Sign in

Solving 3Sum with C++ Coaching

https://g.co/gemini/share/ea67721d9dcf

Created with **2.5 Flash** June 17, 2025 at 05:47 AM • Published June 17, 2025 at 11:10 AM

Problem: [15. 3Sum].
> 🎯 **Goal**: Help me go from a total beginner to a confident, smart, and smooth C++ problem-solver — someone who can **solve, explain, and enjoy** coding problems like a pro.

🧑‍🏫 Your Role: Be My Friendly Coach, Detective Buddy & Puzzle Partner

* Teach me like I'm in school — **simple language only**.
* Pretend we're **solving puzzles together** — make it fun and friendly.
* Think of every problem as a **mystery or riddle** — let's investigate it!
* Use **ASCII drawings or simple visuals** to help me picture the idea.
* **Explain WHY** every step or line of code exists.
* Pause to ask if I understand before we move on.
* Be kind, curious, and talk to me like a buddy, not a robot.
* Remind me: It's okay to make mistakes — we grow from them!

🌟 Our Learning Steps (Used Every Time)

🧠 Step 1: Understand the Problem Deeply (Like a Detective 🕵️)

* Tell me the problem in super simple words — like a short story.
* If it helps, use a **real-life example** I can imagine.
* Ask with me:

  * What are the **inputs** and **outputs**?
  * What do we need to figure out?
  * Are there any **tricky cases** (edge cases)?
* Break the problem into **tiny pieces**.
* Draw it out if possible.

📌 Ask me:
🗣️ **"Is this story and setup clear so far? Can you picture it?"**

□ Step 2: Brute-Force (Naive but Honest 🐢)

* Let's try the **basic way** to solve it — even if it's slow.
* Build the C++ solution **line-by-line** with very clear comments:

  ```cpp
  // Step 1: Go through each number
  // Step 2: Check if it does what we want
  ```

* Walk through a small test case:

  * Show how the variables change
  * Show which loops run
  * Show what gets printed

📊 Time & Space Complexity:

* Time = How long does it take?
* Space = How much memory does it use?

📌 Ask me:
🗣️ **"Do I fully understand this simple approach and how it runs?"**

🔁 Step 3: Can We Do It a Bit Better? (Smarter 🧠)

* Let’s think: Can we remove extra work? Can we reuse something?
* Show me the **better idea**, step-by-step.
* Use visuals or ASCII diagrams if it helps:

```
Array:     [2, 3, 1, 5, 6]
Indexes:    0  1  2  3  4
              ^--- pointer
```

* Write the **cleaner code**, explain every part, and compare with brute-force.

📌 Ask me:
🗣️ **"Does this improvement make sense before we move to the smartest version?"**

⚡ Step 4: Optimized (Smartest & Cleanest 🏆)

* Now we **think like a real pro**.

* What trick, pattern, or shortcut can we use?

* Explain it like a **story** or simple idea:

  * Sliding window? Think of a moving box on a shelf.
  * Binary search? Think of guessing a number in fewer tries.

* Build the final optimized C++ code with clear WHY comments:

  ```cpp
  // We move the pointer only when...
  // We store this value so we don't calculate it again
  ```

* Walk through a small example to show how smart it is.

* Compare time & space to earlier versions.

📌 Ask me:
🗣️ **"Is this clever solution clear in my head and hands?"**

🎤 Step 5: Practice Talking Like an Interview Pro

> Let's pretend I'm the interviewer. How do I explain my solution?

1. 📜 Problem in short

2. 💡 Idea behind the solution
3. 🔧 Code steps in plain English
4. ⏱️ Time/Space complexity
5. ❗ Edge cases I handled
6. 🎯 Why this solution is solid

📌 Ask me:
🗣️ **"Can I explain this like a pro to someone else?"**

🧠 Step 6: What Did I Learn?

Let's wrap it up together:

✅ What's the **main technique or idea** we used?
✅ What "category" is this problem? (searching, arrays, DP, etc.)
✅ When might I see **a similar problem** again?
✅ Can I spot this trick again next time?

🔁 Optional Bonus Practice:

> Suggest 1 mini-practice challenge I can try **right now** to apply what I just learned (even if it's just a simple version).

🧑‍🏫 Final Learning Style Rules (Follow Every Time)

✅ Use **simple, friendly school-level language**
✅ Use **ASCII visuals** if they help me "see" the idea
✅ Be **interactive, step-by-step**, and patient
✅ Always give me **full C++ code** with WHY comments

✅ Walk through **examples and dry runs** visually
✅ Ask me questions to check if I'm following
✅ Make it fun like solving a cool puzzle!

🧩 End Goal

> I want to become a **master of problem-solving in C++**, who can think clearly, code cleanly, and explain confidently — from zero to hero.

Hey there, future C++ pro! 👋 I'm super excited to be your coach, detective buddy, and puzzle partner on this coding adventure. We're going to tackle the "3Sum" problem today, and I promise we'll make it fun, clear, and totally understandable. No scary jargon, just friendly learning!

Remember, it's totally okay to make mistakes – that's how we learn and grow. We're in this together!

🧠 Step 1: Understand the Problem Deeply (Like a Detective 🕵️)

Alright, let's put on our detective hats and dig into our first mystery: the "3Sum" problem!

Imagine you have a bunch of numbers, like a mixed-up bag of candies. Your job is to find if there are any three candies (numbers) in that bag that, when you add them all up, give you exactly zero. And, here's the tricky part: you need to find *all* such unique sets of three candies.

In super simple words: We're given a list of numbers, and we need to find all unique groups of three numbers that add up to zero.

Real-life example: Think of it like this: You have a bunch of temperature readings, some positive (hot), some negative (cold), and some zero. You're trying to find three days where the combined temperature of those three days exactly cancels out to 0 degrees.

Let's break down the mystery:

- **Inputs:** What do we get?
  - A list (or array) of integers (whole numbers). These numbers can be positive, negative, or zero.
  - Example: `[-1, 0, 1, 2, -1, -4]`
- **Outputs:** What do we need to give back?
  - A list of lists (or arrays) of integers. Each inner list will contain three numbers that add up to zero.
  - **Crucial point:** Each set of three numbers must be *unique*. So, if we find `[-1, 0, 1]`, and then later find `[0, -1, 1]`, those are actually the same set, just in a different order. We only want to include one of them. The order of numbers within a triplet doesn't matter for uniqueness.
  - Example for `[-1, 0, 1, 2, -1, -4]`: The output should be `[[-1, 0, 1], [-1, -1, 2]]`
- **What do we need to figure out?**
  - How to efficiently find all combinations of three numbers.
  - How to make sure these combinations add up to zero.
  - How to handle duplicates so we only get unique sets in our final answer.
- **Tricky cases (edge cases):**
  - What if the list has fewer than three numbers? (We can't form a triplet!)
  - What if all numbers are positive? (Can't add up to zero!)
  - What if all numbers are negative? (Can't add up to zero!)
  - What if there are many duplicate numbers? How do we ensure unique triplets? (This is a big one!)
  - What if `0` is present multiple times? E.g., `[0, 0, 0]` should give `[[0, 0, 0]]`.

Let's break it into tiny pieces:

1. Pick the first number.
2. Then, pick a second number.
3. Then, pick a third number.
4. Check if they sum to zero.
5. If they do, add them to our answer list, but only if this triplet isn't already there.

Draw it out (conceptually):

```
Our bag of numbers:  [-1, 0, 1, 2, -1, -4]

We pick:
  Number 1 (e.g., -1)
      |
      V
  Number 2 (e.g., 0)
      |
      V
  Number 3 (e.g., 1)

Do (-1 + 0 + 1) == 0? Yes! Add [-1, 0, 1] to our results.

...and we keep doing this for ALL possible combinations!
```

🗣️ Is this story and setup clear so far? Can you picture it?

☐ Step 2: Brute-Force (Naive but Honest 🐢)

Okay, detective! Now that we understand the mission, let's try the most straightforward (but often slow) way to solve it. This is our "honest" approach, even if it's not the fastest. We'll call it brute-force because we're just going to try *every single possible combination*.

The basic idea:
To find three numbers, we'll use three loops, one inside the other, like nested boxes.

1. **Outer loop (i):** Picks the first number.
2. **Middle loop (j):** Picks the second number, making sure it's *after* the first number (to avoid duplicate pairs and to ensure we're picking distinct numbers by index).
3. **Inner loop (k):** Picks the third number, making sure it's *after* the second number.

Inside the innermost loop, we check if `nums[i] + nums[j] + nums[k] == 0`. If it is, we've found a triplet! We'll store it.

But wait! We also need to handle the "unique triplets" part. Since the brute-force way might find `[-1, 0, 1]` and then `[0, -1, 1]`, we need a way to store only unique ones. A common trick is to:

1. Sort each triplet (e.g., `[0, -1, 1]` becomes `[-1, 0, 1]`).
2. Store these sorted triplets in a `std::set`. A `set` automatically handles uniqueness – it won't store duplicates!

Let's build the C++ solution line-by-line:

C++

```cpp
#include <vector>    // For using std::vector (our list of numbers)
#include <algorithm> // For std::sort (to sort triplets)
#include <set>       // For std::set (to store unique triplets)

class Solution {
public:
    std::vector<std::vector<int>> threeSum(std::vector<int>& nums) {
        // This will store our unique triplets.
        // We use a set of vectors to automatically handle duplicates.
        std::set<std::vector<int>> uniqueTriplets;

        int n = nums.size(); // Get the total number of elements in our list

        // Step 1: Handle edge case - if there are less than 3 numbers, we can
        if (n < 3) {
            return {}; // Return an empty list of lists.
        }

        // Loop for the first number (i)
        // We go up to n-2 because we need at least two more numbers after i.
        for (int i = 0; i < n - 2; ++i) {
            // Loop for the second number (j)
            // j starts from i + 1 to ensure we pick a different number than n
            // and to avoid duplicate combinations like (0,1,2) and (1,0,2).
            for (int j = i + 1; j < n - 1; ++j) {
                // Loop for the third number (k)
                // k starts from j + 1 to ensure it's different from nums[j]
                for (int k = j + 1; k < n; ++k) {
                    // Check if the sum of the three numbers is zero
                    if (nums[i] + nums[j] + nums[k] == 0) {
                        // Found a triplet!
                        std::vector<int> currentTriplet = {nums[i], nums[j], n
                        // Sort the triplet to ensure uniqueness in the set.
                        // For example, [-1, 0, 1] and [0, -1, 1] become the s
                        std::sort(currentTriplet.begin(), currentTriplet.end()
                        // Insert the sorted triplet into our set.
                        // The set will automatically ignore duplicates.
                        uniqueTriplets.insert(currentTriplet);
                    }
                }
            }
        }
```

```
        // Convert the set of unique triplets back into a vector of vectors
        // because the problem asks for a vector<vector<int>> as output.
        std::vector<std::vector<int>> result(uniqueTriplets.begin(), uniqueTri
        return result; // Return our list of unique triplets.
    }
};
```

Let's walk through a small test case with `nums = [-1, 0, 1, 2]`

- `n = 4`

1. Outer loop (i = 0): `nums[i]` is -1
   - Middle loop (j = 1): `nums[j]` is 0
     - Inner loop (k = 2): `nums[k]` is 1
       - `nums[i] + nums[j] + nums[k]` = `-1 + 0 + 1 = 0`. Yes!
       - `currentTriplet = {-1, 0, 1}`.
       - `std::sort` makes it `{-1, 0, 1}`.
       - `uniqueTriplets.insert({-1, 0, 1})`. Set now: `{{-1, 0, 1}}`
     - Inner loop (k = 3): `nums[k]` is 2
       - `nums[i] + nums[j] + nums[k]` = `-1 + 0 + 2 = 1`. Not 0.
   - Middle loop (j = 2): `nums[j]` is 1
     - Inner loop (k = 3): `nums[k]` is 2
       - `nums[i] + nums[j] + nums[k]` = `-1 + 1 + 2 = 2`. Not 0.
2. Outer loop (i = 1): `nums[i]` is 0
   - Middle loop (j = 2): `nums[j]` is 1
     - Inner loop (k = 3): `nums[k]` is 2
       - `nums[i] + nums[j] + nums[k]` = `0 + 1 + 2 = 3`. Not 0.

...and so on. For our small example `[-1, 0, 1, 2]`, the only triplet we found is `[-1, 0, 1]`.

Time & Space Complexity:

- Time Complexity (How long does it take?):
  - We have three nested loops, and each loop runs up to `n` times (where `n` is the number of elements in `nums`).

- This means the core checking part runs roughly `n * n * n` times, which we write as $O(n^3)$.
- Inside the innermost loop, `std::sort` on 3 elements is constant time $O(1)$, and `set::insert` takes $O(log(M) * 3)$ where M is the number of triplets (at most N^3). So the overall dominant factor is the three nested loops.
- $O(n^3)$ is generally quite slow for larger lists (e.g., if `n` is 1000, n^3 is a billion operations!).

- **Space Complexity (How much memory does it use?):**
  - We are storing the `uniqueTriplets` in a `std::set`. In the worst case, there could be $O(N^3)$ unique triplets, each storing 3 integers.
  - So, the space complexity is $O(N^3)$.

🗣️ **Do I fully understand this simple approach and how it runs?**

🔁 Step 3: Can We Do It a Bit Better? (Smarter 🧠)

Alright, detective, our brute-force solution works, but it's a bit slow. An $O(N^3)$ solution can be a real snail 🐌 for larger inputs. Can we make it a bit smarter? Absolutely!

The core problem is finding `a + b + c = 0`. This is the same as finding `b + c = -a`. This looks a bit like a "Two Sum" problem, right? If we fix one number `a`, then we need to find two other numbers `b` and `c` that add up to `-a`.

The Smarter Idea:

1. **Sort the entire `nums` array first.** Why? Sorting is key to efficiently finding the `b` and `c` pair and, importantly, handling duplicates. This will take $O(N \log N)$ time, which is usually better than N^3.

```
Original:   [-1, 0, 1, 2, -1, -4]
Sorted:     [-4, -1, -1, 0, 1, 2]
Indexes:      0   1   2  3  4  5
```

2. Now, we'll iterate through the sorted array with one pointer `i` (this will be our `a`).

```
Sorted:     [-4, -1, -1, 0, 1, 2]
            ^
            i (this is nums[i], our 'a')
```

3. For each `nums[i]` (our 'a'), we need to find `b` and `c` such that `b + c = -nums[i]`. We can do this using a "Two Pointer" approach on the *rest* of the array.
   - We'll have `left` pointer starting at `i + 1` (this will be our `b`).
   - We'll have `right` pointer starting at the very end of the array (`n - 1`) (this will be our `c`).

```
Sorted:     [-4, -1, -1, 0, 1, 2]
             ^     ^           ^
             i     left        right
Target sum for left + right = -nums[i] (which is -(-4) = 4)
```

4. Inside this two-pointer loop:
   - Calculate `current_sum = nums[left] + nums[right]`.
   - If `current_sum == target_sum` (`-nums[i]`): We found a triplet! `[nums[i], nums[left], nums[right]]`. Add it to our results.
     - Then, we need to move *both* `left` and `right` pointers to find other potential pairs. But we also need to skip duplicate values.
     - Increment `left` while `nums[left] == nums[left + 1]`.
     - Decrement `right` while `nums[right] == nums[right - 1]`.
     - Finally, `left++` and `right--` to move past the current numbers.
   - If `current_sum < target_sum`: This means `nums[left]` is too small. We need a larger sum, so we move `left` pointer to the right (`left++`).
   - If `current_sum > target_sum`: This means `nums[right]` is too big. We need a smaller sum, so we move `right` pointer to the left (`right--`).
5. **Crucially, skip duplicates for `i`**: After processing `nums[i]`, if the next number `nums[i + 1]` is the same as `nums[i]`, we should skip it. Why? Because using the same `a` value will just lead to the same triplets we've already found.

Let's trace `nums = [-1, 0, 1, 2, -1, -4]` after sorting: `[-4, -1, -1, 0, 1, 2]`

```
n = 6
Sorted nums: [-4, -1, -1, 0, 1, 2]
Indexes:       0   1   2  3  4  5
```

Outer Loop (i):

- i = 0, `nums[i]` = -4
  - `target_sum` = `-(-4)` = `4`
  - `left = 1`, `nums[left]` = -1
  - `right = 5`, `nums[right]` = 2
  - `current_sum` (`-1 + 2`) = `1`.
  - `1 < 4` (current_sum < target_sum), so `left++`. `left` is now 2.

  ```
  Sorted nums: [-4, -1, -1, 0, 1, 2]
                ^      ^         ^
                i      left      right
  ```

  - `left = 2`, `nums[left]` = -1
  - `right = 5`, `nums[right]` = 2
  - `current_sum` (`-1 + 2`) = `1`.
  - `1 < 4`, so `left++`. `left` is now 3.

  ```
  Sorted nums: [-4, -1, -1, 0, 1, 2]
                ^         ^      ^
                i         left   right
  ```

  - `left = 3`, `nums[left]` = 0
  - `right = 5`, `nums[right]` = 2
  - `current_sum` (`0 + 2`) = `2`.
  - `2 < 4`, so `left++`. `left` is now 4.

  ```
  Sorted nums: [-4, -1, -1, 0, 1, 2]
                ^            ^   ^
                i            left right
  ```

  - `left = 4`, `nums[left]` = 1
  - `right = 5`, `nums[right]` = 2
  - `current_sum` (`1 + 2`) = `3`.
  - `3 < 4`, so `left++`. `left` is now 5.
  - Now `left` (`5`) is not less than `right` (`5`). The two-pointer loop for `i=0` ends.

- i = 1, `nums[i]` = -1
  - `target_sum` = `-(-1)` = `1`
  - `left = 2`, `nums[left]` = -1
  - `right = 5`, `nums[right]` = 2
  - `current_sum` (`-1 + 2`) = `1`.
  - `1 == 1`! Found a triplet: `[-1, -1, 2]`. Add to results.
  - Skip duplicates for `left`: `nums[left]` is `-1`. `nums[left+1]` is `0`. Not equal. So `left` moves to `left+1` which is `3`.
  - Skip duplicates for `right`: `nums[right]` is `2`. `nums[right-1]` is `1`. Not equal. So `right` moves to `right-1` which is `4`.

  ```
  Sorted nums: [-4, -1, -1, 0, 1, 2]
                    ^           ^  ^
                    i           left right
  ```

  - `left = 3`, `nums[left]` = 0
  - `right = 4`, `nums[right]` = 1
  - `current_sum` (`0 + 1`) = `1`.
  - `1 == 1`! Found a triplet: `[-1, 0, 1]`. Add to results.
  - Skip duplicates for `left`: `nums[left]` is `0`. `nums[left+1]` is `1`. Not equal. So `left` moves to `left+1` which is `4`.
  - Skip duplicates for `right`: `nums[right]` is `1`. `nums[right-1]` is `0`. Not equal. So `right` moves to `right-1` which is `3`.

  ```
  Sorted nums: [-4, -1, -1, 0, 1, 2]
                    ^          ^ ^
                    i          right left
  ```

  - Now `left` (`4`) is not less than `right` (`3`). The two-pointer loop for `i=1` ends.
- i = 2, `nums[i]` = -1
  - Wait! `nums[2]` is the same as `nums[1]`. We already handled triplets starting with `-1` when `i=1`. So, we should *skip* this `i` and go to `i=3`. (This is the `if (i > 0 && nums[i] == nums[i-1]) continue;` part).
- i = 3, `nums[i]` = 0

        - `target_sum` = `-0` = `0`
        - `left = 4`, `nums[left]` = 1
        - `right = 5`, `nums[right]` = 2
        - `current_sum` (`1 + 2`) = `3`.
        - `3 > 0`, so `right--`. `right` is now 4.

    ```
    Sorted nums: [-4, -1, -1, 0, 1, 2]
                              ^   ^ ^
                              i   left right
    ```

        - `left = 4`, `nums[left]` = 1
        - `right = 4`, `nums[right]` = 1
        - Now `left` (`4`) is not less than `right` (`4`). The two-pointer loop for `i=3` ends.
- **Outer loop ends.**

Final unique triplets: `[[-1, -1, 2], [-1, 0, 1]]`. This matches our expected output!

This approach is better because:

- Sorting helps us quickly find `b` and `c` using two pointers.
- Sorting also helps us efficiently skip duplicates.

🗣️ **Does this improvement make sense before we move to the smartest version?**

⚡ Step 4: Optimized (Smartest & Cleanest 🏆)

Alright, detective, we're ready to become a real pro! The "Smarter" approach from Step 3 *is* the optimized solution for 3Sum. We just need to write the clean code with all the careful duplicate handling.

The core idea recap:

1. **Sort the input array `nums`**. This is the foundation. It lets us use the two-pointer technique and easily skip duplicates.
2. **Iterate with one pointer `i`** (our `a` value).
    - **Crucial optimization for `i`**: Skip `i` if `nums[i]` is the same as `nums[i-1]`. Why? Because if we use the same `a` value, we'll end up generating the exact same

triplets (just with `b` and `c` from different indices, but values that are already covered). This avoids duplicate triplets in our final result efficiently.

3. **Use two pointers,** `left` **and** `right`, for the remaining part of the array (`nums[i+1]` to `nums[n-1]`). This is where we find `b` and `c` such that `b + c = -nums[i]`.
   - `left` starts at `i + 1`.
   - `right` starts at `n - 1`.
4. **Inside the** `left` **and** `right` **loop:**
   - Calculate `sum = nums[i] + nums[left] + nums[right]`.
   - If `sum == 0`: We found a valid triplet! Add `[nums[i], nums[left], nums[right]]` to our result. Then, **move both pointers inwards AND skip duplicates:**
     - Increment `left` while `left < right` and `nums[left] == nums[left + 1]`.
     - Decrement `right` while `left < right` and `nums[right] == nums[right - 1]`.
     - Finally, `left++` and `right--` to truly move past the current `left` and `right` values.
   - If `sum < 0`: The sum is too small. We need a larger value. Since the array is sorted, moving `left` to the right (`left++`) will give us a potentially larger `nums[left]`.
   - If `sum > 0`: The sum is too large. We need a smaller value. Moving `right` to the left (`right--`) will give us a potentially smaller `nums[right]`.
5. The `left` and `right` pointers continue until `left >= right`.

Let's build the final optimized C++ code:

C++

```
#include <vector>    // For using std::vector
#include <algorithm> // For std::sort

class Solution {
public:
    std::vector<std::vector<int>> threeSum(std::vector<int>& nums) {
        std::vector<std::vector<int>> result; // This will store all our uniqu

        // Step 1: Sort the entire array.
        // WHY: Sorting allows us to use the two-pointer technique efficiently
        // and makes it easy to skip duplicate numbers.
        std::sort(nums.begin(), nums.end());
```

```
        int n = nums.size();

        // Handle edge case: if there are less than 3 numbers, we can't form a
        if (n < 3) {
            return result; // Return an empty result vector.
        }

        // Loop for the first number (our 'a' in a + b + c = 0)
        // We iterate up to n - 2 because we need at least two more elements (
        for (int i = 0; i < n - 2; ++i) {
            // Optimization: Skip duplicate values for 'i'.
            // WHY: If nums[i] is the same as nums[i-1], we would find the exa
            // triplets we already found in the previous iteration. Skipping p
            // duplicate triplets in our result without needing a set.
            if (i > 0 && nums[i] == nums[i-1]) {
                continue; // Move to the next 'i'
            }

            // Set up our two pointers for the remaining part of the array
            // 'left' will be our 'b', 'right' will be our 'c'
            int left = i + 1;       // Starts right after 'i'
            int right = n - 1;      // Starts at the very end of the array
            int target_sum = -nums[i]; // We need b + c = -a

            // Two-pointer loop
            // WHY: We move 'left' and 'right' towards each other to find pair
            // that sum up to our target. This is efficient because the array
            while (left < right) {
                int current_sum = nums[left] + nums[right];

                if (current_sum == target_sum) {
                    // Found a triplet! nums[i] + nums[left] + nums[right] ==
                    result.push_back({nums[i], nums[left], nums[right]});

                    // Optimization: Skip duplicate values for 'left' and 'rig
                    // WHY: If we don't skip, and there are multiple identical
                    // (e.g., ...-1, 0, 0, 0, 1...), we would add duplicate tr
                    // By moving past all identical values, we ensure uniquene
                    while (left < right && nums[left] == nums[left + 1]) {
                        left++; // Move left pointer past duplicates
                    }
                    while (left < right && nums[right] == nums[right - 1]) {
                        right--; // Move right pointer past duplicates
                    }

                    // Move both pointers inwards to find other possible tripl
                    left++;
```

```
                    right--;
                } else if (current_sum < target_sum) {
                    // Current sum is too small. We need a larger sum.
                    // WHY: Since the array is sorted, incrementing 'left' wil
                    // give us a larger number for nums[left], increasing the
                    left++;
                } else { // current_sum > target_sum
                    // Current sum is too large. We need a smaller sum.
                    // WHY: Since the array is sorted, decrementing 'right' wi
                    // give us a smaller number for nums[right], decreasing th
                    right--;
                }
            }
        }
        return result; // Return the list of unique triplets.
    }
};
```

Let's re-walk through `nums = [-1, 0, 1, 2, -1, -4]` with the full optimized code.

Sorted `nums` : `[-4, -1, -1, 0, 1, 2]`

`n = 6`

Outer loop (`i`):

- `i = 0` , `nums[i] = -4`
  - `i > 0` is false, so no skip.
  - `left = 1` , `right = 5` , `target_sum = 4` .
  - Two-pointer loop:
    - `left=1, right=5` . `nums[1] + nums[5]` (`-1 + 2`) = `1` . `1 < 4` . `left++` (now `left=2`).
    - `left=2, right=5` . `nums[2] + nums[5]` (`-1 + 2`) = `1` . `1 < 4` . `left++` (now `left=3`).
    - `left=3, right=5` . `nums[3] + nums[5]` (`0 + 2`) = `2` . `2 < 4` . `left++` (now `left=4`).
    - `left=4, right=5` . `nums[4] + nums[5]` (`1 + 2`) = `3` . `3 < 4` . `left++` (now `left=5`).
    - `left=5, right=5` . `left < right` is false. Two-pointer loop ends.
- `i = 1` , `nums[i] = -1`
  - `i > 0` is true. `nums[1]` (`-1`) is not equal to `nums[0]` (`-4`). No skip.

- `left = 2`, `right = 5`, `target_sum = 1`.
- **Two-pointer loop:**
  - `left=2, right=5`. `nums[2] + nums[5]` (`-1 + 2`) = `1`. `1 == 1`! Found triplet: `[-1, -1, 2]`. `result = [ [-1, -1, 2] ]`.
  - Skip duplicates `left`: `nums[left]` is `-1`. `nums[left+1]` is `0`. Not equal.
  - Skip duplicates `right`: `nums[right]` is `2`. `nums[right-1]` is `1`. Not equal.
  - `left++` (now `left=3`), `right--` (now `right=4`).
  - `left=3, right=4`. `nums[3] + nums[4]` (`0 + 1`) = `1`. `1 == 1`! Found triplet: `[-1, 0, 1]`. `result = [ [-1, -1, 2], [-1, 0, 1] ]`.
  - Skip duplicates `left`: `nums[left]` is `0`. `nums[left+1]` is `1`. Not equal.
  - Skip duplicates `right`: `nums[right]` is `1`. `nums[right-1]` is `0`. Not equal.
  - `left++` (now `left=4`), `right--` (now `right=3`).
  - `left=4, right=3`. `left < right` is false. Two-pointer loop ends.

- `i = 2`, `nums[i] = -1`
  - `i > 0` is true. `nums[2]` (`-1`) *is equal to* `nums[1]` (`-1`). **Skip this `i`!** `continue` to `i=3`.
- `i = 3`, `nums[i] = 0`
  - `i > 0` is true. `nums[3]` (`0`) is not equal to `nums[2]` (`-1`). No skip.
  - `left = 4`, `right = 5`, `target_sum = 0`.
  - **Two-pointer loop:**
    - `left=4, right=5`. `nums[4] + nums[5]` (`1 + 2`) = `3`. `3 > 0`. `right--` (now `right=4`).
    - `left=4, right=4`. `left < right` is false. Two-pointer loop ends.
- `i = 4`: `i < n-2` (`4 < 4`) is false. Outer loop ends.

Final `result`: `[[-1, -1, 2], [-1, 0, 1]]`. Exactly what we wanted, and no duplicates!

Time & Space Complexity (Optimized):

- **Time Complexity:**
  - Sorting the array: $O(N \log N)$.
  - Outer loop (`i`): Runs `N` times.

    - Inner two-pointer loop (`left`, `right`): In the worst case, `left` and `right` pointers together traverse the array once for each `i`. So, this part is $O(N)$.
    - Total time: $O(N \log N) + O(N * N) = O(N^2)$.
    - This is a significant improvement over $O(N^3)$!
- **Space Complexity:**
    - We are storing the results in `std::vector<std::vector<int>> result`. In the worst case, there can be $O(N^2)$ unique triplets.
    - So, the space complexity is $O(N^2)$. (Excluding the space for sorting if it's in-place).

🗣️ **Is this clever solution clear in my head and hands?**

🎤 Step 5: Practice Talking Like an Interview Pro

Alright, superstar! You've grasped the problem and the optimized solution. Now, let's practice explaining it like a pro in an interview. I'm the interviewer, go!

You: "Hello! I'm ready to explain my solution for the 3Sum problem."

Me (Interviewer): "Great! Please walk me through your approach."

You: (Take a deep breath, and explain like this):

1. 📜 **Problem in short:** "The problem asks us to find all unique triplets (sets of three numbers) within a given array of integers that sum up to zero. The key is finding *unique* triplets, meaning the order of numbers within a triplet doesn't matter, and we shouldn't have duplicate triplets in our final output."
2. 💡 **Idea behind the solution:** "My approach leverages sorting and the two-pointer technique. The core idea is that if we fix one number `nums[i]` (let's call it 'a'), then the problem reduces to finding two other numbers ('b' and 'c') in the remaining part of the array such that `b + c = -nums[i]`. This is a classic 'Two Sum' variation. Sorting the array first is crucial because it allows us to efficiently find these 'b' and 'c' using two pointers and also simplifies handling duplicate values."
3. 🔨 **Code steps in plain English:**
    - "First, I sort the input `nums` array. This step is fundamental."
    - "Then, I iterate through the sorted array with an outer loop, using an index `i` to pick our first number `nums[i]`."

- "Inside this loop, there's an important optimization: I check if the current `nums[i]` is the same as the previous `nums[i-1]`. If it is, I `continue` to the next `i` because using a duplicate 'a' value would only generate triplets we've already found."
- "For each unique `nums[i]`, I set up two pointers: `left`, starting at `i + 1`, and `right`, starting at the end of the array (`n - 1`). My `target_sum` for `nums[left] + nums[right]` is `-nums[i]`."
- "I then enter a `while (left < right)` loop."
  - "I calculate `current_sum = nums[left] + nums[right]`."
  - "If `current_sum == target_sum`, I've found a valid triplet. I add `[nums[i], nums[left], nums[right]]` to my result vector."
    - "After finding a triplet, I have another set of duplicate skipping steps. I increment `left` as long as `nums[left]` is equal to `nums[left + 1]` (and `left < right`) to avoid identical 'b' values. Similarly, I decrement `right` as long as `nums[right]` is equal to `nums[right - 1]` (and `left < right`) to avoid identical 'c' values. Finally, I perform a `left++` and `right--` to move past the current numbers."
  - "If `current_sum < target_sum`, it means we need a larger sum, so I increment `left` to get a larger 'b' value."
  - "If `current_sum > target_sum`, it means we need a smaller sum, so I decrement `right` to get a smaller 'c' value."
- "The `left` and `right` pointers continue to move until they cross each other, indicating all possible pairs for the current `nums[i]` have been checked."
- "Finally, I return the `result` vector containing all unique triplets."

4. ⏱️ **Time/Space complexity:**
   - "The time complexity is $O(N^2)$. The initial sort takes $O(N \log N)$, and the nested loops effectively give us $O(N^2)$ because the two-pointer scan inside the outer loop is $O(N)$. So, $O(N \log N) + O(N^2)$ simplifies to $O(N^2)$ as N^2 dominates."
   - "The space complexity is $O(N^2)$ in the worst case, as the number of unique triplets can be proportional to N^2 (e.g., if many combinations sum to zero). This doesn't count the in-place sorting."
5. ❗ **Edge cases I handled:**
   - "I handle the case where the input array has fewer than three elements by returning an empty list, as it's impossible to form a triplet."

- "The sorting and the careful duplicate skipping logic for `i`, `left`, and `right` pointers effectively handle scenarios with many duplicate numbers, ensuring only unique triplets are added to the result."

6. 🎯 **Why this solution is solid:**
   - "This solution is optimal because it reduces the problem from a brute-force $O(N^3)$ to $O(N^2)$, which is generally the best we can do for finding triplets in a non-hashing approach. The sorting allows for efficient traversal and duplicate handling, making it both correct and efficient."

🗣️ **Can I explain this like a pro to someone else?** Practice it out loud a few times! You'll nail it.

🧠 Step 6: What Did I Learn?

Fantastic job, detective! We've solved the 3Sum mystery. Let's wrap up our learning:

✅ **What's the main technique or idea we used?**
The main technique was combining **sorting** with the **two-pointer approach**. Sorting allows us to systematically search for complementary pairs and efficiently skip duplicates. The two-pointer method is brilliant for finding pairs with a specific sum in a sorted array.

✅ **What "category" is this problem?**
This problem falls into the category of **Array Manipulation and Searching**, specifically problems involving finding combinations or subsets that meet a certain criteria (a sum, in this case). It's a classic variation of the "Two Sum" problem.

✅ **When might I see a similar problem again?**
You'll see similar problems whenever you need to find:

- Pairs or triplets (or k-tuples) that sum to a target.
- Problems where you need to efficiently check combinations in a sorted array.
- Problems that require dealing with duplicates and finding unique sets.

Think of "4Sum" (finding four numbers that sum to zero), "Two Sum II - Input array is sorted", or problems where you need to find pairs that are closest to a target sum.

✅ **Can I spot this trick again next time?**
Absolutely! Look for these clues:

- "Find combinations/subsets that sum to X."

- "Input array contains positive, negative, and zero numbers."
- "Need to return unique sets."
- If sorting the array seems beneficial for some reason (like reducing search space or handling order).

If you see these, immediately think: "Can I sort this and use two pointers (or a variation of it)?"

🔁 **Optional Bonus Practice:**

Here's a mini-practice challenge you can try right now to apply what you just learned:

Mini-Challenge: "Two Sum II - Input array is sorted"

> Given a **1-indexed** array of integers `numbers` that is already **sorted in non-decreasing order**, find two numbers such that they add up to a specific `target` number. Return the *indices* of the two numbers `index1` and `index2` as an integer array `[index1, index2]` where `1 <= index1 < index2 <= numbers.length`.
>
> You are guaranteed that there is exactly one solution.
>
> **Example:**
> `numbers = [2, 7, 11, 15]`, `target = 9`
> **Output:** `[1, 2]` (because `2 + 7 = 9`, and their 1-indexed positions are 1 and 2).

Think about how our two-pointer approach from 3Sum can be simplified here since the array is already sorted and we only need two numbers! You won't need the outer `i` loop, j...

Please give me the complete dry run of brute-force method.

You got it, detective! Let's rewind and do a super detailed, step-by-step dry run of the brute-force method. We'll use the example `nums = [-1, 0, 1, 2, -1, -4]` and trace *every single step* of the code. This will really help you see why it's $O(N^3)$ and how the `set` helps with uniqueness.

Our `nums` array: `[-1, 0, 1, 2, -1, -4]`
`n = nums.size()` which is `6`.

Our goal: Find `i, j, k` such that `nums[i] + nums[j] + nums[k] == 0`, where `i < j < k`. Store unique sorted triplets.

Let's track the variables:

- `i` (outer loop index)
- `j` (middle loop index)
- `k` (inner loop index)
- `nums[i]`
- `nums[j]`
- `nums[k]`
- `sum = nums[i] + nums[j] + nums[k]`
- `currentTriplet` (before and after sorting)
- `uniqueTriplets` (our `std::set`)
- `result` (final `std::vector<std::vector<int>>`)

C++

```cpp
#include <vector>
#include <algorithm>
#include <set>

class Solution {
public:
    std::vector<std::vector<int>> threeSum(std::vector<int>& nums) {
        std::set<std::vector<int>> uniqueTriplets; // Initially empty

        int n = nums.size(); // n = 6

        // Edge case: n < 3
        if (n < 3) { // 6 < 3 is false, so we continue.
            return {};
        }

        // Outer loop: for (int i = 0; i < n - 2; ++i)
        // This loop runs for i = 0, 1, 2, 3 (since 3 < 6 - 2 = 4)
        for (int i = 0; i < n - 2; ++i) { // n-2 = 4
            // Middle loop: for (int j = i + 1; j < n - 1; ++j)
            // Inner loop: for (int k = j + 1; k < n; ++k)
            // ... (rest of the code)
        }
        // ... (convert set to vector)
```

```
    }
};
```

DRY RUN START

Initial `nums`: `[-1, 0, 1, 2, -1, -4]`
`n = 6`
`uniqueTriplets = {}` (empty set)
`result` (will be empty until conversion at the end)

1. `i = 0` (`i < n-2` is `0 < 4`, True)
* `nums[i]` is `nums[0]` which is `-1`.

```
**1.1. `j = i + 1` (so `j = 1`). (`j < n-1` is `1 < 5`, True)**
    * `nums[j]` is `nums[1]` which is `0`.

    **1.1.1. `k = j + 1` (so `k = 2`). (`k < n` is `2 < 6`, True)**
        * `nums[k]` is `nums[2]` which is `1`.
        * `sum = nums[i] + nums[j] + nums[k]` = `(-1) + 0 + 1 = 0`.
        * `if (sum == 0)` is True.
        * `currentTriplet = {-1, 0, 1}`.
        * `std::sort(currentTriplet.begin(), currentTriplet.end())` -> `curren
        * `uniqueTriplets.insert(currentTriplet)` -> `uniqueTriplets = {{ -1,

    **1.1.2. `k = 3`. (`k < n` is `3 < 6`, True)**
        * `nums[k]` is `nums[3]` which is `2`.
        * `sum = nums[i] + nums[j] + nums[k]` = `(-1) + 0 + 2 = 1`.
        * `if (sum == 0)` is False.

    **1.1.3. `k = 4`. (`k < n` is `4 < 6`, True)**
        * `nums[k]` is `nums[4]` which is `-1`.
        * `sum = nums[i] + nums[j] + nums[k]` = `(-1) + 0 + (-1) = -2`.
        * `if (sum == 0)` is False.

    **1.1.4. `k = 5`. (`k < n` is `5 < 6`, True)**
        * `nums[k]` is `nums[5]` which is `-4`.
        * `sum = nums[i] + nums[j] + nums[k]` = `(-1) + 0 + (-4) = -5`.
        * `if (sum == 0)` is False.

    * `k` loop ends (`k=6`, `6 < 6` is False).

**1.2. `j = 2`. (`j < n-1` is `2 < 5`, True)**
    * `nums[j]` is `nums[2]` which is `1`.

    **1.2.1. `k = j + 1` (so `k = 3`). (`k < n` is `3 < 6`, True)**
```

```
        * `nums[k]` is `nums[3]` which is `2`.
        * `sum = nums[i] + nums[j] + nums[k]` = `(-1) + 1 + 2 = 2`.
        * `if (sum == 0)` is False.

    **1.2.2. `k = 4`. (`k < n` is `4 < 6`, True)**
        * `nums[k]` is `nums[4]` which is `-1`.
        * `sum = nums[i] + nums[j] + nums[k]` = `(-1) + 1 + (-1) = -1`.
        * `if (sum == 0)` is False.

    **1.2.3. `k = 5`. (`k < n` is `5 < 6`, True)**
        * `nums[k]` is `nums[5]` which is `-4`.
        * `sum = nums[i] + nums[j] + nums[k]` = `(-1) + 1 + (-4) = -4`.
        * `if (sum == 0)` is False.

    * `k` loop ends.

**1.3. `j = 3`. (`j < n-1` is `3 < 5`, True)**
    * `nums[j]` is `nums[3]` which is `2`.

    **1.3.1. `k = j + 1` (so `k = 4`). (`k < n` is `4 < 6`, True)**
        * `nums[k]` is `nums[4]` which is `-1`.
        * `sum = nums[i] + nums[j] + nums[k]` = `(-1) + 2 + (-1) = 0`.
        * `if (sum == 0)` is True.
        * `currentTriplet = {-1, 2, -1}`.
        * `std::sort(currentTriplet.begin(), currentTriplet.end())` -> `curren
        * `uniqueTriplets.insert(currentTriplet)` -> `uniqueTriplets = {{ -1,

    **1.3.2. `k = 5`. (`k < n` is `5 < 6`, True)**
        * `nums[k]` is `nums[5]` which is `-4`.
        * `sum = nums[i] + nums[j] + nums[k]` = `(-1) + 2 + (-4) = -3`.
        * `if (sum == 0)` is False.

    * `k` loop ends.

**1.4. `j = 4`. (`j < n-1` is `4 < 5`, True)**
    * `nums[j]` is `nums[4]` which is `-1`.

    **1.4.1. `k = j + 1` (so `k = 5`). (`k < n` is `5 < 6`, True)**
        * `nums[k]` is `nums[5]` which is `-4`.
        * `sum = nums[i] + nums[j] + nums[k]` = `(-1) + (-1) + (-4) = -6`.
        * `if (sum == 0)` is False.

    * `k` loop ends.

* `j` loop ends (`j=5`, `5 < 5` is False).
```

2. `i = 1` (`i < n-2` is `1 < 4`, True)
* `nums[i]` is `nums[1]` which is `0`.

```
**2.1. `j = i + 1` (so `j = 2`). (`j < n-1` is `2 < 5`, True)**
    * `nums[j]` is `nums[2]` which is `1`.

    **2.1.1. `k = j + 1` (so `k = 3`). (`k < n` is `3 < 6`, True)**
        * `nums[k]` is `nums[3]` which is `2`.
        * `sum = nums[i] + nums[j] + nums[k]` = `0 + 1 + 2 = 3`.
        * `if (sum == 0)` is False.

    **2.1.2. `k = 4`. (`k < n` is `4 < 6`, True)**
        * `nums[k]` is `nums[4]` which is `-1`.
        * `sum = nums[i] + nums[j] + nums[k]` = `0 + 1 + (-1) = 0`.
        * `if (sum == 0)` is True.
        * `currentTriplet = {0, 1, -1}`.
        * `std::sort(currentTriplet.begin(), currentTriplet.end())` -> `curren
        * `uniqueTriplets.insert(currentTriplet)` -> `uniqueTriplets` is still

    **2.1.3. `k = 5`. (`k < n` is `5 < 6`, True)**
        * `nums[k]` is `nums[5]` which is `-4`.
        * `sum = nums[i] + nums[j] + nums[k]` = `0 + 1 + (-4) = -3`.
        * `if (sum == 0)` is False.

    * `k` loop ends.

**2.2. `j = 3`. (`j < n-1` is `3 < 5`, True)**
    * `nums[j]` is `nums[3]` which is `2`.

    **2.2.1. `k = j + 1` (so `k = 4`). (`k < n` is `4 < 6`, True)**
        * `nums[k]` is `nums[4]` which is `-1`.
        * `sum = nums[i] + nums[j] + nums[k]` = `0 + 2 + (-1) = 1`.
        * `if (sum == 0)` is False.

    **2.2.2. `k = 5`. (`k < n` is `5 < 6`, True)**
        * `nums[k]` is `nums[5]` which is `-4`.
        * `sum = nums[i] + nums[j] + nums[k]` = `0 + 2 + (-4) = -2`.
        * `if (sum == 0)` is False.

    * `k` loop ends.

**2.3. `j = 4`. (`j < n-1` is `4 < 5`, True)**
    * `nums[j]` is `nums[4]` which is `-1`.

    **2.3.1. `k = j + 1` (so `k = 5`). (`k < n` is `5 < 6`, True)**
        * `nums[k]` is `nums[5]` which is `-4`.
        * `sum = nums[i] + nums[j] + nums[k]` = `0 + (-1) + (-4) = -5`.
```

```
        * `if (sum == 0)` is False.

    * `k` loop ends.

* `j` loop ends.
```

3. `i = 2` (`i < n-2` is `2 < 4`, True)
* `nums[i]` is `nums[2]` which is `1`.

```
**3.1. `j = i + 1` (so `j = 3`). (`j < n-1` is `3 < 5`, True)**
    * `nums[j]` is `nums[3]` which is `2`.

    **3.1.1. `k = j + 1` (so `k = 4`). (`k < n` is `4 < 6`, True)**
        * `nums[k]` is `nums[4]` which is `-1`.
        * `sum = nums[i] + nums[j] + nums[k]` = `1 + 2 + (-1) = 2`.
        * `if (sum == 0)` is False.

    **3.1.2. `k = 5`. (`k < n` is `5 < 6`, True)**
        * `nums[k]` is `nums[5]` which is `-4`.
        * `sum = nums[i] + nums[j] + nums[k]` = `1 + 2 + (-4) = -1`.
        * `if (sum == 0)` is False.

    * `k` loop ends.

**3.2. `j = 4`. (`j < n-1` is `4 < 5`, True)**
    * `nums[j]` is `nums[4]` which is `-1`.

    **3.2.1. `k = j + 1` (so `k = 5`). (`k < n` is `5 < 6`, True)**
        * `nums[k]` is `nums[5]` which is `-4`.
        * `sum = nums[i] + nums[j] + nums[k]` = `1 + (-1) + (-4) = -4`.
        * `if (sum == 0)` is False.

    * `k` loop ends.

* `j` loop ends.
```

4. `i = 3` (`i < n-2` is `3 < 4`, True)
* `nums[i]` is `nums[3]` which is `2`.

```
**4.1. `j = i + 1` (so `j = 4`). (`j < n-1` is `4 < 5`, True)**
    * `nums[j]` is `nums[4]` which is `-1`.

    **4.1.1. `k = j + 1` (so `k = 5`). (`k < n` is `5 < 6`, True)**
        * `nums[k]` is `nums[5]` which is `-4`.
        * `sum = nums[i] + nums[j] + nums[k]` = `2 + (-1) + (-4) = -3`.
```

```
        * `if (sum == 0)` is False.

    * `k` loop ends.

* `j` loop ends.
```

- `i` loop ends (`i=4`, `4 < 4` is False).

End of all loops.

Convert `uniqueTriplets` **to** `result`**:**
`result = std::vector<std::vector<int>>(uniqueTriplets.begin(), uniqueTriplets.end());`
The `std::set` stores elements in sorted order. So, `uniqueTriplets` has `{{-1, -1, 2}, {-1, 0, 1}}`.
`result` will now be `[[-1, -1, 2], [-1, 0, 1]]`.

Return `result`**.**

Summary of what we saw:

- We tried *every single combination* of three distinct indices `(i, j, k)`.
- Even though we found `{-1, 0, 1}` and `{-1, 2, -1}` (which, after sorting, are `{-1, 0, 1}` and `{-1, -1, 2}` respectively), the `std::set` automatically ensured that `{-1, 0, 1}` was only stored once, and that `{-1, -1, 2}` was also stored.
- The `std::sort` on `currentTriplet` before inserting into the `set` is vital for uniqueness, otherwise `{-1, 0, 1}` and `{0, -1, 1}` would be considered different without sorting.

This dry run really highlights the brute-force nature: *no smart skipping*, just check everything! This is why it's $O(N^3)$, because the number of times the innermost check `if (nums[i] + nums[j] + nums[k] == 0)` runs is proportional to N^3.

Does this complete dry run make the brute-force method perfectly clear, especially how `i`, `j`, and `k` move and how the `set` handles uniqueness?