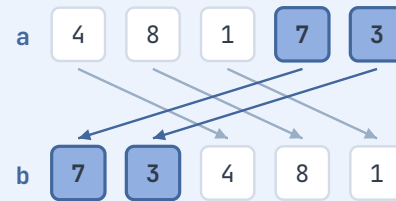


Arrays and strings

Chapter 1.3 · Overall level: Foundation



Almost every program stores a list of values and walks through it. *Where exactly does the list begin and end, and how do you move its values around without losing any?*

This chapter is about getting those details right: numbering positions, scanning forwards and backwards, following a run of equal characters, rotating a list and storing a grid. The ideas are simple; the care with boundaries is what makes programs correct.

1.3.1 Positions, indices and values

An **array** stores n values in numbered slots, side by side in memory. You reach any slot directly by its number, called its **index**, so reading or changing `a[i]` takes the same short time wherever i is. Keep two things apart: the index says *where* a value is, the value says *what* is stored there.

Problem statements usually count positions from 1, as people do. All four of our languages count indices from 0. So position p in the statement is index $p - 1$ in the code, the first value is `a[0]`, and the last is `a[n - 1]`.

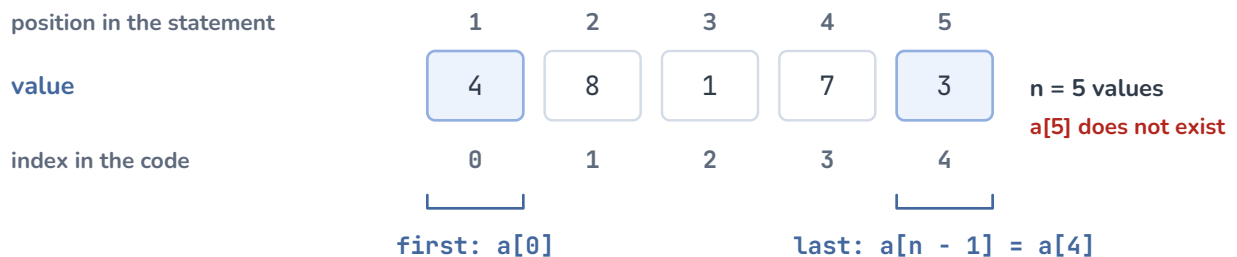


Figure 1.3.1 — The same five values seen two ways. Indices run from 0 to $n - 1$; there is no `a[n]`.

Two small cases are worth checking every time. A list of one value has first and last index both 0, and a loop over it runs once. An empty list has no valid index at all, and a loop over it must run zero times; a loop written as “from 0 to $n - 1$ ” does exactly that.

Arrays in four languages

	Python	C++	Java	JavaScript
n zeros	<code>[0] * n</code>	<code>vector<int> a(n)</code>	<code>new int[n]</code>	<code>Array(n).fill(0)</code>
length	<code>len(a)</code>	<code>a.size()</code>	<code>a.length</code>	<code>a.length</code>
read / change	<code>a[i]</code>	<code>a[i]</code>	<code>a[i]</code>	<code>a[i]</code>

Reading or writing `a[n]` is a bug in every language, but the symptoms differ. Python, Java and JavaScript catch it at run time: Python and Java stop with an error, while JavaScript quietly returns `undefined`, which later turns into `NaN`. C++ does not check at all: the program may read a neighbouring value or crash, so the bug can hide.

1.3.2 Strings are sequences of characters

A **string** behaves like an array of characters. For `s = "aabbcca"` the length is 7, `s[0]` is `'a'`, `s[4]` is `'b'` and the last character is `s[6]`. Characters are compared with `==` just like numbers. In this library strings contain plain lowercase letters, one character per letter, so the index of a character is simply its position minus one.

	Python	C++	Java	JavaScript
read one word	<code>s = data[0]</code>	<code>cin >> s</code>	<code>s = next()</code>	<code>s = data[0]</code>
length	<code>len(s)</code>	<code>s.size()</code>	<code>s.length()</code>	<code>s.length</code>
character i	<code>s[i]</code>	<code>s[i]</code>	<code>s.charAt(i)</code>	<code>s[i]</code>

One difference matters for speed. In Python, Java and JavaScript a string cannot be changed after it is made, so `t = t + c` builds a brand-new string each time. Doing that n times copies about $n^2/2$ characters. Collect the pieces in a list (or a `StringBuilder` in Java) and join them once at the end.

1.3.3 Scanning forwards and backwards

A forward scan visits indices `0, 1, ..., n - 1`. A backward scan starts at the last valid index and stops after index `0`:

```
forward:    for i from 0 to n - 1
backward:   for i from n - 1 down to 0
```

Each language writes the backward loop differently, and each is easy to get wrong by one. In Python it is `range(n - 1, -1, -1)`: the stop value `-1` is excluded, so `0` is the last index visited. In C++, Java and JavaScript it is `for (i = n - 1; i >= 0; i--)`.

Reversing a list is a backward scan. The output in place j is the input value at index $n - 1 - j$. Check both ends: $j = 0$ gives $a[n - 1]$, the last value, and $j = n - 1$ gives $a[0]$. For 4 8 1 7 3 the output is 3 7 1 8 4. Each index is used exactly once, in decreasing order, so every value appears once and none is lost.

1.3.4 The longest run of equal characters

A **run** is a stretch of equal neighbouring characters. In `aabbbca` the runs are `aa`, `bbb`, `c` and `a`, of lengths 2, 3, 1 and 1. To find the longest in one scan, keep two variables:

- **current**: the length of the run that ends at the character just read;
- **best**: the longest run seen anywhere so far.

At each new character, compare it with the previous one. If they are equal, the run continues and **current** grows by one; otherwise a new run starts and **current** resets to 1. Then **best** takes the larger of itself and **current**.

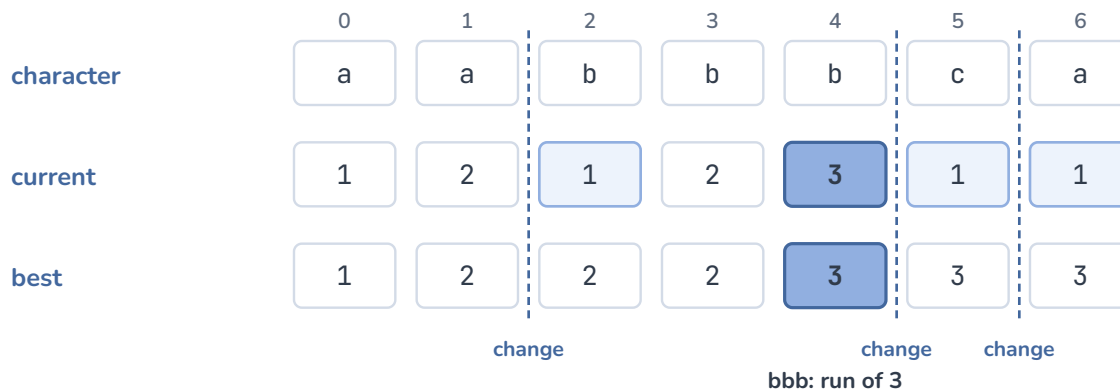


Figure 1.3.2 — The best run seen so far never decreases. Dashed lines mark where the character changes and current starts again at 1.

Index	0	1	2	3	4	5	6
character	a	a	b	b	b	c	a
same as previous?	—	yes	no	yes	yes	no	no
current	1	2	1	2	3	1	1
best	1	2	2	2	3	3	3

Pseudocode

Longest run: $O(n)$ time, $O(1)$ extra memory

```

current ← 1
best ← 1
for i from 1 to n - 1:           ← s[0] alone is a run of 1
    if s[i] = s[i - 1]:
        current ← current + 1
    else:
        current ← 1
    best ← max(best, current)
output best

```

Why it is correct

The loop keeps an invariant: after index i is processed, `current` is the length of the run ending at i , and `best` is the longest run ending at any index from 0 to i . It holds at the start ($i = 0$, both are 1). If $s[i]$ equals $s[i - 1]$, the run ending at i is the run ending at $i - 1$ plus one character; otherwise it is $s[i]$ alone. Taking the maximum keeps `best` right. Every run ends somewhere, so after the last index `best` is the longest run of all.

Notice that `best` is updated at every index, not only when a run ends. A version that updates it only on a change forgets the final run: for `abbb` it would answer 1 instead of 3. If you prefer that style, you must update `best` once more after the loop.

1.3.5 Rotation by an index mapping

A **right rotation by 1** moves the last value to the front and shifts every other value one place right. Rotating right by k does that k times. Instead of moving values k times, ask where each value ends up. Each step moves every value one index to the right, wrapping from the end back to index 0, so after k steps the value at index i sits at index $(i + k) \bmod n$.

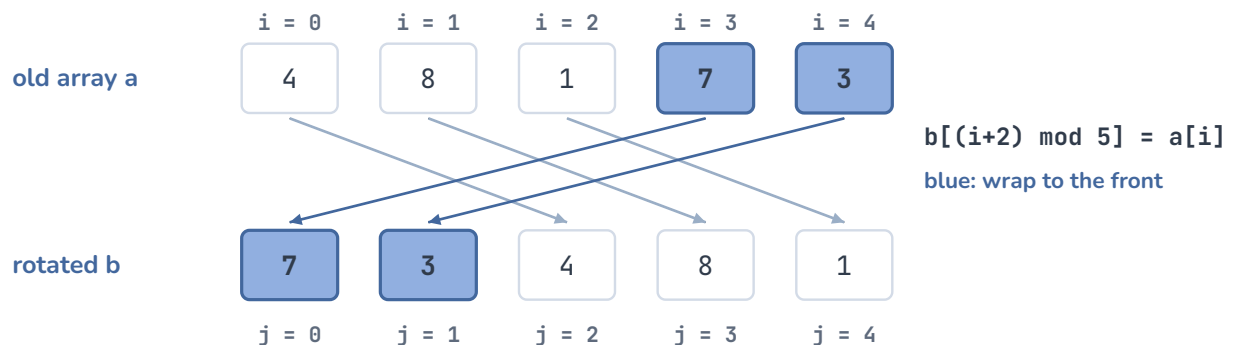


Figure 1.3.3 — Rotation changes positions, while each value is preserved.

For $[4, 8, 1, 7, 3]$ and $k = 2$: index 0 goes to 2, 1 to 3, 2 to 4, 3 to $(3 + 2) \bmod 5 = 0$ and 4 to 1, giving $[7, 3, 4, 8, 1]$. The mapping sends different old indices to different new ones, so every value lands in exactly one slot.

Large k , and building the result safely

Rotating n times returns every value to where it started, so only $k \bmod n$ matters. Reduce k first: a rotation by 10^{12} of 5 values is a rotation by 0. Simulating 10^{12} single steps would never finish.

It is just as natural to fill the result slot by slot: $b[j] = a[(j - k) \bmod n]$. Be careful with the remainder of a negative number. Python's `%` returns a value from 0 to $n - 1$, but in C++, Java and JavaScript $-2 \% 5$ is -2 . Writing $(j - k \% n + n) \% n$ keeps the index in range in every language.

Write into a *new* array b , or print directly from the mapping. Moving values inside a one at a time overwrites values that have not been moved yet: setting $a[2] = a[0]$ destroys the 1 that should go to index 4.

1.3.6 Grids and row-major input

A grid with n rows and m columns is stored as an array of rows: $g[r][c]$ is the value in row r and column c , both counted from 0. Input gives a grid one row per line, left to right, top to bottom. This order is called **row-major**: the column changes fastest.

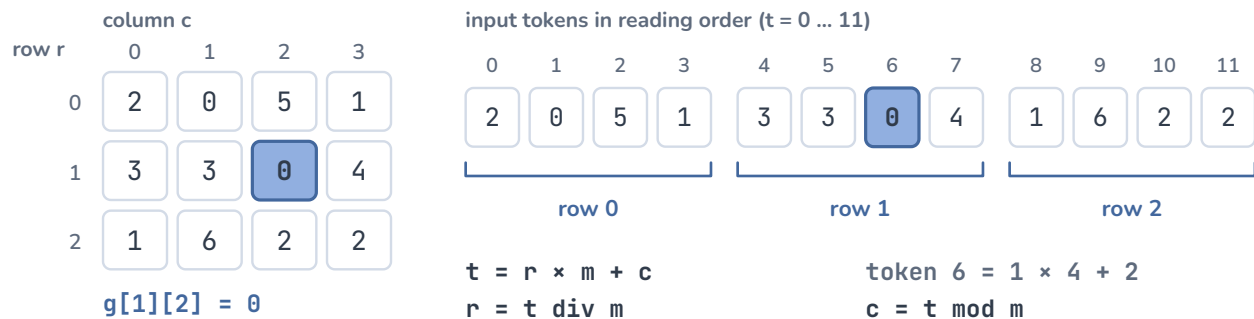


Figure 1.3.4 — A grid and its input tokens. Reading row by row, token t lands in row $t \text{ div } m$ and column $t \bmod m$.

```
read n and m
for r from 0 to n - 1:
    for c from 0 to m - 1:
        read g[r][c]
```

Once stored, the grid can be scanned in any order. To find the largest value in each *column*, let the outer loop run over columns and the inner loop over rows. For the grid in Figure 1.3.4 the column maxima are 3, 6, 5 and 4. Swapping the loops changes the order of the visits, not the set of cells visited: both versions touch each of the nm cells exactly once.

In Python a grid is a list of lists, `[[0] * m for _ in range(n)]`; writing `[[0] * m] * n` instead creates n references to the *same* row, so changing one row changes them all. C++ uses `vector<vector<int>>`, Java `int[n][m]` and JavaScript an array of arrays.

1.3.7 The cost of a scan

A scan does a fixed amount of work per element, so it is $O(n)$ for a list and $O(nm)$ for a grid. The run counter and the rotation mapping need only $O(1)$ extra variables. Reversal and rotation produce an output of size n , so storing the result costs $O(n)$ memory, which is unavoidable if you build it before printing. The repeated-concatenation trap of Section 1.3.2 is the one hidden $O(n^2)$ in this chapter.

1.3.8 Code in four languages

Each program solves the first practice problem, **Reverse the queue**. The first line holds n ($1 \leq n \leq 200,000$), the second line n integers between -10^6 and 10^6 . The output is the same integers from last to first, separated by spaces, on one line.

Sample input	Sample output
5 4 8 1 7 3	3 7 1 8 4

Every program stores the values in an array, then scans it backwards from index $n - 1$ to index 0 and prints with a space between values. Values fit easily in an `int`.

Python 3

```
import sys

def main():
    data = sys.stdin.read().split()
    n = int(data[0])
    a = [int(x) for x in data[1:n + 1]]
    out = []
    for i in range(n - 1, -1, -1):
        out.append(a[i])
    print(" ".join(map(str, out)))

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 i = 0; i < n; i++) cin >> a[i];
    for (int i = n - 1; i >= 0; i--) {
        cout << a[i] << (i > 0 ? ' ' : '\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 {
        int n = Integer.parseInt(next());
        int[] a = new int[n];
        for (int i = 0; i < n; i++) a[i] = Integer.parseInt(next());
        StringBuilder out = new StringBuilder();
        for (int i = n - 1; i >= 0; i--) {
            out.append(a[i]).append(i > 0 ? ' ' : '\n');
        }
        System.out.print(out);
    }
}
```

JavaScript (Node)

```
const data = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const n = Number(data[0]);
const a = [];
for (let i = 0; i < n; i++) a.push(Number(data[1 + i]));
const out = [];
for (let i = n - 1; i >= 0; i--) out.push(a[i]);
console.log(out.join(" "));
```

Python builds the output list and joins it once; Java collects it in a `StringBuilder`. Printing 200,000 values one call at a time would be much slower in those languages. Each program runs in $O(n)$ time and $O(n)$ memory.

1.3.9 Common mistakes

- **Off by one at the last item.** The last index is $n - 1$. A loop up to and including n reads past the end; a backward loop that stops at 1 forgets `a[0]`. Test with $n = 1$.
- **Forgetting to finalize the last run.** If `best` is only updated when the character changes, a string such as `abbbb` never records its last run. Update at every index, or once more after the loop.
- **Overwriting source values during a rotation.** Write into a separate array, or compute each output value straight from the mapping.
- **Simulating a huge rotation step by step.** Reduce k modulo n first; k may be far larger than n .
- **A negative remainder.** In C++, Java and JavaScript, $(j - k) \% n$ can be negative. Add n before taking the remainder.
- **Mixing positions and indices.** “Position 3” in a statement is `a[2]` in the code. Convert once, when reading, and stay with indices.
- **Building a long string piece by piece.** Collect the pieces and join them once.

1.3.10 Where plain arrays stop being enough

Arrays are excellent at reading and replacing values by index, but poor at changing their shape. Inserting or removing a value at the front shifts everything after it, which costs $O(n)$ each time; doing that for every element is $O(n^2)$. That is exactly why the rotation above uses an index mapping instead of repeated moves. When a program must keep adding and removing at both ends, it uses a double-ended queue, which you will meet with breadth-first search in Chapter 3.1.

Two more limits. If many questions ask for totals of different stretches of one array, scanning each stretch again is slow; Chapter 2.1 answers them in $O(1)$ each. And the strings here are plain lowercase letters: real text may contain accents, emoji and other characters that take several bytes or symbols, and searching for patterns inside long strings needs the techniques of Chapter 4.5.

Exercises

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

- What is the last valid index of an array of length 6?
- What is a right rotation by 5 on an array of length 5?
- This loop should print a reversed copy of `a`: `for j from 0 to n - 1: print a[n - j]`. Try it with $n = 3$ and say which indices it reads. Fix it.
- Write the rows `current` and `best` for the string `zzyzzz`.
- Rotate `[1, 2, 3, 4, 5]` right by $k = 7$. Which index does the value 1 move to?
- A grid has $m = 4$ columns. In which row and column does input token number 10 land (counting tokens, rows and columns from 0)?

Answers and explanations

Read these after attempting the exercises.

5. Indices run from 0 to $n - 1$.
- The array is **unchanged**. Each value moves to $(i + 5) \bmod 5 = i$; in general only $k \bmod n$ matters.
- It reads `a[3]`, `a[2]`, `a[1]`: `a[3]` does not exist and `a[0]` is never printed. The correct index is $n - 1 - j$, which runs 2, 1, 0.
- `current = 1, 2, 1, 1, 2, 3` and `best = 1, 2, 2, 2, 3`. The run `zzz` at the end only counts because `best` is updated at every index.
- $7 \bmod 5 = 2$, so the result is `[4, 5, 1, 2, 3]`, and the value 1 moves from index 0 to index $(0 + 2) \bmod 5 = 2$.
- Row $10 \div 4 = 2$, column $10 \bmod 4 = 2$: the third value of the third row. Check: $2 \times 4 + 2 = 10$.

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.

Reverse the queue

Apply

Print an array in reverse order.

Longest signal

Connect

A lowercase string records a sequence of signals. Print the longest consecutive run of one repeated character.

Rotate the display

Reason

Move the last element to the front k times. Print the resulting array. k may exceed the array length.

Row and column totals

Stretch

Given an n by m integer grid, print the sum of each row and then the sum of each column.