

Brute Force to Optimal

can you refine and write properly

Now, Striver mentions that even if you the optimal solution to hard problem, you shouldn't directly answer it to the interviewer instead you must build on that question, ask the interviewer about test cases and all the other required stuff and must understand the question thoroughly and then try to show the brute force approach first (a brute force approach is nothing but the first solution that comes to your mind and then you have to just optimise it further) followed by better solution and then optimise the better solution further to get the most optimal solution. (If you wanna add one layer after better solution like more better than then you can do that too)(The key is to show that you can think logically and critically and your approach to a problem is efficient and genuine (Even if you have to fake it the fake it))

Here's a more refined and professionally written version of your paragraph:

Striver emphasizes that even if you already know the optimal solution to a hard problem, you should **not** directly jump to it during an interview. Instead, take a structured approach. Begin by thoroughly understanding the problem—clarify constraints, edge cases, and ask relevant test cases from the interviewer.

Start with the **brute force approach**—this is usually the first solution that comes to mind, even if it's inefficient. Then, gradually build upon it by proposing a **better solution**, and finally work your way toward the **most optimal solution**. If needed, you can add an intermediate step—something "better than better"—to further demonstrate your thought process.

The goal is to showcase your **logical thinking, problem-solving skills**, and a **genuine, step-by-step approach**. Even if you already know the answer, simulate the thought process as if you're discovering it—**fake it if you have to**—because interviewers value how you arrive at a solution, not just the final answer.

Let me know if you'd like a more casual or technical version too.

All mention that do it for harder problems but not for the easier ones

Here's the revised version with that point clearly included:

Striver emphasizes that if you already know the optimal solution to a **hard problem**, you **should not** jump straight to it in an interview. Instead, approach it systematically. Begin by thoroughly understanding the problem—ask the interviewer about constraints, edge cases, and relevant test cases.

Start by presenting the **brute force solution**—the first idea that comes to mind, even if it's inefficient. Then, gradually build on it by suggesting a **better approach**, and finally work your way toward the **most optimal solution**. You can even add an intermediate step—something “better than better”—if it helps demonstrate your thinking more clearly.

However, this layered approach is generally recommended **only for medium to hard problems**. For **easier problems**, it's better to get straight to the point with the optimal solution, unless specifically asked otherwise.

The key is to showcase your **logical reasoning**, **structured thinking**, and ability to improve solutions step by step. Even if you already know the answer, simulate the thought process—**fake it if you have to**—because interviewers care more about how you think than just the final output.

Let me know if you want a shorter version too.

do we always use sort() for sorting in dsa with cpp?

Great question, Vaibhav! In C++ DSA (Data Structures and Algorithms), you **don't always have to use** `sort()`, but it's **most commonly used** because it's efficient and easy to use.

✓ **When to use** `sort()`:

- When you want to **sort an array or vector** in ascending or descending order.

- When you're solving problems that involve **searching, binary search, greedy approaches, or comparisons**.
- It's a **stable and optimized** function in STL with a time complexity of $O(N \log N)$.

⚙ Syntax:

```
cpp

#include <algorithm>
sort(arr, arr + n); // For arrays
sort(v.begin(), v.end()); // For vectors
```

🌟 Custom Sorting:

You can also pass a custom comparator:

```
cpp

sort(v.begin(), v.end(), greater<int>()); // Descending order
```

Or use your own function:

```
cpp

bool cmp(pair<int,int> a, pair<int,int> b) {
    return a.second < b.second; // sort by second element
}

sort(v.begin(), v.end(), cmp);
```

❌ When `sort()` is not used:

- If data is already sorted or partially sorted.
- In problems where sorting isn't required at all (e.g., counting, hash maps, stacks, etc.).
- If you use **priority queues, multisets, or maps**, they maintain order automatically.
- For **custom data structures**, you might need to write your own sorting logic or use other STL containers.

Summary:

No, you don't **always** use `sort()`, but in many DSA problems where ordering helps, `sort()` is your go-to tool due to its simplicity and efficiency.

Let me know if you want to see some unique sorting-based problems or tricks around `sort()`.

I mean that we don't have to write any sorting algorithms explicitly by ourselves right?

Ahh got it now, Vaibhav! Yes, you're absolutely right — in most DSA problems **you don't have to write sorting algorithms like bubble sort, merge sort, or quicksort by yourself.**

✓ Why?

- In contests, interviews, and CP (competitive programming), you're expected to **use built-in functions** like `sort()` from the C++ STL unless explicitly told otherwise.
- Writing sorting algorithms manually is **only expected**:
 - In **academics or teaching** environments to understand how they work.
 - If a question specifically asks: *"Implement merge sort"* or *"Without using built-in sort..."*.

🚀 What do you use instead?

- ✓ `sort()` — for most use-cases (uses IntroSort internally → hybrid of quicksort, heapsort, and insertion sort).
- ✓ `stable_sort()` — when relative ordering of equal elements must be preserved.
- ✓ Data structures like `set`, `map`, or `priority_queue` — they maintain order automatically.

✗ So when do you write your own sorting?

- Only when you're:
 - Learning sorting algorithms.
 - Practicing algorithm implementation.

- Solving a question that **disallows built-in sort** (rare in interviews).



Bottom Line:

In DSA using C++, you almost **never need to write your own sorting algorithm** unless the question demands it. Use STL's `sort()` — it's fast, optimized, and interview-approved. ✓

Let me know if you want a cheat sheet on custom comparators too!

so why did striver teach me sorting algos if we were to use `sort()` function only?

That's a very smart and valid question, Vaibhav — and you're thinking in exactly the right direction.

Here's why Striver (and other top educators) still teach sorting algorithms even though we usually just use `sort()`:



1. To Build Core Understanding

You need to **understand how sorting works internally** — not just use it like a black box. Knowing:

- Why merge sort is stable and efficient.
- Why quicksort has average-case $O(n \log n)$ but worst-case $O(n^2)$.
- Why bubble sort is slow.

This deepens your **algorithmic thinking** — which helps you **analyze, debug, and optimize** real problems better.



2. To Learn Time & Space Complexities

Different sorting algorithms have different complexities. For example:

- Merge Sort → $O(n \log n)$ time, $O(n)$ space.
- Quick Sort → $O(n \log n)$ time (average), $O(1)$ space (in-place).

Understanding this helps you **choose or modify algorithms** based on problem constraints.

3. Interviewers Might Ask Theory

Even if you're not expected to implement it, an interviewer **might ask you to explain**:

- How merge sort works.
- Why quicksort is better on average.
- Which sort is stable and why.

So it's not always about coding it — but understanding it.

4. Real Implementation Practice

In some rare interviews or coding rounds, they might say:

"You can't use STL functions. Sort this manually."

This is rare but can happen, especially in service-based companies or off-campus hiring tests.

Final Truth:

Striver teaches sorting algorithms so you **understand the backbone of how** `sort()` **works**, not because you'll always write them from scratch.

Just like you learn how an engine works even if you only drive the car — it makes you a better driver (or in our case, a better problem solver).

Let me know if you want a crisp summary of when to use which sorting algorithm or a quick comparison table of all sorting techniques!

what do you mean by stable sorts?

Great question, Vaibhav! The idea of **stable sorting** is simple but very important — especially in interviews and real-world applications like sorting records.

Stable Sort — Definition:

A **stable sort** maintains the **relative order** of elements that are **equal** according to the sorting key.

In Simple Words:

If two elements have the **same value**, a stable sort **keeps them in the same order** they appeared in the original list.

Example:

Let's say you're sorting people by age:

```
cpp

{ {"Amit", 21}, {"Ravi", 19}, {"Tina", 21} }
```

Now sort by age.

Stable sort output:

```
cpp

{ {"Ravi", 19}, {"Amit", 21}, {"Tina", 21} } ← Amit comes before Tina (same as original)
```

Unstable sort might give:

cpp

```
{ {"Ravi", 19}, {"Tina", 21}, {"Amit", 21} } ← Order of 21-year-olds is changed
```

✓ Examples of Stable Sorts:

- Merge Sort
- Insertion Sort
- Bubble Sort
- `stable_sort()` in C++ STL

✗ Examples of Unstable Sorts:

- Quick Sort
- Heap Sort
- C++ `sort()` function (Not guaranteed to be stable)

🔍 When does stability matter?

- When you're sorting by **multiple keys** in stages.
E.g., sort by marks, then by name. The second sort **must be stable** to preserve previous order.

🧠 Summary:


A **stable sort keeps equal elements in their original order**. It's crucial when sorting by multiple criteria or when original order matters.

Let me know if you want an example problem where stable sorting is needed!

what about sort function () is that stable or not? and what does it uses internally?

Awesome follow-up, Vaibhav!

Is `sort()` in C++ STL stable?

 No, the standard `sort()` in C++ is **not guaranteed to be stable**.

If elements have equal keys (e.g., same value when sorting), their **relative order may change**.

But C++ has a stable version too:

- Use `stable_sort()` if you need stability.
-

What does `sort()` use internally?

C++'s `sort()` (from `<algorithm>`) uses a highly efficient hybrid algorithm called:

Introsort = Introspective Sort

It combines:

1. **Quicksort** → for fast average-case performance ($O(n \log n)$).
 2. **Heapsort** → for worst-case guarantee ($O(n \log n)$ even in bad cases).
 3. **Insertion sort** → for very small subarrays (to reduce overhead).
-

Why not stable?

- Quicksort and Heapsort (parts of Introsort) are **not stable** by default.
 - C++ focuses on **performance**, so `sort()` trades off stability for speed.
-

When should you use `stable_sort()`?

Use it when:

- You're sorting by **multiple criteria**.
- You want to **preserve original order** of equal elements.

Example:

cpp

```
vector<pair<int, string>> people = {{21, "Amit"}, {19, "Ravi"}, {21, "Tina"}};  
stable_sort(people.begin(), people.end());
```

This keeps "Amit" before "Tina" since both have age 21.

⚡ Summary:

Function	Stable?	Time Complexity	Internally Uses
<code>sort()</code>	❌ No	$O(n \log n)$ average	Introsort (Quick + Heap + Insertion)
<code>stable_sort()</code>	✅ Yes	$O(n \log^2 n)$ worst	Merge Sort

Let me know if you want a mini project or problem that shows where stability really matters!