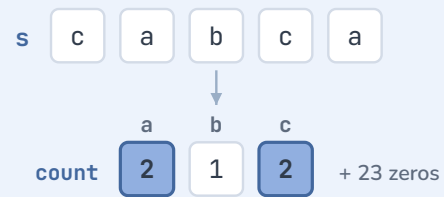


Counting and frequency

Chapter 1.4 · Overall level: Foundation



How many times does each value appear? Which value is the most common? Are two words made of the same letters? *All three questions have the same answer: stop searching and start counting.*

A **frequency** is the number of times a value has occurred. Keeping one counter per value, and adding 1 each time the value turns up, replaces repeated searching with a single pass. This chapter shows how to store those counters and what you can do with them.

1.4.1 From repeated searching to a tally

A sensor reports small whole numbers from 0 to 9. Five readings arrive: 2, 5, 2, 2, 5. How often did each value occur?

One way is to take each possible value in turn and scan the whole list counting it: is 0 there? is 1? ... is 9? With n readings and d possible values that is $n \times d$ comparisons, and most of them find nothing. For 200,000 readings that may each be any of 26 letters, it is over five million comparisons, and the cost grows with every extra possible value.

A tally does it in one pass. Keep a counter for every possible value, all starting at 0. Read each value once and add 1 to *its* counter. Nothing is searched: the value itself tells you which counter to change.

1.4.2 A frequency array for a small range of values

When the values are small non-negative integers, the counters can live in an array indexed by the value: `freq[v]` counts how often v has appeared. For readings from 0 to 9 the array has 10 slots.

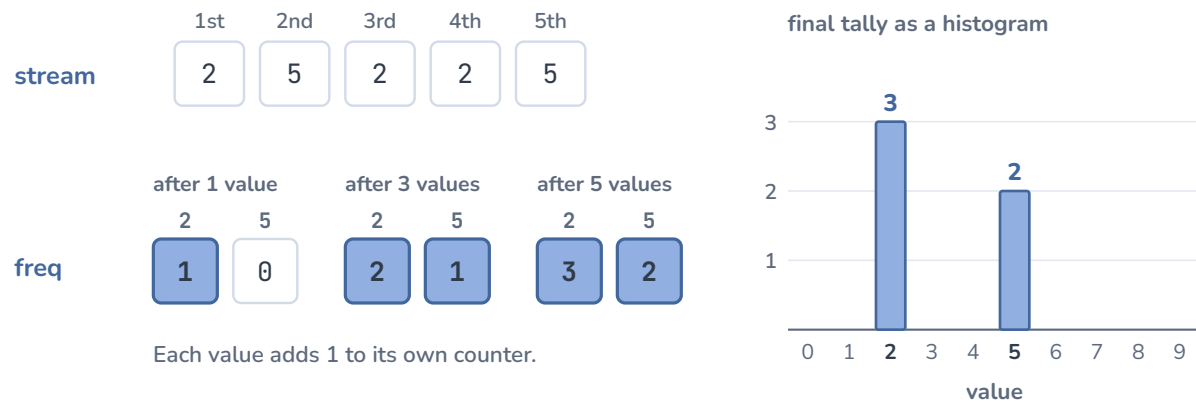


Figure 1.4.1 — A frequency stores how many times a value has occurred, not its most recent position.

```
freq[0..9] ← 0
for each reading v:
    freq[v] ← freq[v] + 1
```

The loop keeps a simple invariant: after k readings, $\text{freq}[v]$ is the number of times v occurs among the first k . Each reading changes exactly one counter, by exactly 1, so the invariant survives every step, and at the end every counter is exact. The cost is $O(n)$ for the pass plus $O(d)$ to create and print d counters.

Counting letters

Letters are not numbers, but every character has a numeric code, and the codes of 'a' to 'z' are consecutive. Subtracting the code of 'a' turns 'a' into 0, 'b' into 1 and 'z' into 25: exactly the index of a 26-slot array.

Python	C++	Java	JavaScript
<code>ord(ch) - ord("a")</code>	<code>ch - 'a'</code>	<code>s.charAt(i) - 'a'</code>	<code>s.charCodeAt(i) - 97</code>

For the string *banana* the counters for a, b and n end at 3, 1 and 2, and the other 23 stay 0. Absent letters still have a counter; it simply holds 0.

1.4.3 Dictionaries for large or negative values

An array of counters only works when the values are small. If readings range from -10^9 to 10^9 , an array indexed by value would need two billion slots, and a negative value is not a valid index at all. Yet a list of 200,000 readings contains at most 200,000 *different* values.

A **dictionary** (also called a map or hash map) stores counters only for the values that actually appear. Each entry links a **key**, the value being counted, to its count. For the readings 1,000,000,000, -7, 1,000,000,000 it holds just two entries: $1,000,000,000 \rightarrow 2$ and $-7 \rightarrow 1$.

	Add 1 to the count of v	Count of v (0 if absent)
Python	<code>freq[v] = freq.get(v, 0) + 1</code>	<code>freq.get(v, 0)</code>
C++	<code>unordered_map<int, int> freq; freq[v]++;</code>	<code>freq.count(v) ? freq[v] : 0</code>
Java	<code>HashMap<Integer, Integer> freq; freq.merge(v, 1, Integer::sum);</code>	<code>freq.getOrDefault(v, 0)</code>
JavaScript	<code>const freq = new Map(); freq.set(v, (freq.get(v) 0) + 1);</code>	<code>freq.get(v) 0</code>

A hash map finds a key in expected $O(1)$ time: on ordinary data each operation takes a small constant time, so counting n values costs expected $O(n)$ time and $O(d)$ memory for d distinct values. You do not need to know how hashing works inside; just remember that it is a little slower than an array, and that an array is the better choice whenever the values are small. C++ also offers `map`, which keeps its keys sorted at $O(\log d)$ per operation. In JavaScript use a `Map`, not a plain object, because an object turns every key into a string.

1.4.4 The most frequent value, with a tie rule

The most frequent value is called the **mode**. Counting is the easy half; choosing needs a precise rule, because several values can share the largest count. A statement might say: “if several values tie, choose the smallest”. Take the readings 7, −3, 7, 4, −3, 4, 1. The counts are $7 \rightarrow 2$, $-3 \rightarrow 2$, $4 \rightarrow 2$ and $1 \rightarrow 1$, so three values tie and the answer must be −3.

Scan the counted entries once, keeping the best value and its count, and replace the best when the new entry is *better*:

```

better = count > bestCount
        or (count = bestCount and value < bestValue)

```

Entry (in insertion order)	Compared with the best so far	Best after
$7 \rightarrow 2$	first entry	7 (count 2)
$-3 \rightarrow 2$	same count, smaller value: better	−3 (count 2)
$4 \rightarrow 2$	same count, larger value: not better	−3 (count 2)
$1 \rightarrow 1$	smaller count: not better	−3 (count 2)

Do not rely on the order in which a dictionary hands back its keys. A rule such as “keep the first maximum I meet” would answer 7 here, because 7 was inserted first. The comparison has to state both the main rule and the tie rule.

1.4.5 Comparing collections by their counts

Two words are **rearrangements** of each other if one can be turned into the other by reordering its letters, using every letter exactly once. `below` and `elbow` are; `aab` and `abb` are not, even though both have three letters.

Rearranging changes positions, never counts, so two rearrangements have the same count for every letter. The converse holds too: if every letter has the same count in both words, match the first `a` of one with the first `a` of the other, the second with the second, and so on for every letter; every character finds exactly one partner. So comparing the 26 counters answers the question completely.

Letter	a	b	e	l	o	w	all others
<code>below</code>	0	1	1	1	1	1	0
<code>elbow</code>	0	1	1	1	1	1	0
<code>aab</code>	2	1	0	0	0	0	0
<code>abb</code>	1	2	0	0	0	0	0

Equal length is necessary but not enough: `aab` and `abb` have the same length and different counts. Different lengths, on the other hand, settle the matter at once. Sorting both words and comparing them letter by letter also works (Chapter 1.5), but costs $O(n \log n)$; counting costs $O(n + 26)$.

1.4.6 Counting equal pairs as values arrive

How many pairs of positions $i < j$ hold equal values? Checking every pair is $O(n^2)$, as Chapter 1.2 showed. Counting does better. When a new value arrives, it forms a pair with every *earlier* copy of itself, and the tally already knows how many earlier copies there are. So add the current count, *then* increment it.

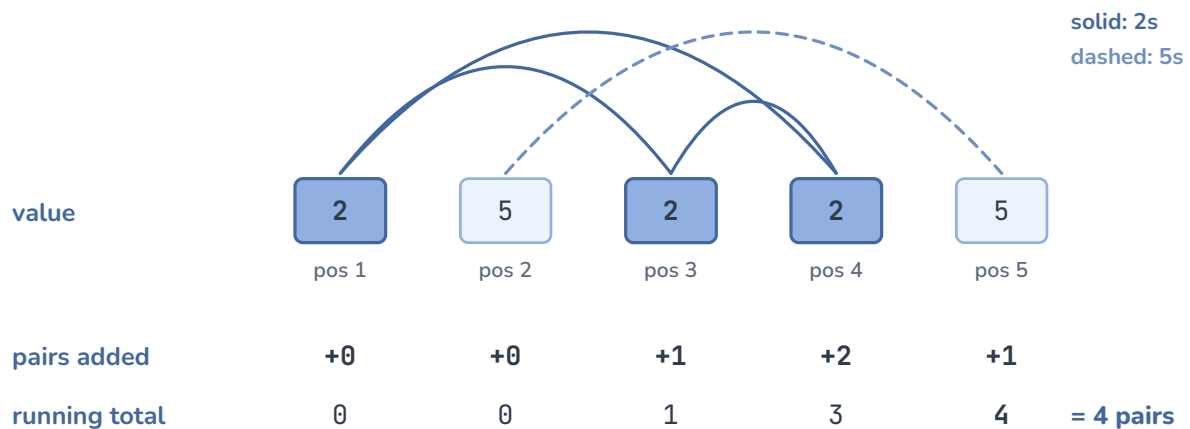


Figure 1.4.2 — When a value arrives, every earlier equal value creates one new pair.

Position	Value	freq[value] before	Pairs added	Running total	freq[value] after
1	2	0	0	0	1
2	5	0	0	0	1
3	2	1	1	1	2
4	2	2	2	3	3
5	5	1	1	4	2

Pseudocode

Equal pairs: expected $O(n)$ time, $O(d)$ extra memory

```

pairs ← 0
freq ← empty dictionary
for each value v in order:
    pairs ← pairs + freq[v]           ← earlier copies of v (0 if none)
    freq[v] ← freq[v] + 1
output pairs

```

Why every pair is counted exactly once

Take any pair of positions $i < j$ with equal values. When position j is processed, position i has already been counted into $\text{freq}[v]$, so the pair is included in the amount added at step j . It is not added at any other step: at step i the later position j has not been seen yet, and at other steps neither endpoint is the new value. Because the count is read *before* the increment, a position never pairs with itself. So the total is exactly the number of equal pairs.

As a check, a value that appears c times forms $c(c - 1)/2$ pairs (Chapter 1.2). Here 2 appears three times and 5 twice: $3 + 1 = 4$.

Watch the size of the answer. If all 200,000 values are equal, there are $200,000 \times 199,999 / 2 = 19,999,900,000$ pairs, more than nine times the 32-bit `int` limit. Keep `pairs` in a `long long` or `Long`; the individual counters still fit in an `int`.

1.4.7 Code in four languages

Each program solves the first practice problem, **Alphabet tally**. The input is one string of lowercase English letters, 1 to 200,000 characters long. The output is 26 integers on one line: how many times a, b, ..., z occur, in that order.

Sample input	Sample output
cabca	2 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Each program reads the string as one token, keeps an array of 26 counters, adds 1 at index “character minus 'a'”, and prints all 26 counters, zeros included. Time $O(n + 26)$, extra memory $O(26)$.

Python 3

```
import sys

def main():
    s = sys.stdin.read().split()[0]
    count = [0] * 26
    for ch in s:
        count[ord(ch) - ord("a")] += 1
    print(" ".join(map(str, count)))

main()
```

C++17

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

int main() {
    string s;
    cin >> s;
    vector<int> count(26, 0);
    for (char ch : s) count[ch - 'a']++;
    for (int k = 0; k < 26; k++) {
        cout << count[k] << (k < 25 ? ' ' : '\n');
    }
    return 0;
}
```

Java 17

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

public class Main {
    static BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
    static StringTokenizer tok = new StringTokenizer("");

    static String next() throws IOException {
        while (!tok.hasMoreTokens()) tok = new StringTokenizer(in.readLine());
        return tok.nextToken();
    }

    public static void main(String[] args) throws IOException {
        String s = next();
        int[] count = new int[26];
        for (int i = 0; i < s.length(); i++) count[s.charAt(i) - 'a']++;
        StringBuilder out = new StringBuilder();
        for (int k = 0; k < 26; k++) {
            out.append(count[k]).append(k < 25 ? ' ' : '\n');
        }
        System.out.print(out);
    }
}
```

JavaScript (Node)

```
const s = require("fs").readFileSync(0, "utf8").trim().split(/\s+/)[0];
const count = new Array(26).fill(0);
for (let i = 0; i < s.length; i++) count[s.charCodeAt(i) - 97]++;
console.log(count.join(" "));
```

In C++ and Java, subtracting two characters gives an `int`, so `ch - 'a'` can be used directly as an index. JavaScript has no character type: `charCodeAt(i)` returns the code, and 97 is the code of `'a'`.

1.4.8 Common mistakes

- **Using the value as an array index when values may be negative or huge.** `freq[-7]` is an error (or, in Python, silently counts from the end), and `freq[1000000000]` needs a billion slots. Use a dictionary unless the values are small and non-negative.
- **Ignoring the tie rule.** “The most frequent value” is not well defined until the statement says how to break ties. Compare by count first, then by the stated tie rule.
- **Incrementing before adding.** In pair counting, adding `freq[v]` after the increment pairs each position with itself and gives n too many.
- **Overflow in pair counts.** $n(n-1)/2$ is about 2×10^{10} for $n = 200,000$. Use a 64-bit total.
- **Assuming equal lengths mean equal letters.** Compare all 26 counts.
- **Forgetting absent values.** A letter that never occurs still needs its 0 in the output.

1.4.9 Where counting is not enough

A tally forgets order and position: it knows that 2 occurred three times, not where. If a problem asks *where* equal values are, store a list of positions for each value instead of a number. If it asks for counts inside many different stretches of the array, recounting each stretch is slow; prefix sums of counts answer each question in $O(1)$ (Chapter 2.1). If the questions are about order, such as “the third smallest value”, sorting (Chapter 1.5) is the natural tool. And if the counts change between questions, Chapter 4.2 keeps them in a structure that updates quickly.

A hash map’s $O(1)$ is an expected cost on ordinary data. Inputs built specifically to collide inside a hash table can slow it down; when that is a concern, a sorted map with $O(\log d)$ operations, or sorting the values first, avoids the risk.

Exercises

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

1. ● If a value has appeared 4 times, how many new pairs does the next copy add?
2. ● Does equal length imply equal character counts?
3. ● Write the 26-counter tally of banana as a short list of the non-zero entries. Which indices of the array are non-zero?
4. ● Readings are integers between 0 and 100, or between -10^9 and 10^9 . Which kind of counter store suits each, and why?
5. ●● Find the most frequent value of 4, 9, 4, 9, 1, 1, 6 if ties go to the smallest value. What would “keep the first maximum met” give?
6. ●● Run the pair-counting loop on 1, 1, 1, 1. Give the amount added at each step and the total, and check it with the formula.

Answers and explanations

Read these after attempting the exercises.

1. **4**. The new copy forms a pair with each of the 4 earlier copies, and with nothing else.
2. **No**. aab and abb have the same length but different counts of a and b.
3. $a \rightarrow 3, b \rightarrow 1, n \rightarrow 2$. The non-zero indices are **0** (a), **1** (b) and **13** (n, because $'n' - 'a' = 13$).
4. For 0 to 100, an **array** of 101 counters: direct indexing, fast and small. For $\pm 10^9$, a **dictionary**: an array would need two billion slots and cannot use negative indices, while the dictionary only stores values that occur.
5. Counts: $4 \rightarrow 2, 9 \rightarrow 2, 1 \rightarrow 2, 6 \rightarrow 1$. Three values tie at 2, so the answer is **1**. “Keep the first maximum” would give 4, the first value inserted.
6. Added: 0, 1, 2, 3; total **6**. The formula gives $4 \times 3 / 2 = 6$.

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.

Alphabet tally

Apply

Count the occurrences of each lowercase English letter in a string.

Most common reading

Connect

Find the value appearing most often. If several values tie, print the smallest of them.

Rearrange the label

Reason

Decide whether one lowercase string can be rearranged to equal another. Every character must be used exactly once.

Equal-reading pairs

Stretch

Count pairs of positions $i < j$ whose values are equal.
