# Understanding the Two Pointers Technique - Part 1

Aug 17, 2024

*The problems in this post are taken from the "Two Pointers" section of NeetCode's roadmap. I think his problem selection is good, but his video explanations are subpar. I hope this post serves as a better explanation of the main ideas and common themes of each problem.*

**Prerequisites:** You should be able to code $O(n^2)$ algorithms for the first two problems.

**Problems to try:**

- https://leetcode.com/problems/two-sum-ii-input-array-is-sorted/
- https://leetcode.com/problems/container-with-most-water/
- https://leetcode.com/problems/trapping-rain-water/

At this point, even if you can't find the best solution, I encourage you to come up with a slow, **quadratic runtime** algorithm for the first two problems. We'll discover the linear time algorithms as we go.

## 167. Two Sum II - Input Array Is Sorted

### Summary of problem statement:

We're given two things:

- a sorted array of integers $a_1$, $a_2$, $\ldots$, $a_n$, where $n$ is the length of the array.
- a target value $t$.

The goal is to find a pair of indices $(i, j)$, where $1 \leq i < j \leq n$, such that $a_i + a_j = t$.

### First attempt (Time Limit Exceeded):

Try all possible pairs of indices $(i, j)$ to see if they satisfy $a_i + a_j = t$. The number of pairs to check is bounded by $n^2$[1], so our algorithm runs in $O(n^2)$ time.

```
vector<int> twoSum(vector<int>& a, int t) {
        const int n = a.size();

        for (int i = 0; i < n; i++) {
                for (int j = i+1; j < n; j++) {
// The code is zero-indexed, but the idea is the same.
// This inner loop checks every pair (i, j) such that 0 <= i < j < n.
                        if (a[i] + a[j] == t) {
                                return {i+1, j+1};
                        }
                }
        }
// The problem statement guarantees we won't get here.
        throw runtime_error("No solution found.");
}
```

### Making Observations:

Since there are about $n^2$ pairs to check, any algorithm that checks all pairs will be too slow.[2] **We'll have to figure out how to rule out some pairs without checking them.**

Consider the array $a = [1, 2, 5, 6, 11]$ and the target $t = 11$. Let's calculate $a_i + a_j$ for all possible pairs of indices $(i, j)$[3]:

| (i, j) | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | | 3 | 6 | 7 | 12 |
| 2 | | | 7 | 8 | 13 |
| 3 | | | | 11 | 16 |
| 4 | | | | | 17 |
| 5 | | | | | |

In this case, $a_3 + a_4 = t$, so the answer is $(3, 4)$. But, what's more important is the key observation[4]:

ⓘ **Claim**

Starting from any cell, moving right or down can only increase the sum. Also, moving left or up can only decrease the sum.

This is always true, because the array is sorted in non-decreasing order.[5]

## Using the claim

Let's try $a = [1, 2, 5, 6, 11]$ and $t = 11$ again, this time trying to minimize the number of pairs we have to check. I recommend writing down your own table and following along:

Initially, we haven't checked any pairs:

| (i, j) | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 |  | **?** | **?** | **?** | **?** |
| 2 |  |  | **?** | **?** | **?** |
| 3 |  |  |  | **?** | **?** |
| 4 |  |  |  |  | **?** |
| 5 |  |  |  |  |  |

Let's check one pair, $(i, j) = (1, 5)$. We get $a_i + a_j = 12$. What does this tell us?

| (i, j) | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 |  | **?** | **?** | **?** | 12 |
| 2 |  |  | **?** | **?** | **?** |
| 3 |  |  |  | **?** | **?** |
| 4 |  |  |  |  | **?** |
| 5 |  |  |  |  |  |

**Answer:** By the claim, we know that the sums for $(2, 5)$, $(3, 5)$, and $(4, 5)$ are at least $12$. But $t = 11$, so these sums can't be equal to $t$ – they're too big![6]

| **(i, j)** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| 1 | | **?** | **?** | **?** | 12 |
| 2 | | | **?** | **?** | ≥12 |
| 3 | | | | **?** | ≥12 |
| 4 | | | | | ≥12 |
| 5 | | | | | |

Now let's check $(1, 4)$[7]. We get $a_1 + a_4 = 7$. This time, our sum is smaller than $t$, which means $(1, 2)$ and $(1, 3)$ have sums smaller than $t$.

| **(i, j)** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| 1 | | ≤7 | ≤7 | 7 | 12 |
| 2 | | | **?** | **?** | ≥12 |
| 3 | | | | **?** | ≥12 |
| 4 | | | | | ≥12 |
| 5 | | | | | |

Next, we check $(2, 4)$. We get $a_2 + a_4 = 8$, which is also too small. If you've been trying this yourself, you should just have $(3, 4)$ left, which turns out to have a sum of exactly $t$.

| **(i, j)** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| 1 | | ≤7 | ≤7 | 7 | 12 |
| 2 | | | ≤8 | 8 | ≥12 |
| 3 | | | | **11** | ≥12 |
| 4 | | | | | ≥12 |
| 5 | | | | | |

We only checked 4 pairs! What happened?

- Initially, any pair $(i, j)$ where $i < j$ and $i, j \in [1, 5]$ could be the answer.
- After checking $(1, 5)$, we narrowed the range down to $i, j \in [1, 4]$.
- After checking $(1, 4)$, we narrowed the range down to $i, j \in [2, 4]$.
- After checking $(2, 4)$, we narrowed the range down to $i, j \in [3, 4]$.
- After checking $(3, 4)$, we found a sum of exactly $t$.

Let's generalize this:

**Algorithm for finding a pair $(i, j)$ with $a_i + a_j = t$**

Initially, let $l = 1$ and $r = n$. The problem statement guarantees that there's a pair $(i, j)$ where $i$ and $j$ are in the range $[l, r]$. There are three cases:

1. If $a_l + a_r = t$, then $(l, r)$ is the desired pair, and we can stop.
2. If $a_l + a_r < t$, then the pair must be in the range $[l + 1, r]$.
3. If $a_l + a_r > t$, then the pair must be in the range $[l, r - 1]$.

If we ended up in case 2 or 3, update $[l, r]$ to the new range and try again.

The number of values in the range $[l, r]$ is initially $n$, and this decreases by $1$ each time we check a pair. That means there are $O(n)$ intervals to check. Our claim also guarantees we will find the correct answer.

### Code (Accepted):

```
vector<int> twoSum(vector<int>& a, int t) {
        const int n = a.size();
        for (int l = 0, r = n-1; l < r; ) {
                if (a[l] + a[r] == t) return {l+1, r+1};
                if (a[l] + a[r] < t) l++;
                else r--;
        }
        throw runtime_error("No pair found.");
}
```

Notice that I didn't mention the words "two pointers" in my solution – they arose naturally when considering the possible range for $(i, j)$. The crux of the problem is the key claim we made and how we used it to avoid checking too many tuples, not the name of the algorithm we used.

**Exercise:** Suppose that using the same index twice is now allowed. How can we change our program to handle this?

**Exercise:** Solve 3Sum. It has a few extra observations and trickier implementation, which is good practice.

# Footnotes

[1] Make sure you understand why this is true. In general, you should aim to **understand every sentence in this post**. If something doesn't make sense, feel free to ask me on Discord. ↩

[2] This is not actually true, since LeetCode's constraints say $n \leq 3 \cdot 10^4$ – there are tricks to squeeze this within the time limit. Serious competitive programming sites will set higher bounds (e.g., $n \le 2 \cdot 10^5) to prevent this. ↩

[3] Wait, didn't I just say we wanted to avoid checking some pairs? Yes, but that's only the goal of the final solution. One of the simplest problem-solving strategies is to try basic things and look for patterns – we check all values, because it helps us get as much information as we can. If it's too tedious to work out by hand, you can also write a program that outputs this table for you. ↩

[4] The wording is slightly awkward because it's possible for two adjacent elements to be equal. Once again, make sure you understand why the claim is true. ↩

[5] When solving problems, it's best to prove the claims you make. In this case, "proving" just means fully convincing yourself. If that's not possible, another good option is to stress test your solution with a bunch of examples (preferably auto-generated ones). ↩

[6] If at any point you think you've figured out how to solve the problem, do that! You can skim through my solution process once you're done or if you get stuck. ↩

[7] How did I know to check $(1, 4)$ first? Good question. The answer is a mix of trial and error and experience. My suggestion is to try other pairs and notice that they aren't as "nice" to work with. ↩