**Array**

**Element Appearing More than 25% in Sorted Array**

Given an integer array sorted in non-decreasing order, there is exactly one integer in the array that occurs more than 25% of the time, return that integer.

1. Use a start pointer to record the first occurrence of an element.

2. Use an i pointer to iterate through the array until encountering an element different from nums[start].

3. Calculate the number of occurrences of the current element using (i - start).

4. If (i - start) / nums.size() is greater than 0.25, the current element is the target element.

5. Update start to the current element's position and continue traversing.

Time Complexity: O(N)

Space Complexity: O(1)

int findSpecialInteger(vector<int>& nums) {

int start = 0;

int threshold = nums.size() / 4;

for (int i = 0; i < nums.size(); i++)

{

if (nums[i] != nums[start]) {

if (i - start > threshold) {

return nums[start];

}

start = i;

}

}

//If the element at the end of the array exceeds the threshold

if (nums.size() - start > threshold) {

return nums[start];

}

return -1;

}

**Find All Numbers Disappeared in an Array**

Given an array nums of n integers where nums[i] is in the range [1, n], return an array of all the integers in the range [1, n] that do not appear in nums.

Solution 1 (Hash Table):

1. Initialize a hash table where each key represents a number from 1 to n, and the value represents the count of occurrences of that key. Initially, set all values to 0.

2. For each element in the given array, use it as a key in the hash table and increment its corresponding value by 1.

3. After traversing the array, go through the hash table and identify keys that have a value of 0. These keys represent the numbers that are missing from the original array.

Time Complexity: O(N)

Space Complexity: O(N)

Solution 2 (In-place Modification Method):

1. For each element 'nums[i]', use 'abs[nums[i]]-1' as the index 'index'. Set 'nums[index]' to a negative value to mark that the number 'index+1' has appeared in the array.

2. Identify all the positions 'i' that hold positive values. These positions correspond to the number 'i+1' that are missing from the original array.

Time Complexity: O(N)

Space Complexity: O(1)

vector<int> findDisappearedNumbers(vector<int>& nums) {

for (int i = 0; i < nums.size(); i++)

{

int index = abs(nums[i]) - 1;

if (nums[index] > 0) {

nums[index] = -nums[index];

}

}

vector<int> result;

for (int i = 0; i < nums.size(); i++)

{

if (nums[i] > 0) {

result.push\_back(i + 1);

}

}

return result;

}

**Maximum Average Subarray Ⅰ**

You are given an integer array nums consisting of n elements, and an integer k.

Find a contiguous subarray whose length is equal to k that has the maximum average value and return this value. Any answer with a calculation error less than 10-5 will be accepted.

Sliding Window Approach

1. First, calculate the sum of the first k elements. This sum represents the sum of the initial window.

2. Starting from the k-th element, slide the window one element at a time to the right. For each new element added to the window, update the window sum by subtracting the element that is left out from the window and adding the new element. This method efficiently updates the current window sum without recalculating the entire sum from scratch.

3. After each shift of the window, compare the current window sum to the maximum sum recorded so far. Update the maximum sum if the current window sum is greater.

4. After processing all possible windows, return the maximum sum divided by k to get the maximum average.

Time Complexity: O(N)

Space Complexity: O(1)

double findMaxAverage(vector<int>& nums, int k) {

double maxSum = 0;

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

maxSum += nums[i];

}

double currentSum = maxSum;

for (int i = k; i < nums.size(); i++)

{

currentSum += nums[i] - nums[i - k];

maxSum = max(maxSum, currentSum);

}

return maxSum / k;

}

**Minimum Size Subarray Sum**

Given an array of positive integers nums and a positive integer target, return the minimal length of a subarray whose sum is greater than or equal to target. If there is no such subarray, return 0 instead.

1. Start with two pointers left and right both pointing to the beginning of the array.

2. Move the right pointer to expand the window, adding the current element to the cumulative sum.

3. Whenever the sum is greater than or equal to the target, move the left pointer to shrink the window while updating the minimal length of the subarray.

4. Repeat the above steps until the right pointer reaches the end of the array.

Time Complexity: O(N)

Space Complexity: O(1)

int minSubArrayLen(int target, vector<int>& nums) {

int left = 0, sum = 0, minLength = INT\_MAX;

for (int right = 0; right < nums.size(); right++)

{

sum += nums[right];

while (sum >= target)

{

minLength = min(minLength, right - left + 1);

sum -= nums[left];

left++;

}

}

return minLength == INT\_MAX ? 0 : minLength;

}

**Subarray Product Less Than K**

Given an array of integers nums and an integer k, return the number of contiguous subarrays where the product of all the elements in the subarray is strictly less than k.

1. Use two pointer left and right to define a sliding window. Track the product of elements within thw window using product, and count valid subarrays with count.

2. Move right pointer across the array, multiplying product by nums[right]. If product>=k, shrink the window by moving left until product<k.

3. Each valid window contributes right-left+1 subarray. And the total count is returned.

Time Complexity: O(n)

Space Complexity: O(1)

int numSubarrayProductLessThanK(vector<int>& nums, int k) {

if (k <= 1) return 0;

int left = 0, count = 0, product = 1;

for (int right = 0; right < nums.size(); right++)

{

product \*= nums[right];

while (product >= k)

{

product /= nums[left];

left++;

}

count += right - left + 1;

}

return count;

}

**Max Consecutive Ones**

Given a binary array nums, return the maximum number of consecutive 1's in the array.

1. Set 'int max = 0' to store the maximum number of consecutive 1's.Initialize a pointer start to 0.

2. Iterate through the array. When a 1 is encountered, if it's the first 1, record its index with start.

3. Continue until a 0 is encountered. When a 0 is found, calculate the length of consecutive 1s as 'i - start' and update max if this length is greater than the current max.

4. After encountering a 0, continue searching for the next 1. Record its index with start and repeat the process.

5. If the end of the array is reached while still in a sequence of 1s, calculate the length of this sequence as 'nums.size() - start' and update max if necessary.

Time Complexity: O(N)

Space Complexity: O(1)

int findMaxConsecutiveOnes(vector<int>& nums) {

int max = 0;

int start = -1;

for (int i = 0; i < nums.size(); i++) {

if (nums[i] == 1) {

if (start == -1)

{

start = i;

}

}

else

{

if (start != -1)

{

max = std::max(max, i - start);

start = -1;

}

}

}

if (start != -1) {

max = std::max(max, (int)nums.size() - start);

}

return max;

}

**Merge Sorted Array**

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.

Use a two-pointer approach, iterating from the end of both arrays and placing the larger element at the end of the merged array in 'nums1'.

1. Create three pointers:

p1 for traversing nums1 from the end of its valid elements, initially pointing to the (m-1)th element. p2 for traversing nums2 from the end, initially pointing to the (n-1)th element. tail for indicating the current position in nums1 where the next largest element should be placed, initially pointing to the (m+n-1)th element.

2. Start a loop with the condition that both arrays have not been fully traversed. Compare the elements pointed to by p1 and p2. Assign the larger element to the position indicated by tail in nums1, and move the corresponding pointer (p1 or p2) backward. Move tail backward as well.

3. If p2 still has remaining elements after the loop ends, copy all remaining elements from nums2 to nums1.

Time Complexity: O(m+n)

Space Complexity: O(1)

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

int p1 = m - 1;

int p2 = n - 1;

int tail = m + n - 1;

while (p1 >= 0 || p2 >= 0) {

if (p1 == -1) {

nums1[tail--] = nums2[p2--];

} else if (p2 == -1) {

nums1[tail--] = nums1[p1--];

} else if (nums1[p1] > nums2[p2]) {

nums1[tail--] = nums1[p1--];

} else {

nums1[tail--] = nums2[p2--];

}

}

}

**Missing Ranges**

You are given an inclusive range [lower, upper] and a sorted unique integer array nums, where all elements are within the inclusive range.

A number x is considered missing if x is in the range [lower, upper] and x is not in nums.

Return the shortest sorted list of ranges that exactly covers all the missing numbers. That is, no element of nums is included in any of the ranges, and each missing number is covered by one of the ranges.

1. Use two variables, prev and cur, to record the left and right values of the range to be output. Since the result involves +1 or -1 of array elements, edge values need to be considered.

2. Define prev as lower - 1, and set the loop condition to i <= nums.size().

3. Start the loop, and when prev + 1 <= cur - 1, it indicates a missing range. Output this range, assign cur to prev, and continue the loop until the algorithm ends.

Time Complexity: O(N)

Space Complexity: O(N)

vector<string> findMissingRanges(vector<int>& nums, int lower, int upper) {

vector<string> result;

int prev = lower - 1;

for (int i = 0; i <= nums.size(); i++)

{

int cur = i < nums.size() ? nums[i] : upper + 1;

if (prev + 1 <= cur - 1) {

result.push\_back(to\_string(prev + 1) + "->" + to\_string(cur - 1));

}

prev = cur;

}

return result;

}

**Plus One**

You are given a large integer represented as an integer array digits, where each digits[i] is the ith digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return the resulting array of digits.

1. Start adding one from the last element of the array, so a pointer is needed to traverse the array from the end.

2. If the current element pointed to is not 9, increment this element by one and stop the traversal.

3. If the current element pointed to is 9, change it to 0 and move the pointer one position to the left. If the first element of the array is still 9 after the traversal, change it to 0 and add a 1 at the beginning of the array.

Time Complexity: O(N)

Space Complexity: O(1) or O(N) (if a new array is created)

vector<int> plusOne(vector<int>& digits) {

int n = digits.size();

for (int i = n - 1; i >= 0; --i)

{

if (digits[i] != 9)

{

digits[i]++;

return digits;

}

digits[i] = 0;

}

digits.insert(digits.begin(), 1);

return digits;

}

**Smallest Equal Index**

Given a 0-indexed integer array nums, return the smallest index i such that i mod 10 == nums[i]. If no such index exists, return -1.

1. Start from the first element and check each element to see if i % 10 == nums[i].

2. If a matching index is found, return it immediately.

3. If no index satisfies the condition, return -1 after the loop completes.

int smallestEqual(vector<int>& nums) {

for (int i = 0; i < nums.size(); ++i) {

if (i % 10 == nums[i]) {

return i;

}

}

return -1;

}

**Summary Ranges**

You are given a sorted unique integer array nums.

A range [a,b] is the set of all integers from a to b (inclusive).

Return the smallest sorted list of ranges that cover all the numbers in the array exactly. That is, each element of nums is covered by exactly one of the ranges, and there is no integer x such that x is in one of the ranges but not in nums.

Each range [a,b] in the list should be output as:

"a->b" if a != b

"a" if a == b

1. Set up two pointers, start and end. start points to the beginning of each interval to be output, initially set to 0. end is the index of the current element being traversed.

2. Iterate through the array and determine if the current element plus 1 is equal to the next element. This checks if two elements are consecutive. When the condition is triggered, either the current element is the last element in the array or the current element plus 1 is not equal to the next element, a discontinuity is found. At this point, output the interval defined by start and end, and then update start to (end + 1).

Time Complexity: O(N)

Space Complexity: O(N)

vector<string> findSummaryRanges(vector<int>& nums) {

vector<string> result;

if (nums.empty()) return result;

int start = 0;

for (int end = 0; end < nums.size(); end++) {

if (end + 1 == nums.size() || nums[end] + 1 != nums[end + 1]) {

if (start == end) {

result.push\_back(to\_string(nums[start]));

}

else

{

result.push\_back(to\_string(nums[start]) + "->" + to\_string(nums[end]));

}

start = end + 1;

}

}

return result;

}

**Two Sum**

Solution 1: Hash Table

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

1. Create a hash table to store the values and their corresponding indices from the array.

2. Iterate through the array. For each element, compute the difference between the target and the current element.

If the difference is not in the hash table, store the current element as the key and its index as the value in the hash table.

If the difference is found in the hash table, it means that the two numbers that add up to the target have been found. Retrieve the index of the difference from the hash table and the current index, and return both indices.

3. The algorithm terminates once the pair is found.

Time Complexity: O(N)

Space Complexity: O(N)

vector<int> twosum(vector<int>& nums, int target) {

unordered\_map<int, int> hashtable;

for (int i = 0; i < nums.size(); i++)

{

auto it = hashtable.find(target - nums[i]);

if (it != hashtable.end()) {

return { it->second,i };

}

hashtable[nums[i]] = i;

}

return {};

}

Solution 2: Two Pointer

1. Sorting the array allows the use of the two-pointer technique effectively.

2. Utilize two pointers, one starting from the beginning (left) and one from the end (right) of the array.

3. Calculate the sum of the elements pointed to by the two pointers.

If the sum equals the target, return the indices of these two elements.

If the sum is less than the target, move the left pointer to the right to increase the sum.

f the sum is greater than the target, move the right pointer to the left to decrease the sum.

4. Create a copy of the array that stores both the values and their original indices before sorting.

5. If a pair is found, return their original indices; if no such pair exists, return an empty array.

Time Complexity: O(nlogn)

Space Complexity: O(N)

vector<vector<int>> twoSumTwoPointers(vector<int>& nums, int targe) {

vector<vector<int>> results;

vector<pair<int, int>> numsWithIndex;

for (int i = 0; i < nums.size(); i++)

{

numsWithIndex.push\_back({ nums[i],i });

}

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

int left = 0;

int right = numsWithIndex.size() - 1;

while (left < right)

{

int sum = numsWithIndex[left].first + numsWithIndex[right].first;

if (sum == targe)

{

results.push\_back({ numsWithIndex[left].second,

numsWithIndex[right].second });

++left;

--right;

}

else if (sum < targe) {

++left;

}

else

{

--right;

}

}

return results;

}

**New Permutation**

A permutation of an array of integers is an arrangement of its members into a sequence or linear order.

The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

1. Traverse the array from right to left to find the first element 'pivot' that is smaller than the element next to it.

2. Traverse the array from right to left to find the first element 'successor' that is larger than 'pivot'.

3. Swap the value of the pivot and the successor.

4. Reverse the sub-array that comes after the original index of the 'pivot' to get the next lexicographical permutation.

Time Complexity: O(N)

Space Complexity: O(N)

void nextPermutation(vector<int>& nums) {

int n = nums.size();

int pivot = -1;

for (int i = n - 2; i >= 0; --i) {

if (nums[i] < nums[i + 1]) {

pivot = i;

break;

}

}

if (pivot == -1) {

reverse(nums.begin(), nums.end());

return;

}

for (int successor = n - 1; successor > pivot; --successor) {

if (nums[successor] > nums[pivot]) {

swap(nums[pivot], nums[successor]);

break;

}

}

reverse(nums.begin() + pivot + 1, nums.end());

}

**Subarray Sum Equals K**

Given an array of integers nums and an integer k, return the total number of subarrays whose sum equals to k.

A subarray is a contiguous non-empty sequence of elements within an array.

Cumulative Sum Approach

1. Initialize a hash map to store cumulative sums and their frequencies.

2. Initialize a variable to keep the cumulative sum and another to count the number of subarrays.

3. Iterate through the array, updating the cumulative sum and using the hash map to check and update the count of subarrays.

Time Complexity: O(N)

Space Complexity: O(N)

int subarraySum(vector<int>& nums, int k) {

int count = 0;

int cumulativeSum = 0;

unordered\_map<int, int> cumulativeSumFreq;

cumulativeSumFreq[0] = 1;

for (int num : nums) {

cumulativeSum += num;

if (cumulativeSumFreq.find(cumulativeSum - k) != cumulativeSumFreq.end()) {

count += cumulativeSumFreq[cumulativeSum - k];

}

cumulativeSumFreq[cumulativeSum]++;

}

return count;

}

**3 Sum Smaller**

Given an array of n integers nums and an integer target, find the number of index triplets i, j, k with 0 <= i < j < k < n that satisfy the condition nums[i] + nums[j] + nums[k] < target.

Approach 1: Binary Search

Approach 2: Two-Pointer Method

1. After sorting the array, iterate through the array.

2. For each element in the array, select two other elements by initializing two pointers—one starting at the element immediately after the current element and the other at the end of the array.

3. If the sum of the three elements is less than the target, due to the sorted nature of the array, all elements between the two pointers satisfy the condition. Thus, the number of valid triplets can be quickly calculated by subtracting the indices of the two pointers. Then, move the left pointer (the one initialized at the next element) to the right.

4. If the sum is greater than or equal to the target, it means the condition is not satisfied, so the right pointer (the one initialized at the end of the array) is moved to the left.

Time Complexity: O(N^2)

Space Complexity: O(1)

int threeSumSmaller(vector<int>& nums, int target) {

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

int count = 0;

for (int i = 0; i < nums.size()-2; i++)

{

int j = i + 1;

int k = nums.size() - 1;

while (j < k) {

if (nums[i] + nums[j] + nums[k] < target) {

count += k - j;

j++;

}

else

{

k--;

}

}

}

return count;

}

**Sum 3**

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

1. Sort the input array nums.

2. Loop through the array with index i from 0 to (len(nums) - 3).

3. For each i, set two pointers: left at (i+1) and right at (len(nums) - 1).

4. Calculate the sum:

If sum is equal to the target, add the triplet to the result.

If sum is less than the target, move the left pointer to the right.

If sum is more than the target, move the right pointer to the left.

Time Complexity: O(N^2)

Space Complexity: O(N^2)

vector<vector<int>> threeSumTarget(vector<int>& nums, int target) {

vector<vector<int>> result;

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

int n = nums.size();

for (int i = 0; i < n - 2; i++) {

int left = i + 1, right = n - 1;

while (left < right) {

int sum = nums[i] + nums[left] + nums[right];

if (target == sum)

{

result.push\_back({ nums[i] ,nums[left] ,nums[right] });

// Avoid duplicates for the second element

while (left < right && nums[left] == nums[left + 1]) {

++left;

}

// Avoid duplicates for the third element

while (left < right && nums[right] == nums[right - 1]) {

--right;

}

++left;

--right;

}

else if (target < sum) {

++left;

}

else

{

--right;

}

}

}

return result;

}

**Sum 4**

Given an array nums of n integers, return an array of all the unique quadruplets [nums[a], nums[b], nums[c], nums[d]]

Sort the input array. Loop through the array from 0 to 'len(nums) - 4'. For each i, use another loop with index j from i+1 to 'len(nums - 3)'. For each pari of i and j, set two pointers: left at j+1 and right at len(nums)-1. Calculate the sum of 4 elements: If the sum is equal to the target, add the quadruplet to the result. If the sum is less than the target, move the left pointer to the right. If the sum is more than the target, move the right pointer to the left.

Time Complexity: O(N^3)

Space Complexity: O(N^2)

vector<vector<int>> fourSum(vector<int>& nums, int target) {

vector<vector<int>> result;

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

int n = nums.size();

for (int i = 0; i < n - 3; i++) {

if (i > 0 && nums[i] == nums[i - 1]) continue;

for (int j = i + 1; j < n - 2; j++) {

if (j > i + 1 && nums[j] == nums[j - 1]) continue;

int left = j + 1, right = n - 1;

while (left < right) {

int sum = nums[i] + nums[j] + nums[left] + nums[right];

if (sum == target) {

result.push\_back({ nums[i], nums[j], nums[left], nums[right] });

while (left < right && nums[left] == nums[left + 1]) {

++left;

}

while (left < right && nums[right]==nums[right - 1]) {

--right;

}

++left; --right;

}

else if (sum < target) { ++left; }

else { --right; }

}

}

}

return result;

}

**Search in Rotated Sorted Array**

There is an integer array nums sorted in ascending order (with distinct values).

Prior to being passed to your function, nums is possibly rotated at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (0-indexed). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums after the possible rotation and an integer target, return the index of target if it is in nums, or -1 if it is not in nums.

You must write an algorithm with O(log n) runtime complexity.

Binary Search Approach

1. Set two pointers, left at the beginning and right at the end of the array.

2. Binary Search Loop:

Calculate the middle index mid and check if the middle element is target. If yes return mid.

Determine which half of the array is sorted:

If the left half is sorted (nums[left]<=nums[mid]), check if the target lies within the sorted left half (nums[left]<=target<=nums[mid]). If yes, move the right pointer to 'mid-1'; Otherwise, move the left pointer to 'mid+1';

If the right half is sorted (nums[mid]<=nums[right]), check if the target lies within the sorted right half (nums[mid]<target<=nums[right]). If yes, move the left pointer to 'mid+1'; Otherwise, move the right pointer to 'mid-1';

If the target is found, return its indesx. If the loop ends without finding the target, return -1;

Time Complexity: O(logn)

Space Complexity: O(1)

int searchInRotatedArray(vector<int>& nums, int target) {

int left = 0, right = nums.size() - 1;

while (left <= right) {

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

if (nums[mid] == target)

{

return mid;

}

if (nums[left] <= nums[mid]) {

if (nums[left] <= target && target < nums[mid])

{

right = mid - 1;

}

else

{

left = mid + 1;

}

}

else {

if (nums[mid] < target && target <= nums[right])

{

left = mid + 1;

}

else

{

right = mid - 1;

}

}

}

return -1;

}

**Find fisrt and last position of element is sorted array**

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

You must write an algorithm with O(log n) runtime complexity.

1. First Binary Search: The objective is to find the starting position of the target element.The comparison uses 'nums[mid] >= target'.

2. Second Binary Search: The objective is to find the ending position of the target element. The comparison uses 'nums[mid] <= target'.

Time Complexity: O(logn)

Space Complexity: O(1)

vector<int> searchRange(vector<int>& nums, int target) {

int left = 0, right = nums.size() - 1;

int first = -1, last = -1;

while (left <= right)

{

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

if (nums[mid] >= target) {

right = mid - 1;

}

else

{

left = mid + 1;

}

if (nums[mid] == target)

first = mid;

}

left = 0;

right = nums.size() - 1;

while (left <= right)

{

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

if (nums[mid] <= target) {

left = mid + 1;

}

else

{

right = mid - 1;

}

if (nums[mid] == target)

last = mid;

}

return { first,last };

}

**Find Peak Element**

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 must write an algorithm that runs in O(log n) time.

Find the middle element. Decide which half contains the peak and adjust left and right to narrow the search. Finally, left will point to a peak element.

Time Complexity: O(logn)

Space Complexity: O(1)

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]) {

right = mid; // Peak is on the left side

}

else {

left = mid + 1; // Peak is on the right side

}

}

return left; // left points to a peak element

}

**Find Minimum in Rotated Sorted Array**

Suppose an array of length n sorted in ascending order is rotated between 1 and n times. Notice that rotating an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums of unique elements, return the minimum element of this array. You must write an algorithm that runs in O(log n) time.

1. Since the array is partially sorted, binary search is an optimal approach.

2. If the middle element 'mid' is greater than the rightmost element 'right', the minimum must be in the right half. If the middle element is less than or equal to the rightmost element, the minimum is in the left half, including possibly 'mid'.

3. If nums[mid]>nums[right], move the left boundary to mid+1. Otherwise, move the right boundary to mid.

4. The search terminates when the left pointer converges with the right, pointing to the minimum element.

Time Complexity: O(logn)

Space Complexity: O(1)

int findMin(vector<int>& nums) {

int left = 0, right = nums.size() - 1;

while (left < right) {

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

if (nums[mid] > nums[right]) {

left = mid + 1;

}

else {

right = mid;

}

}

return nums[left];

}

**Kth Largest Element in an Array**

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?

Quickselect Approach:

1. Rearrange the array so that elements less than or equal to the pivot are on the left and those greater are on the right. Returns the final position of the pivot.

2. Recursively find the k-th largest element by narrowing down the search space based on the pivot's position. Adjusts the search range until the k-th position is located.

int partition(vector<int>& nums, int start, int end) {

int pivotValue = nums[end];

int smallerElementIndex = start;

for (int i = start; i < end; i++)

{

if (nums[i] <= pivotValue) {

swap(nums[i], nums[smallerElementIndex]);

smallerElementIndex++;

}

}

swap(nums[smallerElementIndex], nums[end]);

return smallerElementIndex;

}

int quickSelect(vector<int>& nums, int left, int right, int k) {

if (left == right) {

return nums[left];

}

int pivotIndex = partition(nums, left, right);

if (k == pivotIndex)

{

return nums[k];

}

else if (k < pivotIndex)

{

return quickSelect(nums, left, pivotIndex - 1, k);

}

else

{

return quickSelect(nums, pivotIndex + 1, right, k);

}

}

int mainQuickSelect() {

vector<int> nums = { 3, 2, 1, 5, 6, 4 };

int k = 2, n = nums.size();

int result = quickSelect(nums, 0, n - 1, n - k);

}

**Container With Most Water**

You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return the maximum amount of water a container can store.

Notice that you may not slant the container.

Two-pointer Approach

1. Start with two pointers, one at the beginning (left) and one at the end (right) of the array.

2. Compute the area formed between the two pointers as 'area = ((right-start) \* min(height[left],height[right]))'.

3. Move the pointer which is at the shorter line inward, since the height of the container is limited by the shorter line.

4. Keep track of the maximum are found during the iteration.

Time Complexity: O(n)

Space Complexity: O(1)

int maxArea(vector<int>& height) {

int n = height.size();

int left = 0, right = n - 1;

int maxArea = 0;

while (left < right)

{

int currentArea = (right - left) \* min(height[left], height[right]);

maxArea = max(currentArea, maxArea);

if (height[left] < height[right]) left++;

else right--;

}

return maxArea;

}

**Insert Intervals**

You are given an array of non-overlapping intervals intervals where intervals[i] = [starti, endi] represent the start and the end of the ith interval and intervals is sorted in ascending order by starti. You are also given an interval newInterval = [start, end] that represents the start and end of another interval.

Insert newInterval into intervals such that intervals is still sorted in ascending order by starti and intervals still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return intervals after the insertion.

1. If the current interval's end is less than 'newInterval''s start point, it means the current interval is before 'newInterval' and does not overlap. Add it directly to the result list.

2. If the current interval's start is less than 'newInterval''s end point, it means the current interval overlaps with 'newInterval'. Merge it by updating 'newInterval''s start and end points to encompass the overlappint interval.

3. If the current interval's start point is greater than 'newInterval''s end point, it means the current interval is after 'newInterval' and does not overlap. Add it directly to the result list.

Time Complexity: O(n)

Space Complexity: O(n)

vector<vector<int>> insertInterval(vector<vector<int>>& intervals, vector<int>& newInterval) {

vector<vector<int>> result;

int i = 0;

int n = intervals.size();

while (i < n && intervals[i][1] < newInterval[0])

{

result.push\_back(intervals[i++]);

}

while (i < n && intervals[i][0] <= newInterval[1])

{

newInterval[0] = min(newInterval[0], intervals[i][0]);

newInterval[1] = max(newInterval[1], intervals[i][1]);

i++;

}

result.push\_back(newInterval);

while (i < n)

{

result.push\_back(intervals[i++]);

}

return result;

}

**Merge Intervals**

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.

1. Sort the intervals based on their starting points in ascending order.

2. Create a new array to store the merged intervals.

3. Compare each interval with the last interval in the result array.

If the current interval's starting point is less than or equal to the ending point of the last interval in the result array, merge them.

If the current interval's starting point is greater than the ending point of the last interval in the result array, add the current interval to the result array as a separate interval.

Time Complexity: O(nlogn)

Space Complexity: O(n)

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

if (intervals.empty())

{

return {};

}

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

vector<vector<int>> merged;

merged.push\_back(intervals[0]);

for (int i = 0; i < intervals.size(); i++)

{

auto& last = merged.back();

if (intervals[i][0] <= last[1])

{

last[1] = max(last[1], intervals[i][1]);

}

else {

merged.push\_back(intervals[i]);

}

}

return merged;

}

**Non-decreasing Array**

Given an array nums with n integers, your task is to check if it could become non-decreasing by modifying at most one element.

We define an array is non-decreasing if nums[i] <= nums[i + 1] holds for every i (0-based) such that (0 <= i <= n - 2).

1. Arrays of size 2 or less are always non-decreasing, return true.

2. Iterate through the array, checking if any element is greater than the next.

3. If such a case is found and no modification has been made yet, modify the array to maintain non-decreasing order. Decide whether to adjust the current element or the next based on surrounding values.

4. If a second modification is needed, return false. If the loop completes with at most one modification, return true.

bool checkPossibility(vector<int>& nums) {

int n = nums.size();

bool modified = false;

for (int i = 0; i < n - 1; ++i) {

if (nums[i] > nums[i + 1]) {

if (modified) return false;

modified = true;

// This will cause an out-of-bounds error when i == 0

if (nums[i - 1] <= nums[i + 1]) {

nums[i] = nums[i + 1];

}

else {

nums[i + 1] = nums[i];

}

}

}

return true;

}

**Rotate Array**

Given an integer array nums, rotate the array to the right by k steps, where k is non-negative.

1. Since rotating an array of length n by k steps is the same as rotating it by 'k%n' steps, first compute k=k%n to avoid unnecessary rotations.

2. Reverse the order of all elements in the array.

3. Place the first k elements in the correct order after the array rotation.

4. Restore the correct order for the remaining part of the array.

Time Complexity: O(n)

Space Complexity: O(1)

void reverse(vector<int>& nums, int start, int end) {

while (start < end)

{

swap(nums[start], nums[end]);

start++;

end--;

}

}

void rotate(vector<int>& nums, int k) {

int n = nums.size();

k = k % n;

reverse(nums, 0, n - 1);

reverse(nums, 0, k - 1);

reverse(nums, k, n - 1);

}

**Rotate Image**

You are given an n x n 2D matrix representing an image, rotate the image by 90 degrees (clockwise).

You have to rotate the image in-place, which means you have to modify the input 2D matrix directly. DO NOT allocate another 2D matrix and do the rotation.

1.The nested loops iterate through each element above the main diagonal and swap it with its corresponding element in the opposite position across the diagonal. This operation converts rows into columns.

2.The reverse() function is used to reverse the order of elements in each row of the matrix. This reordering gives the desired 90-degree clockwise rotation.

Time Complexity: O(N^2)

Space Complexity: O(1)

void rotateImage(vector<vector<int>>& matrix) {

int n = matrix.size();

// Transpose the matrix

for (int i = 0; i < n; i++)

{

for (int j = i + 1; j < n; j++) {

swap(matrix[i][j], matrix[j][i]);

}

}

// Reverse each row

for (int i = 0; i < n; i++)

{

reverse(matrix[i].begin(), matrix[i].end());

}

}

**Valid Sudoku**

Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated according to the following rules:

Each row must contain the digits 1-9 without repetition.

Each column must contain the digits 1-9 without repetition.

Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition.

1.Row Check: Create a 9\*9 boolean array row[9][9] where row[i][num-1] indicates whether the digit num has already appeared in the i-th row.

2.Column Check: Create a 9\*9 boolean array column[9][9] where column[j][num-1] indicates whether the digit num has already appeared in the j-th column.

3.Sub-box Check: Create a 9\*9 boolean array subbox[9][9] where subbox[box\_index][num-1] indicates whether digit num has already appeared in the box\_index sub-box. The box\_index is calculated by (i/3)\*3+(j/3).

bool isValidSudoku(vector<vector<char>>& board) {

bool row[9][9] = { false };

bool column[9][9] = { false };

bool subbox[9][9] = { false };

for (int i = 0; i < 9; i++)

{

for (int j = 0; j < 9; j++) {

if (board[i][j] != '.')

{

int num = board[i][j] - '1';

int boxIndex = (i / 3) \* 3 + (j / 3);

if (row[i][num] || column[j][num] || subbox[boxIndex][num])

{

return false;

}

row[i][num] = true;

column[j][num] = true;

subbox[boxIndex][num] = true;

}

}

}

return true;

}

**Single Number**

Given an integer array nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once. You can return the answer in any order.

You must write an algorithm that runs in linear runtime complexity and uses only constant extra space.

Approach 1: Hash Map

Approach 2: Bit Manipulation

1. By performing XOR on all elements of the array, pairs of equal elements cancel each other out, leaving the XOR result of the two elements that appear only once.

2. From the XOR result, extract the lowest bit that is set to 1. This bit differentiates the two unique elements and helps in separating them.

3. By checking diffBit & current element, the elements can be divided into two groups. Those with diffBit set to 1 belong to one group, while those with diffBit set to 0 belong to another. Since diffBit is derived from the XOR of the two unique elements, these two elements will fall into different groups. Performing XOR within each group will isolate and identify the two elements that appear only once.

Time Complexity: O(N)

Space Complexity: O(1)

vector<int> singleNumber(vector<int>& nums) {

int xorResult = 0;

//XOR all numbers to get the XOR of the two unique numbers

for (int num : nums) {

xorResult ^= num;

}

//Find a set bit (any bit that is 1) in xorResult

int diffBit = xorResult & (-xorResult);

//Separate numbers into two groups and XOR within each group

int num1 = 0, num2 = 0;

for (int num : nums) {

if (num & diffBit) {

num1 ^= num;

}

else {

num2 ^= num;

}

}

return { num1, num2 };

}

**Maximum XOR of Two Numbers in an Array**

Given an integer array nums, return the maximum result of nums[i] XOR nums[j], where 0 <= i <= j < n.

1. Each number in the array is inserted into a Trie, with bits inserted from the most significant (31st) to the least significant (0th). Each node represents a bit: 0 to the left, 1 to the right.

2. For each number, the Trie is traversed to find another number that maximizes the XOR result. The search prioritizes paths with bits opposite to the current number's bit to maximize XOR. If such a path exists, it is chosen; otherwise, the available path is followed. If the opposite bit exists, the XOR value is left-shifted and 1 is appended; if not, the XOR value is left-shifted and the current bit is appended.

3. Trie allows efficient bitwise comparisons. The goal is to find paths in the Trie that differ as much as possible from the current number's path to maximize XOR. The algorithm starts from the highest bit to ensure significant contributions to the XOR result.

Time Complexity: O(n×L)

Space Complexity: O(n×L)

struct TrieNode {

TrieNode\* children[2] = { nullptr,nullptr };

};

TrieNode\* buildTrie(const vector<int>& nums) {

TrieNode\* root = new TrieNode();

for (int num : nums) {

TrieNode\* node = root;

for (int i = 31; i >= 0; i--) {

int bit = (num >> i) & 1;

if (node->children[bit] == nullptr) {

node->children[bit] = new TrieNode();

}

node = node->children[bit];

}

}

return root;

}

int findMaximumXOR(const vector<int>& nums) {

TrieNode\* root = buildTrie(nums);

int max\_xor = 0;

for (int num : nums) {

TrieNode\* node = root;

int current\_xor = 0;

for (int i = 31; i >= 0; --i) {

int bit = (num >> i) & 1;

if (node->children[1 - bit] != nullptr) {

current\_xor = (current\_xor << 1) | 1;

node = node->children[1 - bit];

}

else {

current\_xor = (current\_xor << 1);

node = node->children[bit];

}

}

max\_xor = max(max\_xor, current\_xor);

}

return max\_xor;

}

**UTF-8 Validation**

Given an integer array data representing the data, return whether it is a valid UTF-8 encoding. A character in UTF8 can be from 1 to 4 bytes long, subjected to the following rules:

1.For a 1-byte character, the first bit is a 0, followed by its Unicode code.

2.For an n-bytes character, the first n bits are all one's, the n + 1 bit is 0, followed by n - 1 bytes with the most significant 2 bits being 10.

1. Use a bitwise AND operation with a mask to identify the type of UTF-8 character (1-byte, 2-byte, 3-byte, or 4-byte).

2. Ensure that the array has enough remaining bytes to match the required length for the identified UTF-8 character type. If not, return false.

3. For the bytes following the first, use a bitwise AND operation with a mask to check if they conform to the 10xxxxxx pattern. If any byte doesn't match, return false.

Time Complexity: O(n)

Space Complexity: O(1)

bool validUtf8(vector<int>& data) {

int n = data.size();

int i = 0;

while (i < n) {

int num\_bytes = 0;

switch (data[i] & 0b11111000) {

case 0b11110000: // 4-byte

num\_bytes = 4;

break;

case 0b11100000: // 3-byte

num\_bytes = 3;

break;

case 0b11000000: // 2-byte

num\_bytes = 2;

break;

case 0b00000000: // 1-byte(ASCII)

num\_bytes = 1;

break;

default:

return false;

}

if (i + num\_bytes > n) {

return false;

}

for (int j = 1; j < num\_bytes; ++j) {

if ((data[i + j] & 0b11000000) != 0b10000000) {

return false;

}

}

i += num\_bytes;

}

return true;

}

**Count Primes**

Given an integer n, return the number of prime numbers that are strictly less than n.

Sieve of Eratosthenes Algorithm

1. Create a boolean array 'isPrime' of size n, initialized to true. 'isPrime[i]' will be true if i is a prime number, otherwise false.

2. Start with the first prime number 2. Mark all its multiples as non-prime.

3. Move to next number in the list and repeat the marking process for its multiples. Continue this process until you've processed all numbers up to the square root of n.

4. Finally, count the number of true values in the isPrime array, which represents the prime numbers.

Time Complexity: O(nloglogn)

Space Complexity: O(n)

int countPrimes(int n) {

if (n <= 2) return 0;

vector<bool> isPrime(n, true);

isPrime[0] = isPrime[1] = false; // 0 and 1 are not prime numbers

for (int i = 2; i \* i < n; ++i) {

if (isPrime[i]) {

for (int j = i \* i; j < n; j += i) {

isPrime[j] = false;

}

}

}

return count(isPrime.begin(), isPrime.end(), true);

}

About Sieve of Eratosthenes Algorithm

1. The Sieve of Eratosthenes initializes an array, assuming all numbers less than n (except 0 and 1) are prime (true). Through multiplication and addition, non-prime numbers are identified and marked as false in the array. This method ensures that every non-prime number is eventually marked, even though the marking process occurs in a non-sequential manner.

2. It is observed that the sum of a prime number and a non-prime number always results in a non-prime number. Therefore, in the early iterations, the marking process (isPrime[j] = false;) is more intensive, with more values being marked. As iterations progress, fewer marking operations are needed, as many values have already been marked false in previous steps, leading to a faster execution in later stages.

**Product of Array Except Self**

Given an integer array nums, return an array answer such that answer[i] is equal to the product of all the elements of nums except nums[i]. The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer. You must write an algorithm that runs in O(n) time and without using the division operation.

1. 'answer' array is initialized where 'answer[i]' stores the product of all elements to the left of nums[i].

2. Traverse the array from the left to right to fill the answer array with left products.

3. Traverse the array from right to left, calculating the right products and simultaneously updating the answer array to include the final result.

Time Complexity : O(n)

Space Complexity : O(1)

vector<int> productExceptSelf(vector<int>& nums) {

int n = nums.size();

vector<int> answer(n, 1);

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

{

answer[i] = answer[i - 1] \* nums[i - 1];

}

int right\_product = 1;

for (int i = n-1; i >=0 ; i--)

{

answer[i] \*= right\_product;

right\_product \*= nums[i];

}

return answer;

}

**Subarray Sum Equals K**

Given an array of integers nums and an integer k, return the total number of subarrays whose sum equals to k.

A subarray is a contiguous non-empty sequence of elements within an array.

Cumulative Sum Approach

1. Initialize a hash map to store cumulative sums and their frequencies.

2. Initialize a variable to keep the cumulative sum and another to count the number of subarrays.

3. Iterate through the array, updating the cumulative sum and using the hash map to check and update the count of subarrays.

Time Complexity : O(n)

Space Complexity : O(n)

int subarraySum(vector<int>& nums, int k) {

int count = 0;

int cumulativeSum = 0;

unordered\_map<int, int> cumulativeSumFreq;

cumulativeSumFreq[0] = 1;

for (int num : nums) {

cumulativeSum += num;

if (cumulativeSumFreq.find(cumulativeSum - k) != cumulativeSumFreq.end()) {

count += cumulativeSumFreq[cumulativeSum - k];

}

cumulativeSumFreq[cumulativeSum]++;

}

return count;

}

**Majority Element**

Given an integer array of size n, find all elements that appear more than ⌊ n/3 ⌋ times.

Moore's Voting Algorithm

Moore's Voting Algorithm is designed to find elements that appear more than n/3 times in an array. It maintains two candidates and their counters, adjusting them as it traverses the array.

1. Initialize two candidates and their respective counters.

2. Traverse the array:

If the current element matched one of the candidates, increment the corresponding counter.

If the current element does not match either candidate: If a counter is zero, replace the corresponding candidate with the current element and set the counter to one; If both counters are non-zero, decrement both counters.

3. Traverse the array again to confirm whether the selected candidates appear more than n/3 times.

Time Complexity: O(n)

Space Complexity: O(1)

Risk:

1. If the threshold is reduced further (e.g. n/4, n/10), the effectiveness of the algorithm decreases, as more than two candidates may be required to accurately capture all elements meeting the new threshold.

2. The process of replacing candidates when their counters reach zero might cause valid candidates to be lost, potentially leading to inaccurate results if not carefully validated in the second pass.

vector<int> majorityElement(vector<int>& nums) {

int candidate1 = 0, candidate2 = 1, count1 = 0, count2 = 0;

// First pass: find the candidates

for (int num : nums) {

if (num == candidate1) count1++;

else if (num == candidate2) count2++;

else if (count1 == 0) {

candidate1 = num;

count1 = 1;

}

else if (count2 == 0) {

candidate2 = num;

count2 = 1;

}

else {

count1--;

count2--;

}

}

// Second pass: verify the candidates

count1 = count2 = 0;

for (int num : nums) {

if (num == candidate1) count1++;

else if (num == candidate2) count2++;

}

vector<int> result;

int n = nums.size();

if (count1 > n / 3) result.push\_back(candidate1);

if (count2 > n / 3) result.push\_back(candidate2);

return result;

}

Hash Map Approach

1. Traverse the array and use a hash map to record the frequency of each element.

2. Iterate through the hash map to find elements that appear more than n/k times.

Time Complexity: O(n)

Space Complexity: O(n)

vector<int> majorityElementHashMap(vector<int>& nums, int k) {

unordered\_map<int, int> freq\_map;

vector<int> result;

int n = nums.size();

for (int num : nums)

{

freq\_map[num]++;

}

for (auto& pair : freq\_map)

{

int num = pair.first;

int count = pair.second;

if (count > n / k)

{

result.push\_back(num);

}

}

return result;

}

**Find the Duplicate Number**

Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive. There is only one repeated number in nums, return this repeated number. You must solve the problem without modifying the array nums and uses only constant extra space.

Approach 1: Binary Search

Approach 2: Floyd's (Tortoise and Hare/Cycle-Finding) Algorithm

1. Use two pointers, slow and fast. 'slow' moves one step at a time, while 'fast' moves two steps at a time. If there is a cycle, these two pointers will eventually meet. For example: 1 -> 2 -> 3 -> 4 -> 5 -> 3, there is a cycle.

2. Once the intersection is found, reset one pointer to the start of the array, and keep the other at the intersection point. Move both pointers one step at a time; the point where they meet is the entrance to the cycle, which is the duplicate number.

Time Complexity: O(N)

Space Complexity: O(1)

About:

1. When the slow and fast pointers first meet, it confirms that there is a cycle (or loop) within the structure, which, in this case, represents a duplicate element in the array. This first meeting point is somewhere inside the cycle, but not necessarily at the entry point of the cycle.

2. After the first encounter, the slow pointer is reset to the start of the array (nums[0]), while the fast pointer remains at the meeting point within the cycle. Both pointers then move one step at a time. Since the distance from the meeting point to the entry point of the cycle is equal to the distance from the start of the array to the entry point, the two pointers will inevitably meet at the cycle's entry point. This meeting point is the duplicate element being sought.

int findDuplicate(vector<int>& nums) {

int n = nums.size() - 1;

for (int num : nums) {

if (num<1 || num>n) {

throw invalid\_argument("Input array does not meet the problem's constraints.");

}

}

int slow = nums[0];

int fast = nums[0];

// Finding the intersection point of two runners

do

{

slow = nums[slow];

fast = nums[nums[fast]];

} while (slow != fast);

// Finding the entrance to the cycle

slow = nums[0];

while (slow != fast)

{

slow = nums[slow];

fast = nums[fast];

}

return slow;

}

**Game of Life**

According to Wikipedia's article: "The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

The board is made up of an m x n grid of cells, where each cell has an initial state: live (represented by a 1) or dead (represented by a 0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

Any live cell with fewer than two live neighbors dies as if caused by under-population.

Any live cell with two or three live neighbors lives on to the next generation.

Any live cell with more than three live neighbors dies, as if by over-population.

Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

1. Set up variables m and n for the grid dimensions and create next\_state to store the next generation of the grid.

2. Use directions = {-1,0,1} to simplify the calculation of all neighboring cells. 'directions' helps efficiently locate all neighboring cells for each cell in the grid.

3. Loop through each cell (i,j) in the grid. For each cell, use directions to calculate the positions of all eight neighbors and count the number of live neighbors.

4. Determine the next state of each cell based on its live neighbors:

Die if fewer than 2 or more than 3 live neighbors.

Stay alive with 2 or 3 live neighbors.

Come to life if exactly 3 live neighbors.

5. Replace the current grid with next\_state after processing all cells.

void gameOfLife(vector<vector<int>>& board) {

int m = board.size(), n = board[0].size();

vector<vector<int>> next\_state(m, vector<int>(n, 0));

vector<int> directions = { -1,0,1 };

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

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

int live\_neighbors = 0;

for (int x : directions) {

for (int y : directions) {

if (x == 0 && y == 0) continue;

int ni = i + x;

int nj = j + y;

if (ni >= 0 && ni < m && nj >= 0 && nj < n) {

live\_neighbors += board[ni][nj];

}

}

}

if (live\_neighbors == 1)

{

if (live\_neighbors < 2 || live\_neighbors>3) {

next\_state[i][j] = 0;

}

else

{

next\_state[i][j] = 1;

}

}

else

{

if (live\_neighbors == 3) {

next\_state[i][j] = 1;

}

}

}

}

board = next\_state;

}

**Find BLE Array**

Run-length encoding is a compression algorithm that allows for an integer array nums with many segments of consecutive repeated numbers to be represented by a (generally smaller) 2D array encoded. Each encoded[i] = [vali, freqi] describes the ith segment of repeated numbers in nums where vali is the value that is repeated freqi times.

Note: Compression should be done such that the run-length encoded array has the minimum possible length.

For example:

1. Input: encoded1 = [[1, 3], [2, 2]] （nums1 = [1, 1, 1, 2, 2], encoded2 = [[2, 3], [3, 2]] （nums2 = [2, 2, 2, 3, 3]）

2. Expand: nums1 = [1, 1, 1, 2, 2], nums2 = [2, 2, 2, 3, 3]

3. Multiply : proNums = [1\*2, 1\*2, 1\*2, 2\*3, 2\*3] = [2, 2, 2, 6, 6]

4. Encode: proNums = [[2, 3], [6, 2]]

1. Two pointers i and j are initialized to traverse encoded1 and encoded2. Two auxiliary variables remainingFreq1 and remainingFreq2 track the unprocessed frequencies of the current segments.

2. For each pair of segments from encoded1 and encoded2, the minimum of the remaining frequencies (minFreq) is determined. The product of the current values from the two segments is calculated. If the product matches the last segment in the result, merge them by increasing the frequency. Otherwise, a new segment is added to the result.

3. The remaining frequencies are updated by subtracting minFreq. If a segment is fully processed (remainingFreq1 or remainingFreq2 becomes zero), the corresponding pointer (i or j) moves to the next segment.

4. The final run-length encoded array representing the element-wise product is returned.

Optimized Solution:

1. The algorithm avoids redundant multiplications by processing the minimal frequency between corresponding segments from the two encoded arrays at each step. This ensures each multiplication is performed only when necessary.

2. To avoid modifying the input arrays, two auxiliary integer variables remainingFreq1 and remainingFreq2 are used to track the unprocessed portion of the current segment in each array. This approach allows efficient traversal and processing of the segments without altering the original data.

3. The algorithm merges consecutive segments with the same product value to maintain the compressed nature of the run-length encoding, thus minimizing the output array size.

Time Complexity: O(m+n)

Space Complexity: O(1)

vector<vector<int>> findRLEArray(const vector<vector<int>>& encoded1, const vector<vector<int>>& encoded2) {

vector<vector<int>> result;

int i = 0, j = 0;

int remainingFreq1 = 0, remainingFreq2 = 0;

while (i < encoded1.size() && j < encoded2.size()) {

if (remainingFreq1 == 0)

remainingFreq1 = encoded1[i][1];

if (remainingFreq2 == 0)

remainingFreq2 = encoded2[j][1];

int minFreq = min(remainingFreq1, remainingFreq2);

int product = encoded1[i][0] \* encoded2[j][0];

if (!result.empty() && result.back()[0] == product) {

result.back()[1] += minFreq;

}

else {

result.push\_back({ product, minFreq });

}

remainingFreq1 -= minFreq;

remainingFreq2 -= minFreq;

if (remainingFreq1 == 0)

i++;

if (remainingFreq2 == 0)

j++;

}

return result;

}

**Sorted Transformed Array**

Given a sorted integer array nums and three integers a, b and c, apply a quadratic function of the form f(x) = ax2 + bx + c to each element nums[i] in the array, and return the array in a sorted order.

1. Define two pointers at the start and end of the array and determine the filling direction of the result array based on the sign of a.

2. If a >= 0, the graph is an upward-opening parabola, with larger values at the ends and smaller values in the middle. Compare values at the pointers; place the larger value at the end of the result array, then move the corresponding pointer.

If a < 0, the graph is a downward-opening parabola, with smaller values at the ends and larger values in the middle. Compare values at the pointers; place the smaller value at the beginning of the result array, then move the corresponding pointer.

Time Complexity: O(n)

Space Complexity: O(n)

int calculate(int x, int a, int b, int c) {

return a \* x \* x + b \* x + c;

}

vector<int> sortTransformedArray(vector<int>& nums, int a, int b, int c) {

int n = nums.size();

vector<int> result(n);

int left = 0, right = n - 1;

// If a >= 0, fill from the end; otherwise, from the start

int index = (a >= 0) ? n - 1 : 0;

while (left <= right) {

int leftValue = calculate(nums[left], a, b, c);

int rightValue = calculate(nums[right], a, b, c);

if (a >= 0) { // Fill from the end

if (leftValue >= rightValue) {

result[index--] = leftValue;

left++;

}

else {

result[index--] = rightValue;

right--;

}

}

else { // Fill from the start

if (leftValue <= rightValue) {

result[index++] = leftValue;

left++;

}

else {

result[index++] = rightValue;

right--;

}

}

}

return result;

}