for pairs (num1, num2) that meet a certain combined set bit count, we can reduce the complexity by avoiding the direct computation of num1 OR num2 and num1 AND num2 for all pairs. Instead, we preprocess by counting the set bits for each unique number in nums.**

**To ensure that we count each distinct pair only once, we eliminate duplicates in the array by converting it into a set. We then create a counter to keep track of how many numbers have a specific set bit count. This preprocessing step simplifies our task to just combining counts from the counter, avoiding the need to directly compute the set bit sum for every possible pair.**

When we iterate over the unique numbers in nums set, we determine the number of set bits for each unique value using bit\_count(). We then iterate through our set bit count counter and add the count of numbers that have enough set bits to complement the current number's set bit count to reach at least k. This way, we find all the pairs that, when combined, meet or exceed the threshold k.

By adding the counts for all such complementing set bit count pairs, we calculate the total number of excellent pairs, bypassing the problem of having to generate and check every possible pair, which would be computationally inefficient.

**Solution Approach**

The given solution employs a combination of bit manipulation and hash mapping to efficiently compute the number of excellent pairs. Let's break down the implementation step-by-step:

**Eliminating Duplicates:** The solution begins by converting the original list of numbers into a set. This serves two purposes:

Ensures that each number is considered only once, thereby eliminating redundant pairs like (num1, num1) when num1 appears multiple times in the input list.

Helps later in avoiding double counting of excellent pairs with the same numbers but in different positions (since (a, b) and (b, a) are distinct).

s = set(nums)

**Counting Set Bits:** Then, the algorithm uses a Counter from the collections module to keep track of how many numbers share the same set bit count.

cnt = Counter()

for v in s:

cnt[v.bit\_count()] += 1

**The method bit\_count() is used to determine the number of set bits in each number. These counts are stored such that cnt[i] reflects the number of unique numbers with exactly i set bits.**

**Finding Excellent Pairs:** The core of the solution is to find all the pairs that satisfy the condition of having a combined set bit sum (via bitwise OR and AND) greater than or equal to k. Since an OR operation can never reduce the number of set bits, and an AND operation can only produce set bits that are already set in both numbers, **the combined set bit count for a pair (num1, num2) will be equivalent to bit\_count(num1) + bit\_count(num2).**

The solution iterates over each unique value v from the set of numbers and then checks for every bit count i stored in cnt if the sum of bit\_count(v) and i is greater than or equal to k.

ans = 0

for v in s:

t = v.bit\_count()

for i, x in cnt.items():

if t + i >= k:

ans += x

t holds the number of set bits for the current unique number v.

If t + i is greater than or equal to k, all numbers with a set bit count i can form an excellent pair with v. The count of such numbers is x, and we add this to our total count of excellent pairs (ans). This step leverages the precomputed counts of unique set bit numbers to quickly find the number of complements needed to meet the set bit threshold k.

**Returning the result:** Finally, after iterating through all unique numbers and their possible pairings, the solution returns ans, which holds the total number of distinct excellent pairs.

return ans

In summary, the solution follows a smart preprocessing step to calculate and use set bit counts for optimization. This eliminates redundant operations and directly navigates to the crux of the problem, which significantly improves the computational efficiency.

By employing a hash map to store unique set bit counts and identify complementing pairs, the solution reduces what would be a quadratic-time complexity task to a complexity proportional to the product of unique numbers and unique set bit counts present in the input.

**Example Walkthrough**

Let's say we have an array nums = [3, 1, 2, 2] and k = 3.

First, we'll apply step 1 and eliminate duplicates by converting nums into a set, which will give us s = {1, 2, 3}.

Next, we count set bits in step 2. Every number will be processed as follows:

1 has 1 set bit.

2 has 1 set bit.

3 has 2 set bits.

**The Counter object cnt turns out to be {1: 2, 2: 1}, indicating that there are 2 numbers with 1 set bit and 1 number with 2 set bits.**

Now for step 3, we find excellent pairs. We go through each number in s and compare the set bit count with our k value:

For v = 1 with 1 set bit, cnt[1] = 2. Since 1 (set bits of v) + 1 (set bits of another number) is not >= k, no excellent pairs are formed with v = 1.

For v = 2 also with 1 set bit, the situation is the same as for v = 1.

For v = 3 with 2 set bits, we compare it against cnt entries.

2 (set bits of v) + 1 (set bits of another number) is >= k, thus excellent pairs are (3, 1) and (3, 2). Since there are 2 numbers with 1 set bit, we add 2 to ans, resulting in ans = 2 so far.

Finally, since the pairs (1, 3) and (2, 3) are also excellent pairs (order matters), we count them again. This adds another 2 to ans, making ans = 4.

After processing all the unique numbers, we follow step 4 and conclude that there are 4 distinct excellent pairs. The pairs that satisfy the conditions are (3, 1), (3, 2), (1, 3), (2, 3).

This example demonstrates the solution's efficiency, as it avoids checking all possible combinations of nums and directly focuses on the set bit counts to find excellent pairs, which is computationally faster.

**Java Solution**

class Solution {

    // Method to count the number of excellent pairs

    public long countExcellentPairs(int[] nums, int k) {

        // Use a set to eliminate duplicate values from 'nums'

        Set<Integer> uniqueNumbers = new HashSet<>();

        for (int num : nums) {

            uniqueNumbers.add(num);

        }

        long totalPairs = 0;             // To store the total number of excellent pairs

        int[] bitCounts = new int[32];   // Array to store how many numbers have a certain bit count

        // Count the occurrence of each bit count for the unique elements

        for (int num : uniqueNumbers) {

            int bits = Integer.bitCount(num);  // Count the 1-bits in the binary representation of 'num'

            ++bitCounts[bits];                 // Increase the count for this number of 1-bits

        }

        // Iterate over the unique numbers to find pairs

        for (int num : uniqueNumbers) {

            int bits = Integer.bitCount(num); // Count the 1-bits for this number

            // Check for each possible bit count that could form an excellent pair with 'num'

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

                // Check if the sum of 1-bits is at least 'k'

                if (bits + i >= k) {

                    totalPairs += bitCounts[i]; // If it is, add the count of numbers with 'i' bits to the total

                }

            }

        }

        return totalPairs; // Return the total count of excellent pairs

    }

}

**Time and Space Complexity**

The given code counts the number of "excellent" pairs in an array, with a pair (a, b) being excellent if the number of bits set to 1 in their binary representation (a OR b) is at least k.

**Time Complexity**

The time complexity of the function involves several steps:

Creating a set s from nums: This operation has a time complexity of O(N) where N is the number of elements in nums, as each element is added to the set once.

Counting the bit\_count t of each distinct number in s: Each call to v.bit\_count() is O(1). Since we do this for each number in the set s, this step has a complexity of O(U), where U is the number of unique numbers in nums.

Populating the Counter: This again is O(U) since we're iterating over the set s once.

Running the nested loops, where the outer loop is over each unique value v in s and the inner loop is over the counts in Counter: Since the outer loop runs O(U) times and the inner loop runs a maximum of O(U) times, the nested loop part has a worse-case complexity of O(U^2).

The total time complexity is, therefore, O(N + U^2).

**Space Complexity**

The space complexity consists of:

The set s, which takes O(U) space.

The Counter object cnt, which takes another O(U) space.

The additional space for the variable ans and loop counters is O(1).

Hence, the total space complexity of the function is O(U), where U is the count of unique numbers in nums.