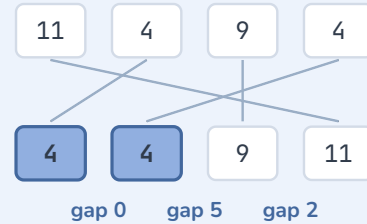


Sorting

Chapter 1.5 · Overall level: Foundation



Here are four sensor readings: 11, 4, 9, 4. Are any two of them equal, and which two are closest? In this order you have to compare every pair. Put them in order and the answers are sitting next to each other.

Sorting is rarely the goal of a problem. It is a way of arranging evidence so that the question becomes easy. This chapter shows how to ask your language to sort, how to sort records by more than one key, why sorting costs $O(n \log n)$, and three ideas that only work after sorting.

1.5.1 Order exposes duplicates and gaps

Take the readings $a = [11, 4, 9, 4]$. To decide whether two of them are equal, the unsorted list gives you no shortcut: any reading might match any other, so you check all $n(n-1)/2$ pairs. With 200,000 readings that is about 2×10^{10} comparisons.

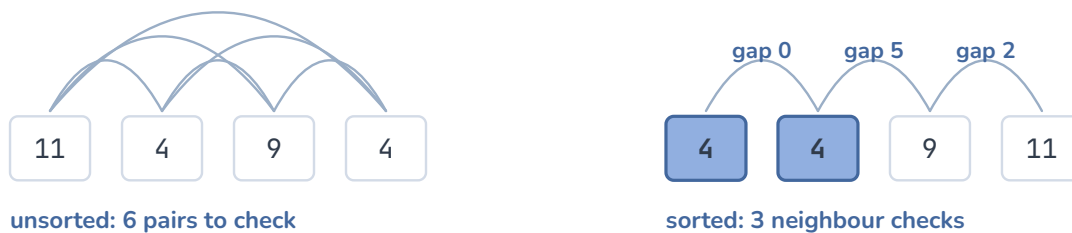


Figure 1.5.1 — Unsorted, any reading may match any other. Sorted, equal or nearby values can only be neighbours.

Sorted, the list is $[4, 4, 9, 11]$. Equal values must now stand side by side, because nothing can fit between two equal values. So one pass that compares each value with its right-hand neighbour finds every duplicate. The same pass reads off the *gaps* between neighbours: $4 - 4 = 0$, $9 - 4 = 5$ and $11 - 9 = 2$.

That pattern, **sort, then look only at neighbours**, is the most common use of sorting in programming problems. Section 1.5.6 proves why neighbours are enough for the closest pair. First, how does a list get sorted at all?

1.5.2 Sorting by hand: insertion

Insertion sort is how most people sort a hand of cards. Keep a sorted part on the left. Take the next value, and slide it left past every larger value until it meets one that is not larger. Each value that steps aside to make room is a *shift*.

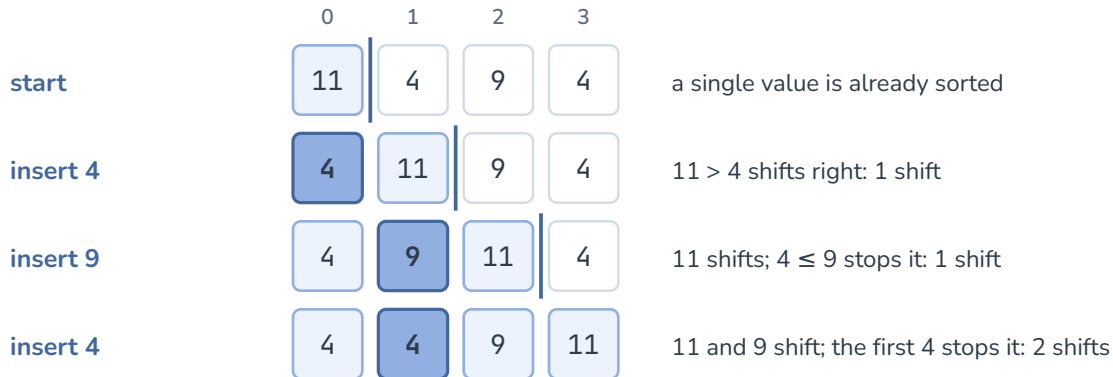


Figure 1.5.2 — Insertion sort on the shared example. The pale cells are the sorted part, the dark cell is the value just inserted and the bar separates sorted from unsorted. The last 4 stops behind the first 4, so equal values keep their input order.

Why it works: before each step the left part is sorted, and inserting one value at the place where everything to its left is no larger and everything to its right is larger keeps it sorted, one value longer. After $n - 1$ insertions the whole list is the sorted part.

The cost is the number of shifts. Here there were $1 + 1 + 2 = 4$, one for each pair that started in the wrong order: (11, 4), (11, 9), (11, 4) and (9, 4). A list in reverse order has every pair wrong, $n(n - 1)/2$ of them, so insertion sort is $O(n^2)$ in the worst case. That is fine for 20 values and hopeless for 200,000. You will not write it in practice; it is the baseline that library sorts beat.

1.5.3 Asking the library to sort

Every language you use ships with a sort that takes $O(n \log n)$ time and is carefully tested. Use it.

Language	Sort a list of numbers ascending	Stable?
Python	<code>a.sort()</code> or <code>b = sorted(a)</code>	yes
C++	<code>sort(a.begin(), a.end())</code>	no; <code>stable_sort</code> is
Java	<code>Arrays.sort(a)</code> (array) or <code>Collections.sort(list)</code>	yes for objects
JavaScript	<code>a.sort((x, y) => x - y)</code>	yes

JavaScript needs the comparator. Without it, `sort()` turns the values into strings and compares them letter by letter, so `[11, 4, 9, 4].sort()` gives `[11, 4, 4, 9]`: the string "11" comes before "4" because the character 1 comes before 4. The comparator `(x, y) => x - y` returns a negative number when x should come first, zero for a tie and a positive number otherwise.

1.5.4 How library sorts reach $O(n \log n)$: merging

Library sorts are based on ideas such as merge sort, quicksort and heapsort. Merge sort is the easiest to see. Its key step merges two lists that are already sorted. Only the two *front* values can be the smallest remaining value, so compare them, output the smaller one and move that front forward.

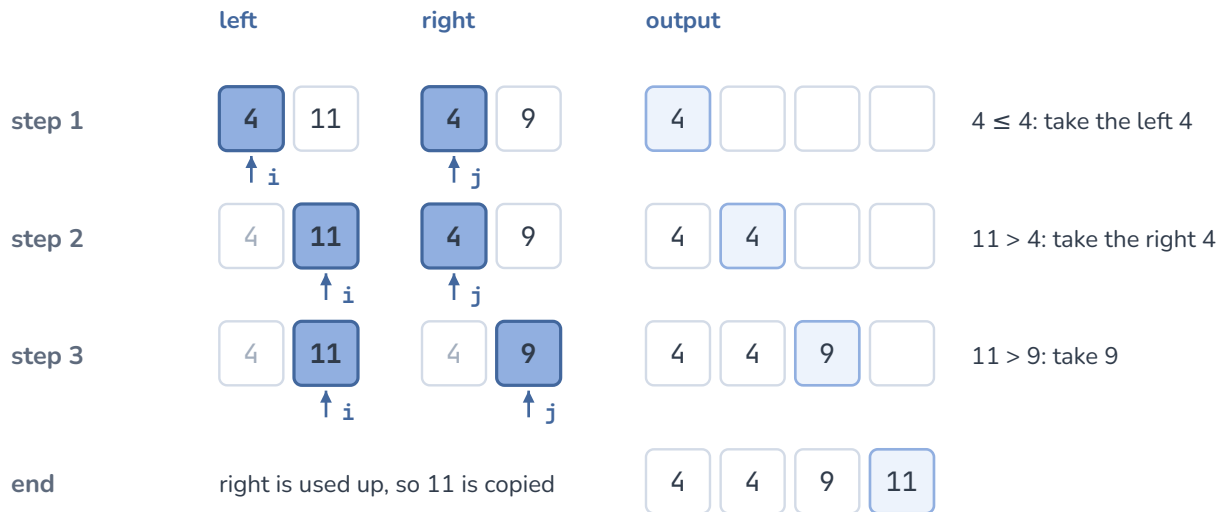


Figure 1.5.3 — The next output is the smaller of the two unconsumed front values. Dark cells are the two values being compared; grey cells are used up. On a tie the left value goes first, which keeps equal values in their input order.

Each comparison outputs one value, so merging two lists with n values in total takes at most $n - 1$ comparisons: $O(n)$.

Merge sort splits the list in half, sorts each half the same way and merges the results. The halves keep halving until single values remain, which are sorted already.



Figure 1.5.4 — Merge sort on 11, 4, 9, 4. Splitting creates $\log_2 4 = 2$ levels of merging, and each level moves all four values once.

Count the work level by level. On every merge level the pieces together hold all n values, so a level costs $O(n)$. Halving reaches single values after about $\log_2 n$ levels: 18 levels for 200,000 values. The total is $O(n \log n)$, about 3.6 million steps for 200,000 values instead of the 2×10^{10} shifts insertion sort may need.

Pseudocode

Merging two sorted lists: $O(n)$

```
merge(L, R):
    i ← 0, j ← 0, out ← empty list
    while i < length(L) and j < length(R):
        if L[i] ≤ R[j]:           ▶ ≤ sends ties left first
            append L[i] to out; i ← i + 1
        else:
            append R[j] to out; j ← j + 1
    append the rest of L, then the rest of R, to out
    return out
```

No sort that learns about the order *only* by comparing pairs can beat $O(n \log n)$ in general: there are $n!$ possible orders, and each comparison has two outcomes, so about $\log_2(n!) \approx n \log_2 n$ comparisons are needed to tell them apart.

When counting beats comparing

The bound does not apply when the values come from a small range. Exam marks from 0 to 100 can be sorted by tallying how often each mark occurs, as in Chapter 1.4, and then writing each mark out as many times as it was counted. That is $O(n + K)$ for values in $0..K$, and it never compares two values. With readings between -10^9 and 10^9 , K is two billion and the tally does not fit; use the library sort.

1.5.5 Records with several keys

Often you sort records, not single numbers. Five contestants have (score, id) = (80, 7), (95, 4), (80, 2), (60, 9), (95, 6). The results list orders them by **decreasing score**, and two equal scores by **increasing id**. The second rule is the *tie-breaker*.

Rule	Order	Correct?
score down, then id up	(95, 4) (95, 6) (80, 2) (80, 7) (60, 9)	yes
sort ascending by (score, id), then reverse	(95, 6) (95, 4) (80, 7) (80, 2) (60, 9)	no: ties reversed too
score down only, unstable sort	ties may appear in either order	not guaranteed

A comparator decides the order of any two records: compare the first key, and only if it is equal, compare the second. In Python, a key does the same job: tuples are compared element by element, and negating the score turns “decreasing” into “increasing”.

Two keys: score down, then id up

```
# Python: tuples compare key by key
recs.sort(key=lambda r: (-r.score, r.id))

// C++: return true when x must come before y
sort(recs.begin(), recs.end(), [](const Rec& x, const Rec& y) {
    if (x.score != y.score) return x.score > y.score;
    return x.id < y.id;
});

// Java: Integer.compare never overflows
recs.sort((x, y) -> x.score != y.score ? Integer.compare(y.score, x.score)
    : Integer.compare(x.id, y.id));

// JavaScript: || falls through to the id only when the scores tie
recs.sort((x, y) => y.score - x.score || x.id - y.id);
```

A sort is **stable** if records that compare as equal keep their input order. Stability lets you sort by the tie-breaker first and then by the main key. When the comparator already names every tie-breaker, as above, no two different records compare equal, so stability no longer matters and `std::sort` is just as correct as a stable sort.

Comparators that subtract

The JavaScript comparator subtracts because JavaScript numbers are exact up to 2^{53} . In C++ and Java, $x - y$ on 32-bit `int` values can overflow: with $x = 2,000,000,000$ and $y = -1,000,000,000$ the true difference 3×10^9 does not fit and wraps to a negative number, so the comparator reports the wrong order. Compare with `<` and `>`, or `Integer.compare`, instead of subtracting. A C++ comparator must also be strict: return `false` for equal records, so use `<`, never `<=`.

Keeping the original positions

Sorting moves values, so it forgets where they came from. If the answer must mention input positions, sort *pairs* (value, position) instead of bare values. For the readings, the pairs (11, 1), (4, 2), (9, 3), (4, 4) sort to (4, 2), (4, 4), (9, 3), (11, 1): the smallest reading was taken at position 2, the largest at position 1. The position also acts as a tie-breaker, so equal values come out in input order.

1.5.6 Neighbours after sorting

Back to the opening question: which two readings are closest? Sort them as $x_1 \leq x_2 \leq \dots \leq x_n$. The claim is that the smallest difference over *all* pairs equals the smallest difference between *neighbours*, so $n - 1$ subtractions replace $n(n - 1)/2$.

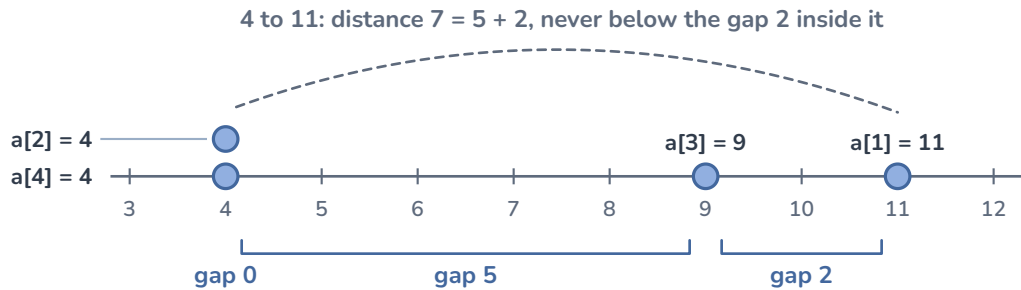


Figure 1.5.5 — Any non-adjacent distance contains an adjacent gap no larger than itself. The two readings equal to 4 are stacked; their gap is 0.

Proof. Take any pair $i < j$ that are not neighbours, so $j \geq i + 2$. The distance between them is a sum of the gaps between them:

$$x[j] - x[i] = (x[i+1] - x[i]) + (x[i+2] - x[i+1]) + \dots + (x[j] - x[j-1])$$

Every gap is at least 0 because the list is sorted, so the whole sum is at least its first gap, $x[i+1] - x[i]$. That gap belongs to a pair of neighbours. So for every pair there is a neighbouring pair that is at least as close, and the minimum over neighbours is the minimum over all pairs. In Figure 1.5.5, the pair 4 and 11 has distance 7, and the gaps 5 and 2 inside it are both smaller.

Equal values give a gap of 0 automatically, which is why duplicates do not need a separate check. The same neighbour argument shows that, after sorting, the number of *distinct* values is 1 plus the number of positive gaps: for $[4, 4, 9, 11]$ that is $1 + 2 = 3$.

1.5.7 Coordinate compression

Sometimes only the *order* of the values matters, not their size. Replacing each value by its rank among the distinct values keeps every comparison the same but shrinks the numbers to $1..d$, where d is the number of distinct values. Ranks fit in an array index, so a tally from Chapter 1.4 can be used even when the original values reach a billion.

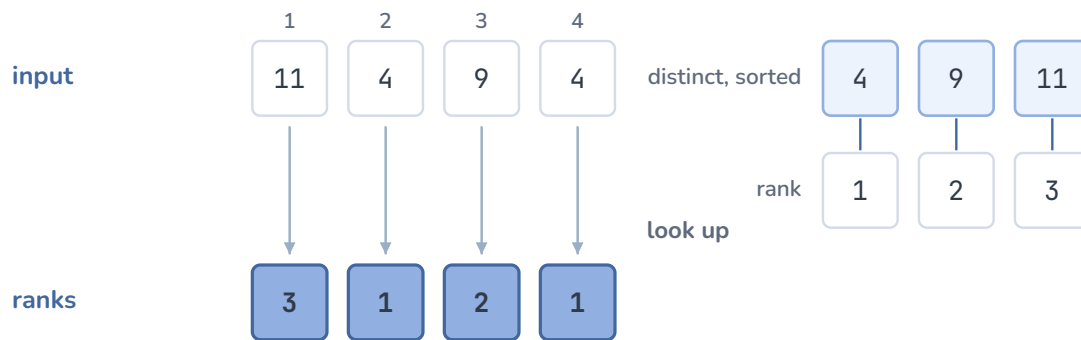


Figure 1.5.6 — Sort a copy, drop duplicates, number the distinct values from 1, then look every input value up. The input order is never changed.

The steps are: copy the array and sort the copy; walk the sorted copy and give each new value the next rank, skipping a value equal to its left neighbour (the neighbour idea again); store value $\rightarrow$ rank in a dictionary; finally go through the *original* array in its own order and replace each value by its rank. For [11, 4, 9, 4] the distinct sorted values are 4, 9, 11, so the result is [3, 1, 2, 1].

Why it is right: equal values look up the same dictionary entry, so they get equal ranks; and if $u < v$, then u appears earlier in the sorted distinct list and gets the smaller rank. Sorting costs $O(n \log n)$ and the lookups $O(n)$ with a dictionary. (Binary search on the distinct list, Chapter 2.3, works instead of a dictionary.)

1.5.8 Code in four languages

Each program solves the first practice problem, **Ordered readings**. The first line holds n and the second line the n values, each between -10^9 and 10^9 . The program prints the values in nondecreasing order on one line, separated by spaces. Equal values all stay in the output.

Sample input

```
4
11 4 9 4
```

Sample output

```
4 4 9 11
```

Every program reads all the numbers, calls the library sort with numeric order and prints the result.

Python 3

```
import sys

def main():
    data = sys.stdin.buffer.read().split()
    n = int(data[0])
    a = sorted(map(int, data[1:1 + n]))
    print(" ".join(map(str, a)))

main()
```

C++17

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    cin >> n;
    vector<int> a(n);
    for (int &x : a) cin >> x;
    sort(a.begin(), a.end());
    for (int i = 0; i < n; i++) {
        cout << a[i] << (i + 1 < n ? ' ' : '\n');
    }
    return 0;
}
```

Java 17

```
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        StreamTokenizer in = new StreamTokenizer(
            new BufferedReader(new InputStreamReader(System.in)));
        in.nextToken();
        int n = (int) in.nval;
        int[] a = new int[n];
        for (int i = 0; i < n; i++) {
            in.nextToken();
            a[i] = (int) in.nval;
        }
        Arrays.sort(a);
        StringBuilder out = new StringBuilder();
        for (int i = 0; i < n; i++) {
            out.append(a[i]).append(i + 1 < n ? ' ' : '\n');
        }
        System.out.print(out);
    }
}
```

JavaScript (Node)

```
const data = require("fs").readFileSync(0, "utf8")
  .trim().split(/\s+/).map(Number);
const n = data[0];
const a = data.slice(1, 1 + n);
a.sort((x, y) => x - y);
console.log(a.join(" "));
```

The JavaScript comparator subtracts two numbers; the difference is at most 2×10^9 , far below 2^{53} , so it is exact.

1.5.9 Cost and number sizes

Sorting n values takes $O(n \log n)$ time; reading and printing take $O(n)$. Memory is $O(n)$ for the array, plus whatever extra space the library sort uses (merge-based sorts need a second buffer of up to n values).

Values from -10^9 to 10^9 fit easily in a 32-bit `int` (limit 2,147,483,647), so sorting them needs nothing wider. But a *difference* or a sum of two of them reaches 2×10^9 , uncomfortably close to that limit. When you subtract or add values after sorting, as the gap and closest-pair ideas do, compute in 64-bit (`long long` in C++, `Long` in Java). Python integers never overflow.

1.5.10 Common mistakes

- **Lexicographic sorting in JavaScript.** `a.sort()` without a comparator sorts numbers as strings: `[2, 10]` becomes `[10, 2]`. Always pass `(x, y) => x - y` for numbers.
- **Subtracting in a C++ or Java comparator.** `x - y` can overflow 32 bits and reverse the order. Compare with `<` or `Integer.compare`.
- **Losing the original positions.** After sorting, `a[0]` is the smallest value, not the first reading. Sort (value, position) pairs, or sort a copy, when the input order is still needed.
- **Reversing to get “descending with ascending ties”.** Reversing an ascending sort also reverses the tie-breaker. Negate the main key or write the comparator directly.
- **A non-strict comparator.** In C++, a comparator that returns `true` for equal elements (`<=`) breaks `std::sort` and can crash it.
- **Removing duplicates by accident.** Putting values in a set before sorting drops repeated readings. Keep duplicates unless the problem asks for distinct values.

1.5.11 When sorting is not the right tool

Sorting is a one-time arrangement. It pays off when the data stays fixed and the question is about values, not about where they were. If the problem is about positions in the input, such as “the longest increasing stretch of consecutive days”, sorting destroys exactly the information you need.

Situation	Better starting point
Values in a small range $0..K$	Counting, $O(n + K)$ (Chapter 1.4)
Values keep arriving; need the smallest after each one	Priority queue, $O(\log n)$ per step (Chapter 3.3)
Answer depends on the input order of positions	Work on the original array, or sort (value, position) pairs
Many “how many values are $\leq x$?” questions	Sort once, then binary search (Chapter 2.3)
Pairs with a given sum in sorted data	Two pointers (Chapter 2.2)

The last two rows are a preview: sorting is the first step of many later techniques, including two pointers (Chapter 2.2), binary search (Chapter 2.3) and many greedy rules (Chapter 2.5).

Exercises

Difficulty: ● quick check, ●● needs an extra idea. Try each one before reading the answers.

- Why can `[2, 10].sort()` in JavaScript, without a comparator, give a surprising result? What does it return?
- Do you need a stable sort when every tie-breaker is written into the comparator? Explain.
- Run insertion sort on `[5, 2, 4, 1]`. How many shifts does it make in total?
- Merge the sorted lists `[1, 6, 8]` and `[2, 3, 9]`. How many comparisons are made before one list is used up?
- What is the smallest difference between two of the values 15, 3, 9, 20, 11? Which pair gives it, and how many subtractions did you need after sorting?
- Give the compressed ranks of `[-5, 100, -5, 7]`, in the input order.

Answers and explanations

Read these after attempting the exercises.

1. Without a comparator, JavaScript compares the values as strings. "10" comes before "2" because the character 1 comes before 2, so the result is **[10, 2]**. Pass $(x, y) \Rightarrow x - y$ to sort numbers by size.
2. **No**. Stability only decides the order of records that compare as equal. If the comparator breaks every tie (for example by a distinct id), no two different records are equal, so any correct sort gives the same order.
3. $[5 \mid 2 \ 4 \ 1] \rightarrow [2 \ 5 \mid 4 \ 1]$ (1 shift) $\rightarrow [2 \ 4 \ 5 \mid 1]$ (1 shift) $\rightarrow [1 \ 2 \ 4 \ 5]$ (3 shifts). Total **5**, one per out-of-order pair: (5, 2), (5, 4), (5, 1), (2, 1), (4, 1).
4. $1 \text{ vs } 2 \rightarrow 1$; $6 \text{ vs } 2 \rightarrow 2$; $6 \text{ vs } 3 \rightarrow 3$; $6 \text{ vs } 9 \rightarrow 6$; $8 \text{ vs } 9 \rightarrow 8$. The left list is now used up after **5** comparisons, and 9 is copied: [1, 2, 3, 6, 8, 9].
5. Sorted: 3, 9, 11, 15, 20; neighbouring gaps 6, 2, 4, 5. The smallest is **2**, between 9 and 11. Only 4 subtractions were needed after sorting, instead of the 10 pairs of the unsorted list.
6. Distinct sorted values -5, 7, 100 get ranks 1, 2, 3, so the answer is **[1, 3, 1, 2]**. Both copies of -5 receive rank 1.

Practice this topic

Four coding problems for this chapter are in the **Practice** tab. Open any of them in any order; the label shows difficulty within this topic.

Ordered readings

[Apply](#)

Print the input values in nondecreasing order.

Results order

[Connect](#)

Each participant has a distinct integer id and a score. Print ids by decreasing score; equal scores are ordered by increasing id.

Closest readings

[Reason](#)

Find the smallest absolute difference between values at two different positions.

Compact coordinates

[Stretch](#)

Replace every array value by its rank among the distinct values in increasing order. Ranks start at 1; equal values receive equal ranks. Print ranks in the original input order.
