Resume-Day

Learn. Practise. Build.

A COMPLETE LEARNING AND PRACTICE WORKBOOK

Data Structures & Algorithms

Using C

Understand the idea. See each step.

Write the code. Explain your answer.

Learn — Practise — Build

CHAPTERS 01-10 / COMPLETE SAMPLE

The first ten chapters, with programs and practice answers.

Preview the teaching, diagrams and page design.

Full edition includes the extra workbook and code archive.

Contents - click a chapter or workbook title

Before you begin 3

01. What are data structures and algorithms? 4

02. C revision: the tools we need for DSA 10

03. Understand the problem before writing code 18

04. Write the steps, draw the flow and dry-run the result 24

05. Time complexity: how does the amount of work grow? 30

06. Memory use, best and worst cases, and choosing an approach 37

07. Arrays: positions, capacity and everyday operations 45

08. Minimum, maximum, second largest and simple counting 52

09. Reverse, rotate and rearrange array values 61

10. Prefix sums, suffix sums and range questions 68

Before you begin

This book is for students who have learned the basics of C and now want to solve problems with it. You do not need C++. If variables, loops, arrays or pointers are new to you, work through a beginner C course first. Chapter 2 is a revision chapter, not a replacement for learning C from the beginning.

How to study one chapter

Read the explanation and follow the diagram. Before running a program, predict its output. Then type or copy the complete source file into your editor, compile it, and run it. Finally close the answer and try the practice questions yourself.

A dry run means following a program by hand. Write down the values that change. If a result surprises you, go back to the first step where your table and the program disagree.

Compile and run a complete example

Save a complete program with the filename printed above it. Use a C17 compiler such as GCC or Clang. An editor alone does not compile C. These commands assume your compiler is installed and available in the terminal.

Build with GCC

```
gcc -std=c17 -Wall -Wextra -Wpedantic lab_seats.c -o lab_seats
```

Run the built program

```
./lab_seats
# Windows PowerShell: .\lab_seats.exe
```

Code fragments explain a small part of a solution; they are not all separate runnable files. Complete programs include main and are included in the companion archive. Output boxes show the expected output for the supplied example, not every possible input.

The chapter material and the extra workbook

Chapters 1-60 contain the full teaching sequence, diagrams, 467 numbered practice questions with answers, 164 complete programs, two guided projects and 300 interview answers across the chapters. The full edition then adds eight worked programs, debugging tasks, blank trace sheets with separate answers, further practice, project extension plans and mock interviews.

The sample contains Chapters 1-10 in full. It does not contain the later chapters or the extra workbook.

Use answers as a teacher, not a shortcut

Try a question first. If you are stuck, write one small input and a possible output. Read a hint or the first step of the answer, then try again. Being able to explain why your answer works matters more than remembering code.

An error tells you what to check next. It does not mean you cannot learn programming. Start with small inputs, fix one issue at a time, and keep a short note of what you learned.

Edition: November 2026. All page numbers in the contents use the PDF page numbers, including the cover. Click a contents entry or use the PDF reader's bookmarks to move around.

Copyright 2026 Resume-Day. All rights reserved. This purchase is for personal learning. Do not redistribute or resell the ebook or companion files. You may adapt the example programs for your own learning projects.

FOUNDATIONS

01. What are data structures and algorithms?

You already know how to write small C programs. Now we will learn how to organise values, choose useful steps and explain why a solution works. Start slowly: DSA is a skill you build through practice, not a list of programs to remember.

What does DSA mean?

DSA stands for **Data Structures and Algorithms**. Let us separate those words before using them together.

Data means the information a program works with. A room number, a book title and a student's mark are examples. Data can be a number, text or a group of related details.

A **data structure** is a way to arrange and keep that information so we can work with it. An array is one example you have already used in C.

An **algorithm** is a clear set of steps for solving a task. For the tasks in this book, those steps must produce the required result and finish after a finite amount of work for each allowed input.

Word	Think about this question	Our lab example
Data	What information do we have?	Which computers are free?
Data structure	How will we keep it?	An array containing one value per computer
Algorithm	What steps will we follow?	Visit each value and count the free computers
Program	How do we write those steps for a computer?	A complete C source file

An algorithm is the plan. A C program is one way to write that plan. The same plan can also be written in other languages. **You do not need C++ for this book.**

A familiar situation: finding a seat in a computer lab

A lab assistant needs to tell a group how many computers are available. We record 1 for a free computer and 0 for a busy computer. These meanings are our choice; C does not give these numbers a special lab meaning.

Suppose the five values are 1, 0, 1, 1 and 0. We should report three free computers. Keeping the values is one job. Counting the values equal to 1 is another job.

See the idea



An index tells us which seat value we are checking. Start the count at zero.

An **element** is one value in an array. An **index** is its position number. C starts these position numbers at zero. Five elements therefore have indexes 0 through 4.

First write the steps in ordinary words

1. Start the free-computer count at zero.
2. Look at the first computer's value.
3. If the value is 1, increase the count by one.
4. Move to the next computer and repeat until every value has been checked.
5. Report the count.

The stopping rule matters. Without moving to the next position, we could keep checking the same computer forever. Without starting at zero, we would not have a reliable count.

Complete program: count the free computers

The sample values are written inside the program so that you can focus on the steps. The program counts values equal to 1; this example assumes every status is either 0 or 1.

lab_seats.c

```c
#include <stdio.h>

int main(void)
{
    int status[] = {1, 0, 1, 1, 0};
    int free_count = 0;

    for (int i = 0; i < 5; i++)
    {
        if (status[i] == 1)
        {
            free_count++;
        }
    }

    printf("Free computers: %d\n", free_count);
    return 0;
}
```

Output

```
Free computers: 3
```

Line or part	What it does
int status[]	Creates the array; its starting list supplies five elements
free_count = 0	Starts with no free computers counted yet
int i = 0	Starts at the first array position
i < 5	Keeps the position inside the five-element array
status[i] == 1	Asks whether this computer is free
free_count++	Adds one to the count
i++	Moves to the next position
printf	Shows the final result

Save the file as `lab_seats.c`. In a terminal opened in that folder, build it with this command. These commands belong in the terminal, not inside the C file.

Terminal command

```
gcc -std=c17 -Wall -Wextra -Wpedantic lab_seats.c -o lab_seats
```

After a successful build, use `.\lab_seats.exe` in Windows PowerShell, or `./lab_seats` on Linux or macOS. If GCC is not set up yet, return to the C tutorial's setup chapter. Run these examples with your installed C compiler.

Follow the count without a computer

Position checked	Value	Does the count increase?	Count afterwards
0	1	Yes	1
1	0	No	1
2	1	Yes	2
3	1	Yes	3
4	0	No	3

Following a program with a small example is called a **dry run**. It helps us check the idea before relying on the output. Chapter 4 will teach this in more detail.

Operations: the jobs we do with stored data

An **operation** means a job performed on data. Learn the job before worrying about its name.

Operation	Simple meaning	Example
Access	Read a chosen item	Read the status at position 2
Traverse	Visit the items one by one	Check all lab computers
Search	Look for an item or condition	Find the first free computer
Insert	Add an item	Add a new booking to a list
Delete	Remove an item	Remove a cancelled booking
Update	Change an existing item	Mark a computer as busy
Sort	Arrange items in an order	Arrange booking numbers from small to large

Different arrangements make different jobs easier. A queue helps model people being served in arrival order. A stack helps model undoing the most recent action first. We will build each one later; you do not need their code today.

A small map of the structures we will learn

Structure	First picture to keep in mind	A familiar use
Array	Values in numbered positions	One attendance value per student
Linked list	Records connected by links to other records	A sequence that can be changed by updating links
Stack	Work with the most recently added item first	Undo the latest edit
Queue	Work with the earliest waiting item first	Serve students in arrival order
Tree	Parent and child relationships	Show a hierarchy of folders
Graph	Items connected in a network	Represent campus places and the paths between them

An arrangement with one sequence is usually called **linear**. An arrangement with branching or network relationships is called **non-linear**. Those names describe the relationships between elements, not a promise about where every byte is physically stored.

For now, recognise the purpose of each picture. Later chapters will explain its operations, code, memory rules, practice problems and interview questions in detail. A linked list is not automatically faster than an array; the required operation and its conditions matter.

Why learn this for real work and interviews?

A working result is the first requirement. We also need to ask what happens with more data, unusual data or limited memory. A plan that feels quick with five items may do too much repeated work with thousands.

An interviewer may ask you to explain the input, solve a small example, choose a structure, write code and discuss its limits. Practising those steps helps more than remembering a finished answer.

You are allowed to pause and draw the values. Understanding a small example is progress. You do not need to recognise every algorithm name on your first day.

Solved problem: count the busy computers

The question has changed: count busy computers instead of free ones. Our meanings are still 1 for free and 0 for busy. Keep the same visit-each-element plan and change the condition being counted.

busy_computers.c

```c
#include <stdio.h>

int main(void)
{
    int status[] = {0, 1, 0, 0};
    int busy_count = 0;

    for (int i = 0; i < 4; i++)
    {
        if (status[i] == 0)
        {
            busy_count++;
        }
    }
    printf("Busy computers: %d\n", busy_count);
    return 0;
}
```

Output

```
Busy computers: 3
```

The zeros are at positions 0, 2 and 3. We count each matching position, not just each different value. The loop limit is four because this example has four elements.

Common mistakes and useful tips

- **Confusing a value with a position:** status[1] means the second element, not a search for the value 1.
- **Using = inside the condition:** = assigns a value; == compares values. We want a comparison here.
- **Reading status[5]:** it is outside the five-element first example. The last valid index is 4.
- **Changing the data but forgetting its count:** Chapter 2 shows how to calculate the count from an actual array.
- **Choosing a complicated structure too early:** first ask what jobs the program must support.
- **Running an old executable after a failed build:** fix the reported problem and build again before trusting the result.

Practice: try before reading an answer

1. For statuses 0, 1, 0, 1, 1 and 1, how many computers are free? Write the count after each position.

2. In a list of library book numbers, identify the data, the structure and an algorithm for finding a chosen book number.

3. What should the free-computer count be if every computer is busy? What if every computer is free?

4. Change busy_computers.c so that all four statuses are 1. Predict the output, then run it.

Answer: Check the practice answers

1. Four are free. The running counts are 0, 1, 1, 2, 3 and 4.

2. The book numbers are the data. An array is one possible structure. Checking the numbers in order until a match is found is one possible search algorithm.

3. All busy gives zero. All free gives the number of computers. These examples help check whether our initial count and comparison are correct.

4. The output becomes Busy computers: 0. None of the four values equals 0, so busy_count never increases.

Interview questions and natural answers

Answer: What is the difference between a data structure and an algorithm?

A data structure is how I keep the information. An algorithm is the steps I follow to do a job with it. For example, I can store computer statuses in an array and use a loop to count free computers.

Answer: Can an algorithm be written without C code?

Yes. I can explain the steps in ordinary words, write pseudocode or draw a flowchart. C code is one implementation of those steps.

Answer: Is the shortest program always the best solution?

No. I first check whether it solves the full question correctly. Then I compare clarity, running work, memory use and the allowed inputs. Fewer source lines do not prove that a solution does less work.

Quick revision

- Data is the information we work with.

- A data structure arranges and keeps that information.

- An algorithm gives clear steps for the required job.

- Check the steps with a small example before trusting the program.

- In this book, C is the language we use to practise those ideas.

02. C revision: the tools we need for DSA

Before building new data structures, let us practise the C ideas we will use repeatedly. This is a revision chapter, not a replacement for the full C course. Read one section, run its small example and explain what changed.

Check these skills before moving ahead

You should be able to save a .c file, build and run it, use if and loops, and write a small function. If those steps feel unfamiliar, use the C tutorial first. There is no benefit in hiding a C difficulty under a new DSA name.

C tool	Why it matters in DSA
Arrays	Keep several values and visit positions safely
Loops	Repeat a step for the needed values
Functions	Give each small job a clear name
Structures	Keep related fields in one record
Pointers	Refer to an existing value or a linked record
Dynamic memory	Request storage while the program is running

Arrays: keep the position and the count separate

An array has elements of one element type. `int seats[] = {8, 4, 6};` contains three int elements. Its valid indexes are 0, 1 and 2. The number 3 is the count, not a valid index.

Inside the scope of this actual array, `sizeof seats` gives its size in C bytes. `sizeof seats[0]` gives the size of one element. Dividing those sizes gives its element count. We do not need to assume that int occupies four bytes.

`sizeof` produces a value of type **size_t**. This is an unsigned integer type used for sizes and counts. Use **%zu** to print it. Unsigned values cannot represent negative numbers; avoid subtracting one from a zero count.

array_positions.c

```c
#include <stdio.h>

int main(void)
{
    int seats[] = {8, 4, 6};
    size_t count = sizeof seats / sizeof seats[0];

    for (size_t i = 0; i < count; i++)
    {
        printf("Position %zu: %d seats\n", i, seats[i]);
    }
    printf("Element count: %zu\n", count);
    return 0;
}
```

Output

```
Position 0: 8 seats
Position 1: 4 seats
Position 2: 6 seats
Element count: 3
```

The i < count condition keeps each visit inside the array. **A C array does not automatically reject an invalid index for you.** Reading past its end is not a way to get a useful extra value.

Functions: give one small job a name

A function groups instructions for a job. The values supplied in a call are **arguments**. The names receiving them in the function definition are **parameters**. A returned value carries a result back to the caller.

In the next example, count_open receives access to the elements and a separate count. `const int values[]` means the function will not change those elements through values. The return type size_t suits a count.

count_open.c

```c
#include <stdio.h>

size_t count_open(const int values[], size_t count)
{
    size_t open = 0;
    for (size_t i = 0; i < count; i++)
    {
        if (values[i] == 1)
        {
            open++;
        }
    }
    return open;
}

int main(void)
{
    int status[] = {1, 0, 0, 1};
    size_t count = sizeof status / sizeof status[0];
    printf("Open: %zu\n", count_open(status, count));
    printf("Empty list: %zu\n", count_open(NULL, 0));
    return 0;
}
```

Output

```
Open: 2
Empty list: 0
```

With count zero, the loop body never runs, so this function does not read values. That is why this particular function permits NULL with a zero count. With a positive count, the caller must supply access to at least that many valid int elements.

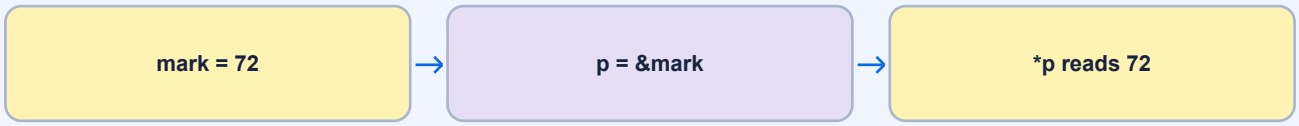
In a function parameter, the array notation is adjusted to a pointer. **It does not bring the array's length along.** Do not use sizeof values / sizeof values[0] inside count_open to discover the caller's length. Pass a truthful count instead. A definition placed before main also declares the function; it does not need a second separate prototype here.

Pointers: work with an existing value

A pointer can hold the address of an object. In these examples, an **object** means a stored C value, not an object-oriented class.

`&mark` asks for mark's address. `int *place` declares a pointer to int. Once place holds that valid address, `*place` accesses the int there. In the declaration, * is part of declaring a pointer; in *place = 76, it is used to access the pointed-to value.

See the idea



The pointer keeps an address. Following that address lets us read or change the original value.

update_mark.c

```c
#include <stdio.h>

void set_mark(int *place, int new_mark) { *place = new_mark; }

int main(void)
{
    int mark = 62;
    set_mark(&mark, 76);
    printf("Updated mark: %d\n", mark);
    return 0;
}
```

Output

```
Updated mark: 76
```

The function receives a copy of the address. That address still refers to main's mark, so writing through it changes main's mark. **C passes arguments by value**, including pointer values. This does not mean that a pointed-to object cannot be changed.

This function requires a pointer to a writable, live int. It does not accept NULL. A **live** value is one whose storage is still valid to use. Do not return the address of a local variable and expect that variable to remain alive after its function returns.

Structures: keep related details together

A structure groups named fields. For a lab booking, the room number and number of reserved seats belong to the same record. The dot selects a field of a structure value. The arrow selects a field through a structure pointer.

booking_record.c

```c
#include <stdio.h>

struct Booking
{
    int room;
    int reserved;
};

void add_one_seat(struct Booking *booking) { booking->reserved++; }

int main(void)
{
    struct Booking lab = {204, 3};
    add_one_seat(&lab);
    printf("Room %d: %d reserved seats\n", lab.room, lab.reserved);
    return 0;
}
```

Output

```
Room 204: 4 reserved seats
```

The example increases 3 to 4, which fits in int. A real booking function would also check the room's capacity. A structure groups the information; it does not automatically enforce that rule. Later, linked-list nodes will group a stored value with a link to another node.

Dynamic memory: request, check, fill and release

Sometimes a program needs to request storage while it is running. **malloc** requests a number of C bytes. It returns NULL if it cannot provide the requested storage. The returned storage does not already contain our chosen values.

Use stdlib.h for malloc and free. **free** releases an allocation that is no longer needed. The part of the program responsible for releasing it is called its **owner**. This book will say clearly who owns each allocation.

temporary_readings.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    size_t count = 3;
    int *readings = malloc(count * sizeof *readings);
    if (readings == NULL)
    {
        fprintf(stderr, "Could not reserve storage.\n");
        return EXIT_FAILURE;
    }

    readings[0] = 18;
    readings[1] = 21;
    readings[2] = 19;
    printf("Readings: %d %d %d\n", readings[0], readings[1], readings[2]);

    free(readings);
    readings = NULL;
    return 0;
}
```

Output

```
Readings: 18 21 19
```

That is the successful run. The failure branch reports a message on the error stream and stops without using the failed allocation. The count is a small fixed three here. If an outside input supplies a count, we must check that the requested byte calculation fits before multiplying it. The full C memory chapter explains that check.

`sizeof *readings` asks for the size of one pointed-to int; it does not read an uninitialised int in this expression. In C, a cast is not required for malloc's return value. After free, do not read or write the old allocation, and do not free it again. Setting this one pointer to NULL does not repair any other pointer that held the old address.

Solved problem: change two caller values

Swap two small marks. A **temporary variable** is a spare place used while we move values. Without it, overwriting the first mark would lose the value needed for the second assignment.

swap_marks.c

```c
#include <stdio.h>

void swap_marks(int *first, int *second)
{
    int saved = *first;
    *first = *second;
    *second = saved;
}

int main(void)
{
    int first = 52;
    int second = 87;
    swap_marks(&first, &second);
    printf("First: %d, second: %d\n", first, second);
    return 0;
}
```

Output

```
First: 87, second: 52
```

Both arguments must point to writable live int objects. Follow the temporary value: saved becomes 52, first becomes 87, then second becomes 52. The helper also leaves a single object unchanged if both arguments refer to that same object.

Common mistakes and tips

- Pass the element count, not the last index, to a function that asks for count.

- Keep i < count for visiting positions 0 through count - 1.

- Do not confuse an array with a pointer, even though an array argument usually supplies a pointer to its first element.

- Initialise ordinary local variables before reading them.

- Check an allocation before using it; fill an element before reading it.

- Name the owner and the cleanup step before building a dynamically allocated structure.

Practice questions

1. An array has six elements. What are the first and last valid indexes?

2. Change count_open's sample to 0, 0, 1, 0 and 1. Predict and check the returned count.

3. Why would set_mark(mark, 76) be wrong for the function shown here?

4. Trace swapping the values 9 and -2. What is saved after the first line of the helper?

5. In temporary_readings.c, where does the program stop if malloc returns NULL? Is there an allocation to free on that path?

6. Which change needs an extra rule: grouping room and reserved in a structure, or preventing a booking beyond the room's capacity?

Answer: Check the practice answers

1. The first is 0 and the last is 5. Six is the count, not a valid index.
2. It returns 2, because exactly two supplied values equal 1. Calculate the new count from the actual array in main.
3. The parameter needs an address. mark supplies an int value, while &mark supplies its address. With the function definition visible, the compiler should diagnose the wrong argument type.
4. saved is 9. Then the first value becomes -2 and the second becomes 9.
5. It reports the error and returns EXIT_FAILURE. That failed request did not produce an allocation for this pointer to release.
6. The structure groups the fields. Preventing an overbooking requires additional checking code; the structure definition alone does not do it.

Interview questions and answers

Answer: Does an array parameter know how many elements the caller has?

No. An array parameter is adjusted to a pointer parameter. I pass the length separately and keep it within the actual accessible elements.

Answer: If C passes by value, how can a function change my mark?

It receives a copy of a pointer value. Through that copied address it can reach the caller's mark, provided that mark is still alive and writable.

Answer: What is a memory leak?

It is allocated storage that has not been released and can no longer be reclaimed through the program's intended ownership paths. I avoid it by deciding who owns each allocation and keeping cleanup on every relevant exit path.

Answer: Is a pointer automatically safe because it is not NULL?

No. It might refer to released storage, an object whose lifetime ended or an unsuitable location. I need a valid target, an allowed access and a live object, not just a non-NULL value.

Quick revision

- An array's count and a valid index are different things.
- A function gives one job a name; describe its expected inputs and result.
- A pointer can reach an existing object; its target must remain valid.
- A structure groups fields but does not enforce application rules by itself.
- For requested storage, check the result, fill values, use them safely and release the allocation once.

03. Understand the problem before writing code

Many coding mistakes begin before the first line of code. We solve a slightly different question, miss a rule or assume that an example tells the whole story. This chapter teaches a repeatable way to understand a task before choosing an algorithm.

Start with a question you can explain

Suppose a college canteen offers meals at different prices. A student has a fixed amount to spend on one meal. We want to count how many listed meal options the student can afford.

Read that again: **count the options**, not add their prices or buy several meals together. The student chooses only one option. Two options can have the same price and still count as two listed options.

A useful first sentence is: Given the meal prices and a budget, return the number of listed prices that are less than or equal to the budget.

Input, output and rules

Input is what a solution receives. It can already be stored in an array; input does not always mean somebody typing into a keyboard. **Output** is the required result. A **constraint** is a rule or limit on the allowed inputs.

Part	Rule for this exercise
Prices	A list of non-negative whole-rupee prices that fit in int
Budget	A non-negative whole-rupee amount that fits in int
Count	The number of valid price elements supplied
Affordable	Price is less than or equal to budget
Repeated prices	Each listed option counts separately
Empty list	Return zero
Required result	Number of affordable options, not their total price
Change the input?	No; leave the original prices unchanged

In this lesson, the function expects the caller to meet these rules. It does not read and validate arbitrary typed text. Saying what a function expects is part of explaining it honestly.

Work through a normal example

For prices 35, 60, 45 and 45 with budget 45, the answer is 3. The 35-rupee meal qualifies. The 60-rupee meal does not. Both 45-rupee options qualify because equality is allowed.

Price	Compare with 45	Count afterwards
35	35 <= 45, yes	1
60	60 <= 45, no	1
45	45 <= 45, yes	2
45	45 <= 45, yes	3

Do not write <= merely because it appears in an answer you remember. It comes from the meaning of affordable in this question. If a different question says strictly below the budget, the comparison changes.

Ask about cases the first example does not show

A **boundary case** sits at an important limit, such as no elements, one element or a price exactly equal to the budget. An **edge case** is an unusual but allowed situation that can expose a mistake. People sometimes use these terms loosely; the useful habit is to identify the cases and their expected results.

Prices and budget	Expected result	What it checks
Empty list, budget 45	0	No element must be read
[45], budget 45	1	Equality belongs in the answer
[46], budget 45	0	No matching option
[20, 20], budget 20	2	Duplicates count as separate options
[0, 5], budget 0	1	A zero budget and a free option
[9, 2, 7], budget 10	3	Every option qualifies

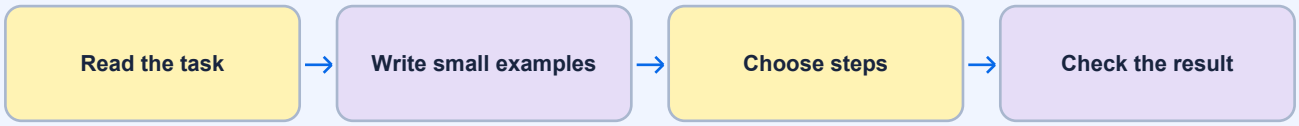
An **invalid input** breaks the stated rules, such as a negative price here. Do not silently mix invalid inputs with allowed edge cases. In an interview, ask whether the input is guaranteed valid or whether you must reject it.

Choose the smallest clear plan

We can visit each price once. Start a count at zero. Increase it whenever a price qualifies. Return the count after the final price.

This is a **direct approach**: it follows the definition of the question. Some people call a straightforward approach brute force. That label does not automatically mean the approach is bad. For this unsorted list, every price may affect the result, so a full scan is a sensible choice.

See the idea



Decide what the answer means before writing C code. Include a case where no answer exists.

Sorting is not needed. Sorting would change the arrangement and add work before answering this single counting question. We should not add an algorithm merely because we recently learned its name.

Complete program: count affordable options

affordable_options.c

```c
#include <stdio.h>

size_t count_affordable(const int prices[], size_t count, int budget)
{
    size_t affordable = 0;
    for (size_t i = 0; i < count; i++)
    {
        if (prices[i] <= budget)
        {
            affordable++;
        }
    }
    return affordable;
}

int main(void)
{
    int prices[] = {35, 60, 45, 45};
    size_t count = sizeof prices / sizeof prices[0];
    printf("Affordable options: %zu\n", count_affordable(prices, count, 45));
    printf("Empty list: %zu\n", count_affordable(NULL, 0, 45));
    return 0;
}
```

Output

```
Affordable options: 3
Empty list: 0
```

The parameter budget holds the spending limit. The const on prices prevents changing the prices through this parameter. The function returns a size_t count, which main prints with %zu. It counts matches; it does not add meal prices, so there is no accumulating price total to overflow in this solution.

For positive count, prices must provide access to at least that many valid int elements. When count is zero, no array access happens, so this implementation permits NULL for that empty case. Passing a larger count than the actual array is still a caller error.

A small test program checks the boundary cases

A **test case** pairs an input with the result you expect. Decide the expected result yourself before running the code. Otherwise you might simply accept a wrong result because it appeared on the screen.

affordable_cases.c

```c
#include <stdio.h>

size_t count_affordable(const int prices[], size_t count, int budget)
{
    size_t result = 0;
    for (size_t i = 0; i < count; i++)
    {
        if (prices[i] <= budget)
        {
            result++;
        }
    }
    return result;
}

int main(void)
{
    int equal[] = {45};
    int repeated[] = {20, 20};
    int zero_budget[] = {0, 5};
    int none[] = {46, 70};

    printf("Equal price: %zu\n", count_affordable(equal, 1, 45));
    printf("Repeated options: %zu\n", count_affordable(repeated, 2, 20));
    printf("Zero budget: %zu\n", count_affordable(zero_budget, 2, 0));
    printf("None affordable: %zu\n", count_affordable(none, 2, 45));
    printf("No options: %zu\n", count_affordable(NULL, 0, 45));
    return 0;
}
```

Output

```
Equal price: 1
Repeated options: 2
Zero budget: 1
None affordable: 0
No options: 0
```

These cases do not prove every possible input is correct, but each one checks a specific rule. Keeping that purpose beside a test makes it easier to understand a failure later.

Change the question, then change the condition

Now ask: How many listed options cost **more than** the budget? Keep the same visit-each-price structure, but count prices greater than budget. The answer for our first list is one, because only 60 is greater than 45.

expensive_options.c

```c
#include <stdio.h>

int main(void)
{
    int prices[] = {35, 60, 45, 45};
    size_t count = sizeof prices / sizeof prices[0];
    size_t expensive = 0;
    int budget = 45;

    for (size_t i = 0; i < count; i++)
    {
        if (prices[i] > budget)
        {
            expensive++;
        }
    }
    printf("Too expensive: %zu\n", expensive);
    return 0;
}
```

Output

```
Too expensive: 1
```

Notice how explaining the requirement makes the comparison almost choose itself. This is more reliable than editing operators until a sample happens to pass.

Common mistakes and interview tips

- Restate what the result means: a count, a value, a position and a total are different answers.

- Ask whether repeated values count separately and whether equality qualifies.

- Say whether you may change the input. Do not sort a caller's array without considering that rule.

- Include an empty case before reading the first element.

- A typed-input parser and the algorithm are different jobs. Keep their responsibilities clear.

- Do not promise a faster solution before checking whether a full input read is needed.

Before coding, write one sentence about the task and three test cases. That small pause often saves much more time than it takes.

Practice questions and answers

1. Prices are 12, 28, 28 and 50. The budget is 28. Give the affordable count and explain the duplicates.

2. A question asks for prices strictly below 28. Which comparison do you need, and what is the count for the same list?

3. Give a test that catches the mistake of using < instead of <= for the affordable task.

4. A student returns 125 for prices 35, 60, 45 and 45 with budget 45. What different question may that student have solved?

5. Would a list containing -10 be an allowed edge case under this chapter's rules?

6. Write a short task statement for counting marks greater than or equal to 40. State the allowed mark range and the empty-list result.

Answer: Check the practice answers

1. Three. The 12-rupee option and both 28-rupee options qualify. We count listed options, not distinct prices.

2. Use price < 28. Only 12 qualifies, so the count is one.

3. One option priced exactly at the budget is enough: [28] with budget 28 must return one.

4. They added the affordable prices: 35 + 45 + 45 = 125. That answers a different question. We need the number of affordable options, which is three, not their combined price.

5. No. Prices must be non-negative. It is invalid input for this task, and its handling must be specified separately.

6. One possible statement: Given marks from 0 through 100 and their count, return how many marks are at least 40. Count equal marks separately, leave the input unchanged and return zero for an empty list.

Interview questions

Answer: What do you ask before choosing an algorithm?

I clarify the input, required output, allowed sizes and values, duplicate handling, empty cases and whether I may change the input. Then I work through a small example.

Answer: Why is visiting every price reasonable here?

Each unsorted price can change the answer. If I skip a price without any other information, it could be an affordable option I failed to count. A single full scan gives a clear solution for this task.

Answer: Does passing the sample prove my solution is correct?

No. The sample is one case. I also check the rules with cases such as equality, empty input, duplicates, no matches and all matches, and explain why the approach works for the allowed inputs.

Quick revision

- Explain the required result before choosing code.

- Write the input rules and limits explicitly.

- Separate invalid inputs from unusual allowed cases.

- Test equality, duplicates, empty input and other relevant boundaries.

- Start with a clear correct plan before trying to improve it.

04. Write the steps, draw the flow and dry-run the result

Once you understand a task, you need a plan you can follow. We will write that plan without C syntax, draw its choices and test it by hand. Our example is finding the largest value, including when every value is negative.

What is pseudocode?

Pseudocode means writing a program's steps in a readable form without following all the rules of a programming language. You can use words such as IF, REPEAT and RETURN to make the choices clear.

There is no single compulsory spelling for pseudocode. The important part is that another person can follow it. Do not paste pseudocode into a C compiler and expect it to run.

The task is: given a list of integers, report its largest value. If the list is empty, report that there is no largest value to return. For a non-empty list, repeated largest values are allowed; we need the value, not every position where it occurs.

Start with a valid value from the list

Consider the changes in a club's daily seat availability: -8, -3, -11 and -3. A negative value means the number of available seats fell. The largest change is -3, since -3 is greater than -8 and -11.

Starting best at zero would be wrong. Zero is greater than all these values, but it is not an element of the list. Instead, after handling the empty case, start best with the first supplied element.

Walkthrough

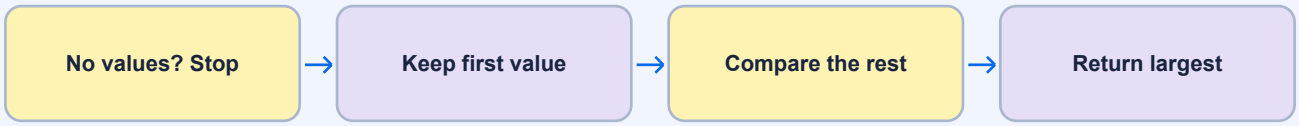
```
IF the list is empty
    REPORT no largest value
OTHERWISE
    best = the first value
    FOR each remaining value
        IF this value is greater than best
            best = this value
    REPORT best
```

The word best names the largest value seen **so far**. It does not promise to be the final answer before we finish checking the list.

A flowchart shows where the choices go

A **flowchart** draws steps and decisions with connecting arrows. A box represents an action. A decision asks a question and leads to different paths. Both comparison paths must eventually return to checking the next item.

See the idea



Use an existing value as the starting answer. Zero would give a wrong answer for all-negative input.

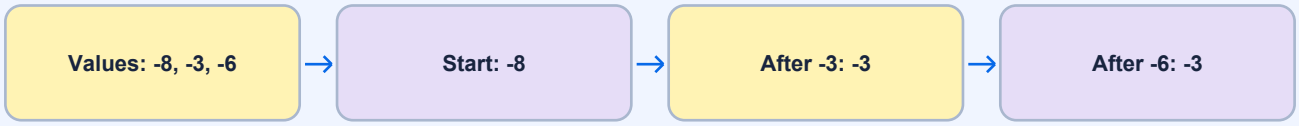
The empty path ends without reading the first element. The non-empty path starts with one real value. The repeated part advances to another position, so it eventually runs out of elements to check.

Dry run: keep a small table

Write the important values after each step. Do not change several rows in your head at once. The first value sets the starting best; it does not need a comparison with itself.

Step	Current value	Best before the step	Decision	Best afterwards
Start at index 0	-8	Not set yet	Use the first value	-8
Index 1	-3	-8	-3 is greater	-3
Index 2	-11	-3	-11 is not greater	-3
Index 3	-3	-3	Equal; no replacement needed	-3

See the idea



The best value changes only when a larger value is found.

Read the values one at a time. Predict whether best changes before checking the next value in your dry-run table. When the final value has been checked, there are no remaining candidates, so best is the answer.

Complete C program

We need to distinguish an empty list from a legitimate answer such as zero or -1. The helper therefore returns a **status**: 1 for a result and 0 for no result. On success, it writes the largest value through answer.

largest_change.c

```c
#include <stdio.h>

int find_largest(const int values[], size_t count, int *answer)
{
    if (count == 0 || values == NULL || answer == NULL)
    {
        return 0;
    }

    int best = values[0];
    for (size_t i = 1; i < count; i++)
    {
        if (values[i] > best)
        {
            best = values[i];
        }
    }
    *answer = best;
    return 1;
}

int main(void)
{
    int changes[] = {-8, -3, -11, -3};
    size_t count = sizeof changes / sizeof changes[0];
    int largest = 0;

    if (find_largest(changes, count, &largest))
    {
        printf("Largest change: %d\n", largest);
    }
    if (!find_largest(NULL, 0, &largest))
    {
        printf("Empty list: no largest value.\n");
    }
    return 0;
}
```

Output

```
Largest change: -3
Empty list: no largest value.
```

The early return happens before values[0]. For a successful call, values must refer to count accessible int elements, and answer must refer to a writable live int. The NULL checks do not prove that a non-NULL pointer or an exaggerated count is safe.

The loop starts at index 1 because index 0 supplied the starting best. The strict > comparison leaves best unchanged when a value is equal. That is fine because the task asks for the largest value only.

`*answer = best` writes the result only after the search is complete. When the helper returns 0, it leaves the previous answer unchanged; callers must check the status before treating the destination as a new result. The initial zero in main is not a candidate in the search.

Why should we believe the plan?

After reading the first element, best is the largest of the one value seen so far. Now suppose best is correct for the portion already checked.

When the next value is larger, replacing best makes it the largest of the extended portion. When the next value is smaller or equal, keeping best has the same effect. After the final step, the checked portion is the whole list.

This is a simple **correctness argument**: an explanation of why the steps produce the required result. The fact that a useful statement stays true after each loop step is called a **loop invariant**. Here that statement is: best is the largest value in the portion already checked.

You can understand the reasoning before remembering its formal name.

Boundary tests make the explanation stronger

List	Required result	Reason for the test
[]	No result	Do not read an absent first element
[12]	12	One element, no remaining comparisons
[-8, -3, -11]	-3	All values negative
[7, 7, 7]	7	Repeated equal values
[9, 2, 4]	9	First value already largest
[2, 4, 9]	9	Last value becomes largest

For n positive elements, this loop makes n - 1 comparisons between a remaining value and best. It also does other work, such as checking the loop condition. Chapter 5 will explain why counting one chosen kind of work can still help us describe growth.

Solved debugging exercise: a largest value below zero

A student starts best at zero for the input -5, -2 and -9. No value is greater than zero, so the program incorrectly keeps zero. The correction is to start from the first element after handling empty input.

largest_negative.c

```c
#include <stdio.h>

int main(void)
{
    int changes[] = {-5, -2, -9};
    size_t count = sizeof changes / sizeof changes[0];
    int best = changes[0];

    for (size_t i = 1; i < count; i++)
    {
        if (changes[i] > best)
        {
            best = changes[i];
        }
    }
    printf("Largest: %d\n", best);
    return 0;
}
```

Output

```
Largest: -2
```

This smaller program has a fixed, non-empty list. Unlike the earlier helper, it is not offering an empty-list interface. State that difference when explaining it.

Tips for useful dry runs

- Start with three or four values you can check yourself.

- Include an input that challenges your starting value, such as all-negative values.

- Write what each variable means, not just its current number.

- Follow the actual loop starting point and stopping condition.

- Keep the original input visible when the algorithm changes it.

- When an output differs, find the first step where your table and program disagree.

Practice questions

1. Dry-run the largest-value plan on 4, 9, 1 and 9. Show best after every element.

2. Write pseudocode for the smallest value. What comparison and starting value will you use?

3. Why is returning -1 alone not enough to represent an empty result for an unrestricted int list?

4. A loop uses i <= count. Which position does it visit after the last valid one?

5. Explain why the largest-value plan stops, using count and i.

6. Write the statement that should remain true while counting affordable options from Chapter 3.

Answer: Check the practice answers

1. The successive best values are 4, 9, 9 and 9. The final equal 9 does not need to replace it.
2. Reject or report empty input first. Start smallest with the first value. Visit the remaining values and replace smallest when a current value is less than it. Report smallest after the final visit.
3. -1 may be the actual largest value, for example in [-8, -1]. A separate status makes success and no result distinguishable.
4. It visits index count, which is outside an array containing count elements. Use i < count for the loop shown here.
5. i starts at 1 and increases once per visit. The list has a finite count. Once i reaches count, the condition is false and the loop ends.
6. The running count equals the number of affordable options in the portion already checked. Checking the next price either increases that correct count by one or leaves it unchanged.

Interview questions

Answer: What is the difference between pseudocode and a dry run?

Pseudocode describes the general steps. A dry run follows those steps with one specific input and records what changes.

Answer: How do you explain that your largest-value loop is correct?

I start from the first real value. After each visit, best is the largest among all values checked so far. Comparing the next value preserves that fact. When every element has been checked, best is the largest of the whole non-empty list.

Answer: Why return a status and write the answer separately?

The input can contain any int, so an ordinary number might be a real answer. A status lets the caller tell whether an answer was produced without reserving a valid data value as a failure marker.

Quick revision

- Pseudocode explains the plan without full C syntax.
- A flowchart makes actions, choices and repeated paths visible.
- A dry run tests the plan using a specific example.
- Handle empty input before using a first element.
- Explain what remains true after every step, then explain why the loop finishes.

05. Time complexity: how does the amount of work grow?

A program may feel quick with five values. What happens with fifty thousand? Time complexity helps us discuss that question without promising an exact number of seconds. We will begin by counting small pieces of work, then give the growth patterns their usual names.

Define n before counting anything

The letter **n** often means the size of the input. For counting affordable meals, n is the number of listed prices. For finding the largest array value, n is the number of elements.

Do not assume that n always means the same thing. A rectangular table may need two sizes: rows and columns. A graph later in the course will need counts of places and connections.

Time complexity describes how the running work grows as the chosen input size grows. We normally use a simple model where reading one array element, comparing ordinary machine-sized integers or updating a counter has a fixed cost.

Real running time also depends on the device, compiler, operating system and details of the program. Complexity does not replace measuring a real application; it gives us another way to reason about an approach.

Count one clearly named operation

For the largest-value loop from Chapter 4, let us count comparisons of a remaining value with best. The first value sets best; each of the other values needs one comparison.

Number of elements n	Comparisons with best
1	0
4	3
8	7
100	99

That is n - 1 such comparisons for a non-empty list. It is not a count of every machine instruction. The loop still performs indexing, updates and condition checks. Their costs follow the same overall linear growth here.

Name what you are counting. "There are n comparisons" and "the entire program takes exactly n steps" are different claims.

Big O: a useful upper bound on growth

Big O, written O(...), describes an upper bound on a cost's growth for sufficiently large inputs, allowing constant factors. We often use it to discuss worst-case time, but **Big O does not itself mean worst case**. You must say which cost or case you are describing.

For this book, we usually give the closest common growth bound that describes the work, rather than an unnecessarily loose one. For example, a linear scan is described as O(n), even though O(n squared) is also a valid but less useful upper bound for it.

You do not need a formal mathematical proof today. Start by asking: if n becomes larger, does the repeated work stay fixed, grow with n, grow with pairs of elements, or shrink the remaining problem each time?

O(1): work that stays bounded as n grows

Reading a known valid position in an array takes a fixed amount of work in our model. It does not require visiting every earlier position.

O(1) is read as **constant time**. It does not mean exactly one instruction or zero time. Ten fixed assignments are still a fixed amount of work, as long as their count does not grow with n.

Before reading position 2, you still need to know that at least three elements exist. Complexity is not permission to skip a safety check.

O(n): one scan and other linear patterns

When we visit n elements and do a bounded amount of work per element, the work grows in proportion to n. This is **linear time**, written O(n).

Two complete scans take about twice the repeated work of one scan, but both grow linearly. A scan doing three fixed jobs per element is also linear. Constants matter to actual speed; they do not change this growth category.

If one cost is 3n + 7, the n-dependent part eventually matters more than the fixed seven. We describe that growth as O(n). This is a summary of growth, not a claim that the seven operations disappeared.

O(n squared): one full scan inside another

Suppose each of n students is compared with every one of n student entries. The inner job runs n times for each outer visit, giving n times n, or n squared, visits. We write this as O(n^2), read "O of n squared."

For n = 4, n squared is 16. For n = 8, it is 64. Doubling n makes this count four times as large. This pattern is called **quadratic**, not exponential.

Nested loops are not automatically O(n^2). If the inner loop always runs three times, the count is 3n, which is linear. Look at each loop's actual bounds.

Complete program: compare four loop patterns

This demonstration counts visits rather than measuring seconds. It deliberately uses the small sizes 4 and 8, so all counters stay comfortably within size_t's range. It is not a safe counter for arbitrarily enormous products.

loop_work.c

```c
#include <stdio.h>

void show_work(size_t n)
{
    size_t one_scan = 0;
    size_t two_scans = 0;
    size_t all_pairs = 0;
    size_t three_per_item = 0;

    for (size_t i = 0; i < n; i++)
    {
        one_scan++;
    }
    for (size_t i = 0; i < n; i++)
    {
        two_scans++;
    }
    for (size_t i = 0; i < n; i++)
    {
        two_scans++;
    }
    for (size_t i = 0; i < n; i++)
    {
        for (size_t j = 0; j < n; j++)
        {
            all_pairs++;
        }
    }
    for (size_t i = 0; i < n; i++)
    {
        for (size_t j = 0; j < 3; j++)
        {
            three_per_item++;
        }
    }

    printf("n=%zu: one=%zu, two=%zu, pairs=%zu, fixed-inner=%zu\n", n, one_scan,
           two_scans, all_pairs, three_per_item);
}

int main(void)
{
    show_work(4);
    show_work(8);
    return 0;
}
```

Output

```
n=4: one=4, two=8, pairs=16, fixed-inner=12
n=8: one=8, two=16, pairs=64, fixed-inner=24
```

The loop lengths are the lesson: one_scan is linear, two_scans is linear, all_pairs is quadratic and three_per_item is linear. The sample program has fixed calls, but show_work demonstrates how these patterns would grow with its parameter n under the stated counting model.

O(log n): repeatedly make the remaining problem smaller

Logarithmic growth often appears when each step reduces the remaining problem by a fixed factor, such as half. Imagine narrowing a range from 16 candidates to 8, then 4, then 2 and finally 1. Only four halvings were needed.

The expression log2 n asks how many factors of two produce n. For powers of two, it also counts these repeated halvings to one. The log base does not change the Big O category when the base is a fixed number greater than one.

halving_steps.c

```c
#include <stdio.h>

size_t count_halvings(size_t remaining)
{
    size_t steps = 0;
    while (remaining > 1)
    {
        remaining /= 2;
        steps++;
    }
    return steps;
}

int main(void)
{
    printf("4 candidates: %zu halvings\n", count_halvings(4));
    printf("8 candidates: %zu halvings\n", count_halvings(8));
    printf("16 candidates: %zu halvings\n", count_halvings(16));
    return 0;
}
```

Output

```
4 candidates: 2 halvings
8 candidates: 3 halvings
16 candidates: 4 halvings
```

Integer division discards the fractional part. For 10, the values are 10, 5, 2 and 1, so there are three halvings. For 0 or 1, the loop makes zero visits. We describe the growing positive-input case as O(log n); near zero we handle the boundary separately.

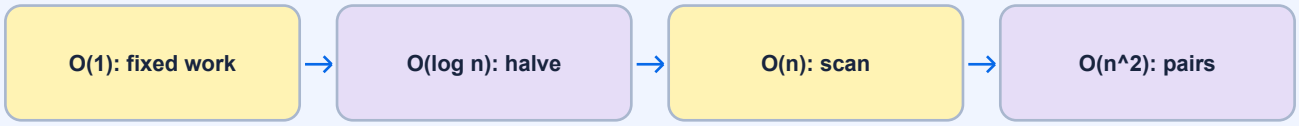
Halving alone is not a search algorithm. Binary search later uses an ordered array to decide which half can safely be discarded. You cannot discard half an arbitrary list without a reason.

Other growth names you will meet later

Pattern	Read it as	Idea to recognise
O(1)	Constant	A bounded amount of work
O(log n)	Logarithmic	Repeatedly reduce the remaining size by a fixed factor
O(n)	Linear	Work grows in proportion to the input size
O(n log n)	n log n	Roughly linear work across logarithmically many levels
O(n^2)	Quadratic	Work on all pairs in a full nested scan
O(2^n)	Exponential bound	A doubling pattern with each extra input item
O(n!)	Factorial bound	Multiplying the choices n, n-1, down to 1

These are growth patterns, not promises about every program with a matching name. Producing every subset or permutation also takes work to write each result; output size must be included in a full analysis. Later chapters will state the precise cost of the code being taught.

See the idea



These describe growth as the input gets larger. They are not measured running times.

The diagram compares n with n squared, using small counted sizes. It shows growth in work units, not seconds or benchmark results.

Two sizes and adding different stages

If one list has n values and a separate list has m values, scanning each once is O(n + m). Comparing every value of the first list with every value of the second is O(n times m). Do not rename both sizes n unless a relationship justifies it.

For a single input size, n work followed by n squared work is described as O(n^2): the faster-growing term dominates as n grows. But do not drop an independent variable simply because it appears after a plus sign.

Common mistakes and tips

- Big O is not a stopwatch, and O(1) is not "one second."

- Count loop iterations from their actual bounds, not from the number of for keywords.

- A fixed sample array helps run an example; analyse the general algorithm with its input size clearly defined.

- Do not call n squared exponential; its growth is polynomial.

- A better growth bound may still have higher costs for small inputs. Compare real requirements too.

- Do not ignore work hidden in function calls, printing long results or copying large arrays.

Practice questions

1. A loop visits each of n elements once and makes one comparison per visit. What growth describes these comparisons?

2. Two such loops run one after the other. What is their combined growth?

3. The outer loop runs n times and the inner loop always runs five times. Is the work quadratic?

4. One n-length loop contains another n-length loop. How many inner visits occur for n = 6?

5. Starting at 32, how many repeated integer halvings reach 1?

6. Why is O(n) a more informative description of a simple scan than O(n^2), even though both are upper bounds?

7. A rectangular matrix has r rows and c columns. What is the work of visiting every cell once?

8. Can you safely halve the candidate set in an unordered list merely because logarithmic growth sounds better?

Answer: Check the practice answers

1. Linear, O(n). The chosen comparison count is n.

2. O(n). The two scans make 2n visits; two is a constant factor.

3. No. Five visits per outer step gives 5n visits, which is O(n).

4. There are 36 inner visits. In general this full nested pattern is O(n^2).

5. Five: 32 to 16 to 8 to 4 to 2 to 1.

6. O(n) gives a closer growth bound. O(n^2) permits far more growth than the scan actually needs, so it hides useful information.

7. O(r times c). Each row contributes c cells and there are r rows.

8. No. A correct algorithm needs a reason that the discarded candidates cannot contain the required answer. Binary search gets that reason from ordering and comparisons.

Interview questions

Answer: Why do you not give the time complexity in milliseconds?

Milliseconds measure a particular run on a particular setup. Complexity describes how the cost grows with input size under a stated model. I can use both complexity reasoning and real measurements, but they answer different questions.

Answer: Does Big O always describe the worst case?

No. Big O is an upper-bound notation. I can apply it to a worst-case, average-case or another cost function, so I must identify which one I mean. Interviews commonly ask for worst-case bounds.

Answer: Are two nested loops always quadratic?

No. I inspect their bounds and the work in the body. An n-length outer loop with a fixed three-step inner loop is linear. Two independent variable lengths can give O(n times m).

Answer: Is an O(n) algorithm always faster than an O(n^2) algorithm?

Not for every small input or every implementation. Constants and actual operations matter. The linear growth is a useful advantage for sufficiently large inputs, but I also check the allowed sizes and real costs.

Quick revision

- Define what n counts before analysing a program.
- Name the operation you count and the model you use.
- Constant, logarithmic, linear and quadratic are growth descriptions.
- Sequential stages add work; full nested stages multiply their visit counts.
- Big O supplies an upper bound, not an exact runtime or an automatic choice of case.

06. Memory use, best and worst cases, and choosing an approach

Running work is only one part of a solution. We also need storage, and two inputs of the same size may require different amounts of searching. This chapter connects those ideas using small programs you can follow.

Space complexity asks how storage grows

Space complexity describes how memory needs grow with the input size. Think about the most storage needed at the same time, not the number of allocation calls made over the program's entire life.

There are different reporting conventions. Always say what you include.

Term	What we mean in this book
Input storage	Memory already holding the supplied values
Output storage	Memory holding the result we produce
Auxiliary space	Working storage besides the input and, when stated, the required output
Total space	Input, output and other working storage counted together

For an array of n int elements, the array storage grows linearly with n. Adding two counters does not change that total growth. But an algorithm using only those counters may need **O(1) auxiliary space**, because the extra working storage stays bounded.

O(1) space does not mean no memory. It means a bounded amount independent of the chosen growing input size, under our ordinary fixed-size C value model.

Reverse an array without creating another array

An **in-place** approach changes the original data rather than keeping a separate full result array. Here we reverse the array using one temporary int and one position counter.

Swap the first and last elements. Then swap the second and second-last. Stop after count / 2 swaps. An odd-length list leaves its middle element alone.

reverse_in_place.c

```c
#include <stdio.h>

void reverse_values(int values[], size_t count)
{
    for (size_t i = 0; i < count / 2; i++)
    {
        int saved = values[i];
        values[i] = values[count - 1 - i];
        values[count - 1 - i] = saved;
    }
}

int main(void)
{
    int values[] = {2, 7, 3, 9};
    size_t count = sizeof values / sizeof values[0];
    reverse_values(values, count);

    printf("Reversed:");
    for (size_t i = 0; i < count; i++)
    {
        printf(" %d", values[i]);
    }
    printf("\n");
    return 0;
}
```

Output

```
Reversed: 9 3 7 2
```

For count zero or one, the body does not run. In particular, count - 1 is inside that body, so it is not evaluated for the zero case. With positive count, the caller must supply that many writable elements.

There are count / 2 swaps, so the helper's time grows linearly. Its working variables occupy a bounded amount of storage: **O(n) time and O(1) auxiliary space**. If we include the input array too, total space is O(n).

Keep the original by building a reversed copy

Sometimes changing the input is not allowed. We can request room for a second array and write the values in reverse order there. This example uses four known input values; it is a small demonstration of allocation and copying, not an arbitrary-size input reader.

reverse_copy.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int original[] = {2, 7, 3, 9};
    size_t count = sizeof original / sizeof original[0];
    int *copy = malloc(count * sizeof *copy);
    if (copy == NULL)
    {
        fprintf(stderr, "Could not reserve copy storage.\n");
        return EXIT_FAILURE;
    }

    for (size_t i = 0; i < count; i++)
    {
        copy[i] = original[count - 1 - i];
    }
    printf("Original:");
    for (size_t i = 0; i < count; i++)
    {
        printf(" %d", original[i]);
    }
    printf("\nReversed copy:");
    for (size_t i = 0; i < count; i++)
    {
        printf(" %d", copy[i]);
    }
    printf("\n");
    free(copy);
    return 0;
}
```

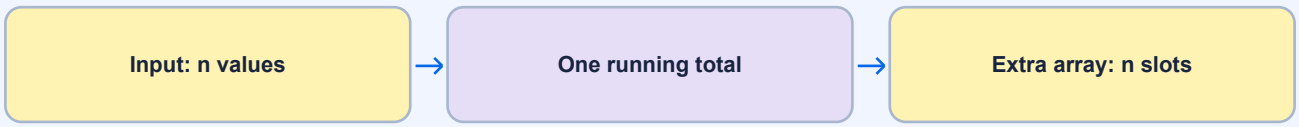
Output

```
Original: 2 7 3 9
Reversed copy: 9 3 7 2
```

This approach adds n int slots for the reversed result, so **extra storage including that result is O(n)**. If a question excludes required output storage when asking for auxiliary space, the other working storage is O(1). Explain the convention instead of arguing about the word space alone.

The count here comes from a small actual array. A general helper accepting a requested count must also check empty input and make sure the allocation-size multiplication fits before requesting storage. For a copied result, somebody must own and eventually release that allocation.

See the idea



One total uses O(1) extra space; a second n-element array uses O(n). State whether input storage is counted.

Neither choice is automatically better. The in-place version changes the original. The copied version preserves it but needs another array. Choose using the requirements and available memory.

Recursion can use storage even without malloc

A **recursive function** calls itself. A **base case** is the condition that stops making more recursive calls. Each unfinished call may need space for its own parameters and the place to continue after the next call finishes. This pending-call storage is usually called the **call stack**.

countdown_calls.c

```c
#include <stdio.h>

void countdown(unsigned remaining)
{
    if (remaining == 0)
    {
        return;
    }
    printf("%u\n", remaining);
    countdown(remaining - 1);
    printf("Finished call for %u\n", remaining);
}

int main(void)
{
    countdown(3);
    return 0;
}
```

Output

```
3
2
1
Finished call for 1
Finished call for 2
Finished call for 3
```

The call with 3 waits for the call with 2; that waits for 1; that reaches the base call with 0. The finishing lines then appear in the reverse order as the calls return. There are n + 1 calls along the deepest path for starting value n.

In the usual call-stack analysis, this pattern needs **O(n) auxiliary space**. Reusing the parameter name remaining does not mean all unfinished calls share one stored value. Do not assume that an unspecified compiler optimisation removes the pending calls. Large recursion depths can exceed available stack space.

This small printing example shows the call order. For a full runtime analysis of text generation, the amount of output also matters; we are counting calls here rather than promising exactly n machine instructions.

Best and worst cases: keep the input size the same

Now search an array by visiting elements in order and stopping at the first match. **Best case** means the least work among allowed inputs of a given size. **Worst case** means the most work among those inputs.

For a non-empty array, finding the target at the first position needs one equality comparison. Finding it at the last position, or not finding it at all, can require n equality comparisons.

search_cases.c

```c
#include <stdio.h>

struct SearchResult
{
    int found;
    size_t checks;
};

struct SearchResult search(const int values[], size_t count, int target)
{
    struct SearchResult result = {0, 0};
    for (size_t i = 0; i < count; i++)
    {
        result.checks++;
        if (values[i] == target)
        {
            result.found = 1;
            break;
        }
    }
    return result;
}

int main(void)
{
    int numbers[] = {14, 28, 37, 61};
    size_t count = sizeof numbers / sizeof numbers[0];
    int targets[] = {14, 61, 99};
    for (size_t i = 0; i < 3; i++)
    {
        struct SearchResult result = search(numbers, count, targets[i]);
        printf("Target %d: found=%d, checks=%zu\n", targets[i], result.found,
               result.checks);
    }
    struct SearchResult empty = search(NULL, 0, 14);
    printf("Empty: found=%d, checks=%zu\n", empty.found, empty.checks);
    return 0;
}
```

Output

```
Target 14: found=1, checks=1
Target 61: found=1, checks=4
Target 99: found=0, checks=4
Empty: found=0, checks=0
```

The found field tells us whether a match was found. The checks field counts only element-to-target equality comparisons. For positive count, the values parameter must refer to enough readable elements. For count zero, this implementation does not read it and permits NULL.

For the growing non-empty-input family, best-case time is O(1) and worst-case time is O(n). The search uses O(1) auxiliary space because the result fields and loop counter do not grow in number with the list size.

Average case needs assumptions

Average-case work means expected work under a stated way that inputs occur. It is not automatically the average of the best and worst numbers.

For a successful search in n distinct elements, suppose each target position is equally likely. The possible equality-comparison counts are 1 through n, so the average is (n + 1) / 2. That grows linearly. For four elements, the average is 2.5 comparisons; an average need not be a whole number.

If targets are usually at the front, or are usually absent, the distribution and average change. Without a distribution or useful assumptions, do not invent a precise average.

Upper, lower and tight bounds are not case names

You will meet these three notations in interview discussions. Think of them as ways to describe a growth function, not three separate kinds of inputs.

Notation	Plain meaning
Big O	An upper growth bound
Big Omega, Ω	A lower growth bound
Big Theta, Θ	Matching upper and lower growth bounds; a tight description

For the linear search shown, worst-case work grows linearly, so a tight worst-case description is Θ(n). Its best-case work for positive list sizes has a tight description Θ(1). **Omega does not mean best case, and Theta does not mean average case.** Any of these bounds must be attached to the cost function being discussed.

You can initially answer with clear words: first match needs one comparison; an absent target may need every element checked. Add the notation once that meaning is understood.

A practical comparison sheet

Approach	Work as the input grows	Storage beyond input	Important requirement
Reverse by swapping ends	O(n) time	O(1) working space	Input may be changed
Build a reversed copy	O(n) copying time	O(n) including copied result	Enough room for the result
Search one by one	O(n) worst-case time	O(1) working space	No sorted order required
Countdown with pending recursive calls	O(n) calls	O(n) usual stack space	A stopping case and manageable depth

When comparing algorithms, also check correctness, whether the input must stay unchanged, the work needed to prepare the data and what output must be produced. A good interview answer names the trade-off instead of calling every extra allocation bad.

Common mistakes and tips

- Freeing an n-element temporary array afterwards does not erase the peak O(n) storage used while it existed.
- No malloc calls does not imply O(1) space; local arrays and recursive calls also need storage.
- A loop that runs n times does not automatically need n variables at the same time.
- State whether space includes the input and the result.
- Best and worst cases compare different allowed inputs of the same size, not simply a small input against a large one.
- Always explain the assumptions behind an average-case claim.

Practice questions

1. A scan uses a supplied n-element array and three counters. What are its auxiliary and total space growths under our convention?
2. A function allocates another n-element array, uses it and frees it before returning. Does that make its peak extra storage O(1)?
3. Dry-run reversing 4, 8, 1, 7 and 9 in place. Which element stays in the middle?
4. Searching [14, 28, 37, 61] for 28 makes how many equality comparisons? What about 70?
5. Why can a recursive program need O(n) space even when every call uses only a few local values?
6. If a search target is equally likely to be at any of five distinct positions, and it is guaranteed present, what is the expected comparison count?
7. Is an in-place reverse suitable when the caller requires the original array to remain unchanged?
8. An interviewer says auxiliary space excludes the returned array. How should you report the reversed-copy approach?

Answer: Check the practice answers

1. O(1) auxiliary space because the counters have a fixed number of slots. Total space including the supplied array is O(n).
2. No. While the extra array existed, it held n elements. Freeing it is correct cleanup, but the peak extra storage including that array remains O(n).
3. Swap 4 with 9, then 8 with 7. The result is 9, 7, 1, 8 and 4. The middle 1 needs no swap.
4. Two for 28. Four for absent 70, because every element must be checked before this search can report absence.
5. Many calls can be unfinished at the same time. A linear chain of pending calls needs a number of call records growing with n in the usual stack model.

6. Three comparisons: (1 + 2 + 3 + 4 + 5) / 5 = 3. The stated successful, equally likely positions justify this average.

7. No, not directly. Use a separate result or obtain permission to change the input. Copying the input first also has a storage cost that must be counted.

8. State O(n) output storage for the copy and O(1) other working storage. The total extra storage including the result is O(n). Naming the convention prevents a misleading answer.

Interview questions

Answer: Can an algorithm use O(1) auxiliary space and O(n) total space?

Yes. A scan may use a supplied n-element array and only a few extra counters. The extra working storage is constant, while the array makes total storage linear.

Answer: What is a time-space trade-off?

It is a choice where changing the storage used can change the running work. For example, a later solution may keep previously computed answers to avoid repeating work. I must explain both the benefit and the added storage, rather than assuming more memory always makes a solution faster.

Answer: What is the worst-case time of this linear search, and why?

It is O(n). A missing target requires checking all n elements, and a target found at the last position also requires n equality comparisons. Each visit does a bounded amount of work under our model.

Answer: Is Big Omega another name for best-case complexity?

No. Omega is a lower-bound notation. Best case describes which inputs supply the least work for a given size. I can describe either a best-case or a worst-case cost using upper, lower or tight bounds.

Quick revision and what comes next

- Describe peak storage and say whether you include input and output.

- Constant auxiliary space can still use real memory.

- Recursion and local arrays count even without malloc.

- State which case you analyse and what assumptions support an average.

- Explain correctness and trade-offs alongside time and space.

These six chapters prepare the learning method. Next we will study array operations in detail, then build more structures and practise interview patterns. Do not rush into those names if tracing a small loop still feels difficult; repeat a small example and explain it in your own words.

ARRAYS AND PROBLEM-SOLVING PATTERNS

07. Arrays: positions, capacity and everyday operations

You have used arrays in small examples. Now we will treat an array as a place for a list that can change. We will read, search, update, insert and delete while keeping a clear boundary between stored items and spare room.

An array gives values numbered places

An array contains elements of one element type, stored next to one another in memory. Each element has the same size. A valid index tells C which element we mean; C does not need to scan earlier elements to reach it.

For int codes[6], the physical array contains six int elements. Their indexes are 0 through 5. The exact byte size of int depends on the implementation. We use sizeof when we need the actual size; we do not assume four bytes.

Reading a known valid element is **O(1)** work in our model. Finding a value whose position we do not know is a different job and may need a search.

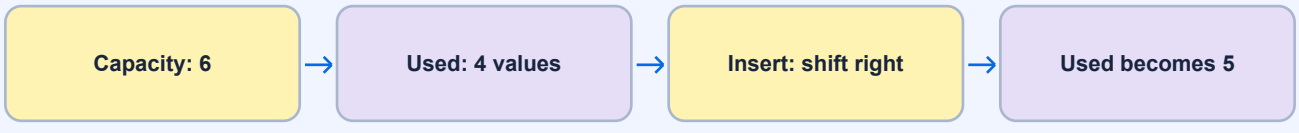
Capacity and used count are different

A college club keeps booking codes. The array has space for six codes, but only three bookings are currently in use.

Name	Meaning	Example
Capacity	Number of available element slots	6
Used count	Number of entries belonging to our current list	3
Used positions	The entries we may process as bookings	0, 1, 2
Spare positions	Storage not currently part of the booking list	3, 4, 5

The list must satisfy **0 <= count <= capacity**. Initialising spare slots to zero does not turn them into real bookings. After a deletion, old values may still remain in spare slots; count decides which entries belong to the list.

See the idea



Check the insertion position and free capacity before moving values. Used length and capacity are different.

An ordinary C array does not expand automatically. Inserting into spare room changes our used count, not the physical capacity. If the storage is full, our simple implementation rejects the insertion. A later dynamic-array implementation could request a larger allocation, with its own cost and cleanup rules.

Access, traversal and update

Access reads a chosen position. **Traversal** visits the used entries. **Update** replaces an entry already in use. An update does not create an extra booking, so the used count stays the same.

booking_access.c

```c
#include <stdio.h>

int main(void)
{
    int codes[6] = {310, 420, 530};
    size_t count = 3;
    size_t capacity = sizeof codes / sizeof codes[0];

    printf("Count: %zu, capacity: %zu\n", count, capacity);
    printf("Second booking: %d\n", codes[1]);
    codes[1] = 425;
    printf("Used bookings:");
    for (size_t i = 0; i < count; i++)
    {
        printf(" %d", codes[i]);
    }
    printf("\n");
    return 0;
}
```

Output

```
Count: 3, capacity: 6
Second booking: 420
Used bookings: 310 425 530
```

The loop ends at count, not capacity. Accessing codes[1] is safe here because count is three. A general update operation must first check position < count. Reading an uninitialised spare element would be an additional error, even if its index was inside the physical array.

Search: return a position only when found

For an unsorted list, visit entries from the beginning until the target is found. The helper below returns 1 for a match and writes its zero-based position. Otherwise it returns 0 and leaves the destination unchanged.

booking_search.c

```c
#include <stdio.h>

int find_booking(const int codes[], size_t count, int target, size_t *position)
{
    for (size_t i = 0; i < count; i++)
    {
        if (codes[i] == target)
        {
            *position = i;
            return 1;
        }
    }
    return 0;
}

int main(void)
{
    int codes[] = {310, 420, 530, 420};
    size_t position = 0;
    if (find_booking(codes, 4, 420, &position))
    {
        printf("First match at index %zu\n", position);
    }
    if (!find_booking(codes, 4, 999, &position))
    {
        printf("Booking not found.\n");
    }
    return 0;
}
```

Output

```
First match at index 1
Booking not found.
```

The contract requires a writable position destination and count readable elements when count is positive. A zero count performs no array reads. Repeated codes are possible in this teaching example; the function deliberately returns the first match, not all matches.

Insert at a position without losing existing values

To insert 375 at index 1 in [310, 420, 530], first make a space. Move 530 right, then move 420 right, then write 375 into the gap. Moving from the right prevents overwriting a value that we still need.

Step	Used entries plus the newly prepared slot
Start	[310, 420, 530]
Move index 2 to 3	[310, 420, 530, 530]
Move index 1 to 2	[310, 420, 420, 530]
Write new value at 1	[310, 375, 420, 530]

Valid insertion positions are **0 through count inclusive**. Position count means append at the end. We need count < capacity before writing the new slot.

booking_insert.c

```c
#include <stdio.h>

int insert_booking(int codes[], size_t capacity, size_t *count, size_t position,
                   int value)
{
    if (codes == NULL || count == NULL || *count > capacity || position > *count ||
        *count == capacity)
    {
        return 0;
    }
    for (size_t i = *count; i > position; i--)
    {
        codes[i] = codes[i - 1];
    }
    codes[position] = value;
    (*count)++;
    return 1;
}

int main(void)
{
    int codes[6] = {310, 420, 530};
    size_t count = 3;
    if (insert_booking(codes, 6, &count, 1, 375))
    {
        printf("After insertion:");
        for (size_t i = 0; i < count; i++)
        {
            printf(" %d", codes[i]);
        }
        printf("\nCount: %zu\n", count);
    }
    return 0;
}
```

Output

```
After insertion: 310 375 420 530
Count: 4
```

Here count is passed by address because the caller's count must change. The parentheses in (*count)++ mean increase the count value, not move the count pointer.

The caller must supply a truthful capacity backed by writable storage, a truthful count, and a separate count variable that is not an element of codes. NULL checks cannot prove that an exaggerated capacity is safe. Failed checks return before any booking or count changes.

Delete while preserving the remaining order

Deleting index 1 from [310, 375, 420, 530] moves 420 left into index 1 and 530 left into index 2. The used count becomes three. No allocation is being freed; the physical array is still present.

booking_delete.c

```c
#include <stdio.h>

int delete_booking(int codes[], size_t *count, size_t position)
{
    if (codes == NULL || count == NULL || position >= *count)
    {
        return 0;
    }
    for (size_t i = position; i + 1 < *count; i++)
    {
        codes[i] = codes[i + 1];
    }
    (*count)--;
    return 1;
}

int main(void)
{
    int codes[6] = {310, 375, 420, 530};
    size_t count = 4;
    if (delete_booking(codes, &count, 1))
    {
        printf("After deletion:");
        for (size_t i = 0; i < count; i++)
        {
            printf(" %d", codes[i]);
        }
        printf("\nCount: %zu\n", count);
    }
    return 0;
}
```

Output

```
After deletion: 310 420 530
Count: 3
```

This helper assumes the supplied count already fits the real writable array and that the count variable is separate. A valid deletion position must be below count. The old contents beyond the new count are ignored; clearing them is not what removes them from our logical list.

Compare the costs

Job on this fixed-capacity list	Time	Extra working space
Read or update a known valid index	O(1)	O(1)
Traverse all used entries	O(n)	O(1)
Search an unsorted list	O(n) worst case	O(1)
Append with spare capacity	O(1)	O(1)
Insert at the beginning, keeping order	O(n)	O(1)
Delete the last used entry	O(1)	O(1)
Delete at the beginning, keeping order	O(n)	O(1)

Here n is the used count. Copying a whole list into new storage is not the same as appending with spare room. If preserving order is unnecessary, replacing a deleted entry with the final used entry can avoid shifts; it also changes the order and must be allowed by the task.

Common mistakes and tips

- Do not confuse the value 420 with the position where 420 is stored.

- Insertion accepts position count; deletion does not.

- Shift right from the end for insertion; shift left from the deleted position for ordered deletion.

- Change the used count only after the operation is accepted.

- A zero in a spare slot is not automatically a valid list entry or a deletion marker.

- For several elements, keep a count rather than searching for a special end value that might be real data.

Practice questions

1. An array has capacity eight and used count five. Which indexes are used, and which are spare?

2. Insert 18 at index 0 in [25, 40, 60]. Show the shifts in their actual order.

3. Why would moving values from left to right lose information during that insertion?

4. Delete the last entry of a one-entry list. What count remains, and how many elements shift?

5. What should insertion do when count equals capacity? Should any existing entry change?

6. Search [8, 4, 8, 2] for 8 with the helper's first-match rule. What position is returned?

7. Is assigning zero to an entry the same as deleting it? Explain with a list where zero is allowed.

Answer: Check the practice answers

1. Used indexes are 0 through 4. Spare indexes are 5 through 7. Eight is the physical capacity, not a valid index.
2. Move 60 from 2 to 3, 40 from 1 to 2 and 25 from 0 to 1. Then write 18 at 0, giving [18, 25, 40, 60]. This requires one spare slot.
3. Writing index 1 from index 0 first overwrites the 40 that the next move still needs. Right-to-left shifting preserves those source values.
4. Count becomes zero and no elements shift. The storage remains available for later insertions.
5. Reject it in this implementation. Neither the used entries nor count should change on that failure path.
6. Index 0. The function stops at the first match even though another 8 exists.
7. No. It is an update to zero while the entry still belongs to the list. Deletion changes list membership and therefore the used count.

Interview questions

Answer: Why is array access constant time but searching can be linear?

A known index identifies the element directly. Searching asks which position, if any, contains a value. Without useful ordering or another index, I may need to inspect every used element.

Answer: Is array insertion always O(n)?

No. Inserting into the middle while preserving order may shift many entries. Appending at the used end of a fixed-capacity buffer with spare room is O(1). Allocating and copying into larger storage is a separate cost.

Answer: Does deleting an element shrink a C array?

Not in this implementation. I shift entries if needed and reduce the used count. The original array keeps its physical capacity.

Answer: What must be true before your insertion helper runs?

The capacity must describe real writable storage, the count must be truthful and within it, and the count variable must remain separate. The function checks its position and spare room before changing data, but it cannot verify an invented allocation size.

Quick revision

- Capacity is available storage; count is current list membership.
- Process only used entries.
- Direct access, searching and shifting are different operations with different costs.
- Preserve source values by choosing the correct shift direction.
- Handle empty lists, full lists and invalid positions before changing anything.

08. Minimum, maximum, second largest and simple counting

A small scan can answer many useful questions: which value is lowest, which is highest, what is the next distinct value below the highest, and how often does a value occur? We will choose a clear meaning for each result before writing its loop.

Start minimum and maximum with real data

For a non-empty list, start both candidates with the first element. Compare the remaining values and replace the relevant candidate when needed. Starting the largest at zero fails for an all-negative list. Starting the smallest at zero can fail for an all-positive list.

We will find both values together. A structure keeps the two results in one named group. A returned status distinguishes an empty input from legitimate values such as zero or -1.

value_limits.c

```c
#include <stdio.h>

struct Limits
{
    int smallest;
    int largest;
};

int find_limits(const int values[], size_t count, struct Limits *answer)
{
    if (count == 0 || values == NULL || answer == NULL)
    {
        return 0;
    }
    struct Limits result = {values[0], values[0]};
    for (size_t i = 1; i < count; i++)
    {
        if (values[i] < result.smallest)
        {
            result.smallest = values[i];
        }
        if (values[i] > result.largest)
        {
            result.largest = values[i];
        }
    }
    *answer = result;
    return 1;
}

int main(void)
{
    int changes[] = {-12, -5, -12, 3, 3};
    struct Limits answer;
    if (find_limits(changes, 5, &answer))
    {
        printf("Minimum: %d\nMaximum: %d\n", answer.smallest, answer.largest);
    }
    return 0;
}
```

Output

```
Minimum: -12
Maximum: 3
```

After each visit, the two fields describe the portion already checked. The caller supplies count readable elements and a writable result object. On failure, the destination is unchanged and must not be treated as a newly produced result.

This version makes two comparisons for each remaining element. It is still O(n) time and O(1) auxiliary space. Finding both with two separate scans would also be O(n), but the exact repeated work would differ.

What does second largest mean?

For [12, 12, 7, 9], the second entry in descending order is 12. But the **second distinct largest value** is 9: the greatest value strictly below 12. These are different questions.

We will use the distinct meaning. An empty list, a one-element list or a list containing only equal values has no second distinct largest value. We will report that through a status rather than inventing a numeric answer.

A simple two-pass solution

First find the largest value. Then scan again and consider only values strictly smaller than it. Among those candidates, keep the greatest.

The helper uses have_second to say whether a candidate exists. The initial zero in second is not a real candidate until the flag becomes true. This avoids reserving a valid number, such as INT_MIN, to mean "not found."

second_distinct.c

```c
#include <stdio.h>

int second_distinct(const int values[], size_t count, int *answer)
{
    if (count < 2 || values == NULL || answer == NULL)
    {
        return 0;
    }
    int largest = values[0];
    for (size_t i = 1; i < count; i++)
    {
        if (values[i] > largest)
        {
            largest = values[i];
        }
    }

    int second = 0;
    int have_second = 0;
    for (size_t i = 0; i < count; i++)
    {
        if (values[i] < largest && (!have_second || values[i] > second))
        {
            second = values[i];
            have_second = 1;
        }
    }
    if (!have_second)
    {
        return 0;
    }
    *answer = second;
    return 1;
}

int main(void)
{
    int scores[] = {12, 12, 7, 9};
    int equal[] = {6, 6};
    int answer = 0;
    if (second_distinct(scores, 4, &answer))
    {
        printf("Second distinct largest: %d\n", answer);
    }
    if (!second_distinct(equal, 2, &answer))
    {
        printf("Equal list: no second distinct value.\n");
    }
    return 0;
}
```

Output

```
Second distinct largest: 9
Equal list: no second distinct value.
```

Two complete scans still give **O(n)** time. Only a few values and flags are stored, giving **O(1)** auxiliary space. The original array is unchanged. NULL checks do not replace the caller's responsibility to supply a truthful accessible count.

A one-pass version and its three cases

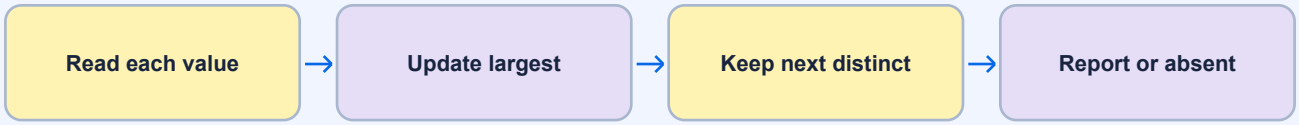
We can maintain both candidates as we go. A new value can be larger than the current largest, equal to it, or smaller.

- New largest: the old largest becomes the second candidate, then largest changes.

- Equal to largest: ignore it for the distinct result.

- Below largest: use it if there is no second candidate yet, or if it improves that candidate.

See the idea



A repeated largest value is not a second distinct value. Do not use zero as a missing-value marker.

second_one_pass.c

```c
#include <stdio.h>

int second_one_pass(const int values[], size_t count, int *answer)
{
    if (count < 2 || values == NULL || answer == NULL)
    {
        return 0;
    }
    int largest = values[0];
    int second = 0;
    int have_second = 0;

    for (size_t i = 1; i < count; i++)
    {
        int current = values[i];
        if (current > largest)
        {
            second = largest;
            have_second = 1;
            largest = current;
        }
        else if (current < largest && (!have_second || current > second))
        {
            second = current;
            have_second = 1;
        }
    }
    if (!have_second)
    {
        return 0;
    }
    *answer = second;
    return 1;
}

int main(void)
{
    int scores[] = {-5, 6, 6, 2, 9, 9};
    int answer = 0;
    if (second_one_pass(scores, 6, &answer))
    {
        printf("Second distinct largest: %d\n", answer);
    }
    return 0;
}
```

Output

```
Second distinct largest: 6
```

When 9 becomes the largest, the previous largest 6 must be saved before it is overwritten. Repeated 9 values do not displace the distinct second value. The one-pass version is also O(n) time and O(1) auxiliary space; it reduces the number of scans, not the growth category.

Count occurrences when the allowed range is small

A **frequency** is how often a value appears. If ratings are whole numbers from 0 through 5, six counters are enough. Counter 2 stores the number of ratings equal to 2.

Validate a rating before using it as a counter index. A negative value or a value above 5 must not index this six-element array.

rating_counts.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int ratings[] = {2, 5, 2, 0, 5, 2};
    size_t count = sizeof ratings / sizeof ratings[0];
    size_t frequency[6] = {0};

    for (size_t i = 0; i < count; i++)
    {
        if (ratings[i] < 0 || ratings[i] > 5)
        {
            fprintf(stderr, "Rating outside 0 through 5.\n");
            return EXIT_FAILURE;
        }
        frequency[ratings[i]]++;
    }
    for (size_t rating = 0; rating < 6; rating++)
    {
        printf("Rating %zu: %zu\n", rating, frequency[rating]);
    }
    return 0;
}
```

Output

```
Rating 0: 1
Rating 1: 0
Rating 2: 3
Rating 3: 0
Rating 4: 0
Rating 5: 2
```

The counters begin at zero. The sample's six counters sum to six because every valid input rating contributes once. Counting is O(n) time with O(1) extra space for this fixed six-value range. If the allowed range grows to R possible values, creating and reporting all counters costs O(n + R) time and O(R) storage.

Do not create an enormous counter array merely because values are integers. Large or sparse ranges may need another approach, such as sorting or a hash table, taught later.

Useful boundary cases

Input	Minimum/maximum	Second distinct largest
Empty	No result	No result
[4]	Both 4	No result
[4, 4]	Both 4	No result
[-8, -2, -5]	-8 and -2	-5
[9, 3, 9, 7]	3 and 9	7

The same caution applies to INT_MIN and INT_MAX: they are legitimate int values. Comparing them is fine; performing an overflowing subtraction to compare them is not needed here.

Common mistakes and tips

- Clarify whether second largest allows equal values before solving it.
- Use actual data to initialise minimum and maximum.
- Use a separate existence flag rather than treating a valid value as a missing-result marker.
- When updating two candidates, save the old largest before changing it.
- Keep value and counter index separate unless the range rule explicitly connects them.
- Check that frequency totals agree with the number of valid input items.

Practice questions

1. Find the minimum and maximum of [-7, -4, -9, -4].
2. What is the second distinct largest value in [15, 15, 11, 8]? What would "second entry after descending sorting" give?
3. Trace the one-pass version on [3, 1, 5, 5, 4]. Show largest and second after each element.
4. Why is [INT_MIN, INT_MAX] a useful test for a distinct-second helper?
5. Give the six frequency counts for [0, 2, 2, 5, 0].
6. Why must a rating of -1 be rejected before indexing frequency?
7. Two passes versus one pass: do they have different Big O growth here?

Answer: Check the practice answers

1. Minimum is -9 and maximum is -4. Zero is not a candidate in this list.
2. The distinct answer is 11. The second sorted entry is 15 because the largest occurs twice. Ask which meaning the task requires.

3. Start largest=3 with no second. After 1: largest=3, second=1. After 5: largest=5, second=3. The repeated 5 changes nothing. After 4: largest=5, second=4.

4. INT_MIN is the correct second value. A helper that uses INT_MIN alone as a "missing" marker can incorrectly reject a valid answer. A separate flag avoids that confusion.

5. Counts for ratings 0 through 5 are [2, 0, 2, 0, 0, 1]. Their sum is five.

6. Negative indexing is outside the array and invalid. The data rule must be checked before the value is used as an index.

7. Both are O(n) time and O(1) auxiliary space. Their exact work and the clarity of their bookkeeping differ.

Interview questions

Answer: Why not sort to find the second largest?

Sorting can solve some versions, but I must clarify duplicates and whether input order may change. For the distinct result, a scan with candidate tracking gives O(n) time and constant auxiliary space without sorting.

Answer: How do you handle a list where every value is equal?

It has a largest value but no strictly smaller second candidate. I return a no-result status rather than repeating the largest or inventing zero.

Answer: Why use a flag if second is initially zero?

The zero only initialises the C variable. The flag says whether it currently represents a real candidate. That lets negative values and every valid int participate correctly.

Answer: Is frequency counting always constant-space?

Only when the number of counters is fixed independently of the growing input. If the permitted value range has size R, I report O(R) counter storage and include its initialisation or output cost where relevant.

Quick revision

- Initialise extreme candidates from real input.

- Separate a missing answer from a valid numeric answer.

- Define duplicate handling for second-largest questions.

- One pass and two passes can have the same growth rate.

- A small, checked value range makes direct frequency counting useful.

09. Reverse, rotate and rearrange array values

Changing an array's order is a common interview task. First decide what kind of change is required. Reversing, rotating and moving selected values are different jobs, even when they all use the same original array.

Reverse and rotate do different things

Job	Start with [11, 22, 33, 44, 55]	Result
Reverse	Read the order backwards	[55, 44, 33, 22, 11]
Rotate left by one	Move the first item to the end	[22, 33, 44, 55, 11]
Rotate right by one	Move the last item to the front	[55, 11, 22, 33, 44]
Rotate left by two	Move the first two items to the end in order	[33, 44, 55, 11, 22]

Rotation wraps values around; it does not discard them. A shift that drops an endpoint value is a different operation. State the direction and number of positions before writing code.

Reverse by swapping from the ends

Chapter 6 reversed an array using one temporary value. Swap the first and last entries, then move inward. For n entries, only n / 2 swaps are needed. Reversing twice restores the original order.

We will reuse that idea below on a **range**: part of the array. We describe ranges as [first, end), meaning include first but stop before end. The end position is allowed to equal the array count because it is a boundary, not an element we read.

A clear first rotation: one position left

Save the first value before overwriting its slot. Move every following value one position left. Put the saved value at the final position.

rotate_one.c

```c
#include <stdio.h>

void rotate_left_one(int values[], size_t count)
{
    if (count < 2)
    {
        return;
    }
    int saved = values[0];
    for (size_t i = 1; i < count; i++)
    {
        values[i - 1] = values[i];
    }
    values[count - 1] = saved;
}

int main(void)
{
    int turns[] = {201, 202, 203, 204};
    rotate_left_one(turns, 4);
    printf("Next order:");
    for (size_t i = 0; i < 4; i++)
    {
        printf(" %d", turns[i]);
    }
    printf("\n");
    return 0;
}
```

Output

```
Next order: 202 203 204 201
```

This can model rotating which club member gets the first turn. A one-entry list stays unchanged. A zero count returns before reading any element. Otherwise the caller provides count writable int elements.

This is O(n) time and O(1) auxiliary space. Repeating it k times costs O(n times k). It is an understandable baseline, but there is a better approach when k is large.

Reduce unnecessary full rotations

Rotating a five-entry list left five times brings it back to the start. Rotating left seven times therefore has the same effect as rotating left twice.

For non-empty input, use **k %= count** to retain only the useful remainder. Handle count zero first, because taking a remainder with divisor zero is invalid. k is a non-negative size_t in our interface; negative rotation requests are not part of its contract.

An alternative solution could copy each value into a new result array. That uses O(n) extra result storage. We will instead use three reversals to rotate in place.

Rotate left by k using three reversals

Split the list into block A, the first k elements, and block B, the rest. We want A followed by B to become B followed by A, without reversing the order inside the finished blocks.

1. Reverse A.
2. Reverse B.
3. Reverse the whole list.

See the idea



A left rotation keeps the order inside both blocks. Reduce the shift modulo n only after checking n is nonzero.

For [11, 22, 33, 44, 55] and k = 2, the stages are [22, 11, 33, 44, 55], then [22, 11, 55, 44, 33], then [33, 44, 55, 11, 22]. The final reversal restores each block's internal order while swapping the blocks' places.

rotate_by_count.c

```c
#include <stdio.h>

void reverse_range(int values[], size_t first, size_t end)
{
    while (end - first > 1)
    {
        end--;
        int saved = values[first];
        values[first] = values[end];
        values[end] = saved;
        first++;
    }
}

void rotate_left(int values[], size_t count, size_t k)
{
    if (count < 2)
    {
        return;
    }
    k %= count;
    if (k == 0)
    {
        return;
    }
    reverse_range(values, 0, k);
    reverse_range(values, k, count);
    reverse_range(values, 0, count);
}

int main(void)
{
    int values[] = {11, 22, 33, 44, 55};
    rotate_left(values, 5, 2);
    printf("Left by two:");
    for (size_t i = 0; i < 5; i++)
    {
        printf(" %d", values[i]);
    }
    printf("\n");
    return 0;
}
```

Output

```
Left by two: 33 44 55 11 22
```

The range helper requires 0 <= first <= end <= count for the actual writable array. The rotation helper calls it only with valid boundaries. end - first counts the remaining range length. Decrementing end happens only when at least two entries remain, so the helper never reads the excluded boundary.

For a non-trivial rotation, the three reversals together do O(n) work, not O(n times k). They reuse a constant amount of storage. The zero-remainder case can return immediately. The algorithm changes the original array; ask permission if input order must be preserved.

Move zeros to the end while keeping the other order

Now suppose zero means an unused reading. We want the non-zero readings first, in their original relative order, followed by all the zeros. This is a **stable** rearrangement: the items we keep preserve their relative order.

Use read to inspect each item and write to mark the next kept position. Copy only non-zero values into that prefix. After the scan, fill the remaining slots with zero.

move_empty_readings.c

```c
#include <stdio.h>

void move_zeros(int values[], size_t count)
{
    size_t write = 0;
    for (size_t read = 0; read < count; read++)
    {
        if (values[read] != 0)
        {
            values[write] = values[read];
            write++;
        }
    }
    while (write < count)
    {
        values[write] = 0;
        write++;
    }
}

int main(void)
{
    int readings[] = {0, 7, 0, 4, 7, 0};
    move_zeros(readings, 6);
    printf("Rearranged:");
    for (size_t i = 0; i < 6; i++)
    {
        printf(" %d", readings[i]);
    }
    printf("\n");
    return 0;
}
```

Output

```
Rearranged: 7 4 7 0 0 0
```

write never moves ahead of read during the copying scan. Therefore a write cannot destroy an unread future element. The two 7 values remain on opposite sides of 4 in the same relative order. We did not sort the values.

There is no allocation and no change to the element count. Both the scan and the final zero fill are linear, so the total is O(n) time and O(1) auxiliary space. Empty input makes neither loop read an element. With positive count, provide that many writable elements.

Choose tests from the movement rules

Situation	Expected behaviour
Empty list	No reads and no remainder by zero
One element	Rotation and reversal leave it unchanged
k = 0 or a multiple of n	Rotation leaves the order unchanged
k larger than n	Reduce it using the remainder
Repeated values	Keep the correct number of occurrences
All zeros	Zero-moving result is still all zeros
No zeros	Zero-moving result preserves the original order

To check a rotation, compare the whole resulting sequence, not just its first element. Also check that no value was lost or duplicated. A wrong shift can accidentally make several entries equal and still look plausible at the front.

Common mistakes and tips

- Reversal changes every position's direction; rotation wraps a block around.

- Save a value before a shift overwrites its only remaining copy.

- Handle zero count before remainder calculations or count - 1 expressions.

- Write the rotation direction beside a test case.

- Stability is a requirement, not an automatic property of every rearrangement.

- Do not repeatedly delete zeros with shifts if a single read/write pass will do.

Practice questions

1. Reverse [4, 8, 12, 16]. Then rotate the original left by one. Are the results equal?

2. Rotate [10, 20, 30, 40] left by six. Show the reduced k and the result.

3. Show all three reversal stages for rotating [2, 4, 6, 8, 10] left by three.

4. Move zeros to the end of [0, -2, 0, 5, -2]. Preserve the non-zero order.

5. Why should zero-count handling happen before k %= count?

6. A zero-moving method swaps each front zero with the final non-zero. Is that guaranteed stable?

7. How could you express a right rotation by r using the left-rotation helper for a non-empty list?

Answer: Check the practice answers

1. Reversal gives [16, 12, 8, 4]. A left rotation gives [8, 12, 16, 4]. They are different operations.

2. 6 % 4 is 2, so the result is [30, 40, 10, 20].

3. Reverse the first three: [6, 4, 2, 8, 10]. Reverse the rest: [6, 4, 2, 10, 8]. Reverse everything: [8, 10, 2, 4, 6].

4. [-2, 5, -2, 0, 0]. The relative order of the kept values stays the same.

5. A divisor of zero is invalid. The empty case must return without that calculation.

6. No. For [0, 3, 4], swapping the first and last entries gives [4, 3, 0], which reverses the kept values' order.

7. First reduce r by n. A right rotation is a left rotation by (n - r) % n. Handle empty input before either remainder calculation.

Interview questions

Answer: What is the difference between rotating and shifting?

A rotation wraps displaced values around to the other end. A shift may discard values or require new values to fill vacated positions. I clarify which behaviour is required before choosing a loop.

Answer: Why is three-reversal rotation linear?

Each reversal visits a range a bounded number of times. The combined range lengths are k, n-k and n, so the total is proportional to n. The number of useful rotations no longer multiplies the full-array scan.

Answer: Why does the read/write zero-moving method preserve unread data?

The write position is never ahead of the read position. Each kept value moves to its own position or an earlier one, so future source elements remain available for later visits.

Answer: Does in-place mean the input stays unchanged?

No. In-place usually means the method uses little extra working storage. The input is normally changed, so it is unsuitable when the caller requires the original contents to remain intact.

Quick revision

- Define the required movement and direction first.

- A saved temporary value prevents information loss during a shift.

- Reduce rotation counts only after handling empty input.

- Three reversals can rotate a list in O(n) time with constant working space.

- Stable read/write compaction preserves the order of kept values.

10. Prefix sums, suffix sums and range questions

Suppose a library records how many books it lends each day. One person asks for the total from Monday to Wednesday. Another asks for Tuesday to Friday. Adding the same days repeatedly wastes work. We can prepare running totals once and reuse them.

Start with the straightforward range sum

A **range** selects consecutive positions. To total positions 1 through 3 in [3, 1, 4, 2, 5], visit those entries and add 1 + 4 + 2, giving 7.

For one short question, a direct scan is fine. With q questions, each covering as many as n entries, repeated scanning can cost O(n times q). The total depends on the actual range lengths.

We will use **[begin, end)** boundaries: include begin and stop before end. Thus [1, 4) includes positions 1, 2 and 3. A range [i, i) is empty. end may equal n because it is the stopping boundary, not an element being read.

What is a prefix sum?

A **prefix** is a beginning part. A prefix-sum array stores the total of each beginning part. Our version has n + 1 entries, with a leading zero.

Define P[0] = 0. Then P[i + 1] = P[i] + values[i]. P[i] is the total of the first i elements, not the total through array index i.

Prefix position	Entries included	Total
P[0]	None	0
P[1]	3	3
P[2]	3, 1	4
P[3]	3, 1, 4	8
P[4]	3, 1, 4, 2	10
P[5]	All five entries	15

See the idea



A leading zero makes a half-open range sum prefix[right] - prefix[left]. The right position is excluded.

The extra starting zero makes ranges beginning at zero follow the same formula as every other range.

Why one subtraction answers a range question

P[end] contains everything before end, including the unwanted part before begin. P[begin] contains exactly that unwanted beginning part. Subtracting it leaves the selected range.

Sum of [begin, end) = P[end] - P[begin]. For [1, 4), this is 10 - 3 = 7. For the whole list [0, 5), it is 15 - 0 = 15. For [2, 2), the result is zero.

An inclusive range from left through right can be written [left, right + 1). Make sure right is a valid element index before forming its next boundary.

Complete program: prepare once, ask several questions

For the sum examples in Chapters 10-12, assume at most 1,000 values, each from -1,000 through 1,000 unless a narrower rule is stated. We use **long long** for totals and **%lld** for printing. Under these limits, the shown totals and differences fit comfortably. A wider type alone is not a guarantee against overflow for unlimited inputs.

The prefix destination needs n + 1 writable long long slots and must not overlap the input array. The caller supplies a truthful count. The builder assumes those rules; it is not a general text-input validator.

library_range_totals.c

```c
#include <stdio.h>

void build_prefix(const int values[], size_t count, long long prefix[])
{
    prefix[0] = 0;
    for (size_t i = 0; i < count; i++)
    {
        prefix[i + 1] = prefix[i] + values[i];
    }
}

int range_total(const long long prefix[], size_t count, size_t begin, size_t end,
                long long *answer)
{
    if (prefix == NULL || answer == NULL || begin > end || end > count)
    {
        return 0;
    }
    *answer = prefix[end] - prefix[begin];
    return 1;
}

int main(void)
{
    int books[] = {3, 1, 4, 2, 5};
    long long prefix[6];
    long long answer = 0;
    build_prefix(books, 5, prefix);

    if (range_total(prefix, 5, 1, 4, &answer))
    {
        printf("Indexes 1 through 3: %lld\n", answer);
    }
    if (range_total(prefix, 5, 0, 5, &answer))
    {
        printf("Whole list: %lld\n", answer);
    }
    if (range_total(prefix, 5, 2, 2, &answer))
    {
        printf("Empty range: %lld\n", answer);
    }
    if (!range_total(prefix, 5, 4, 7, &answer))
    {
        printf("Invalid range rejected.\n");
    }
    return 0;
}
```

Output

```
Indexes 1 through 3: 7
Whole list: 15
Empty range: 0
Invalid range rejected.
```

range_total checks the requested boundaries before indexing. It assumes prefix is the correctly built n + 1-entry summary for the supplied list. Failure leaves answer unchanged. Building an empty list still needs one prefix slot for the leading zero.

The preparation costs O(n) time and O(n) summary storage. Each accepted range query is O(1). Preparing once and answering q queries therefore costs O(n + q), excluding the cost of reading or printing unusually large external representations.

Negative values and changes to the original list

Prefix sums also work with negative values. The totals may go down, but subtraction still cancels the unwanted beginning. For [4, -6, 3], the prefix values are [0, 4, -2, 1]. The range [1, 3) totals 1 - 4 = -3.

However, a prefix array describes the values used when it was built. If values[0] changes from 3 to 8, old prefixes that include it are stale. Rebuild or update the affected summaries before using them. Ordinary prefix arrays suit many range queries with few changes; more advanced update-friendly structures come later.

Do not assume this subtraction trick gives a range minimum. Knowing the smallest value before two boundaries does not let us subtract away an earlier minimum.

Suffix sums: totals from the other end

A **suffix** is an ending part. Let S[n] = 0. Working backwards, set S[i] = values[i] + S[i + 1]. S[i] then contains the total from index i through the final element.

library_suffix_totals.c

```c
#include <stdio.h>

void build_suffix(const int values[], size_t count, long long suffix[])
{
    suffix[count] = 0;
    for (size_t i = count; i > 0; i--)
    {
        suffix[i - 1] = values[i - 1] + suffix[i];
    }
}

int main(void)
{
    int books[] = {3, 1, 4, 2, 5};
    long long suffix[6];
    build_suffix(books, 5, suffix);
    printf("Suffix totals:");
    for (size_t i = 0; i < 6; i++)
    {
        printf(" %lld", suffix[i]);
    }
    printf("\nFrom index 2 onward: %lld\n", suffix[2]);
    return 0;
}
```

Output

```
Suffix totals: 15 12 11 7 5 0
From index 2 onward: 11
```

The destination again needs n + 1 slots. The loop uses i > 0 and then i - 1. Using an unsigned counter with i >= 0 would not be a valid stopping rule: unsigned values are never negative. Building a suffix array is O(n) time and O(n) result storage.

Solved problem: a balance position

Find the first index whose values strictly to the left have the same total as the values strictly to the right. The value at the chosen index belongs to neither side.

For [2, 1, 3, 3], index 2 balances: the left total is 2 + 1 = 3 and the right total is 3. We could store all prefixes and suffixes, but we only need one total and a running left total here.

balance_position.c

```c
#include <stdio.h>

int balance_position(const int values[], size_t count, size_t *position)
{
    long long total = 0;
    for (size_t i = 0; i < count; i++)
    {
        total += values[i];
    }
    long long left = 0;
    for (size_t i = 0; i < count; i++)
    {
        long long right = total - left - values[i];
        if (left == right)
        {
            *position = i;
            return 1;
        }
        left += values[i];
    }
    return 0;
}

int main(void)
{
    int values[] = {2, 1, 3, 3};
    size_t position = 0;
    if (balance_position(values, 4, &position))
    {
        printf("First balance index: %zu\n", position);
    }
    return 0;
}
```

Output

```
First balance index: 2
```

This helper follows the stated numeric limits, requires a writable position destination, and requires readable values when count is positive. On a zero count neither loop reads values and there is no balance position. A one-element list balances at index 0 because both outside sides are empty and total zero.

This solution takes O(n) time and O(1) auxiliary space. It uses the prefix idea without needing a stored prefix array. Store only the information the required answers actually need.

Common mistakes and tips

- State whether a range includes or excludes its end.

- Reserve n + 1 slots for this prefix convention; the extra zero is intentional.

- P[i] represents i elements, not all elements through index i.

- Keep the original and summary counts consistent.

- A signed total can overflow in C. Use a suitable type and justified input limits.

- Changing the underlying data can invalidate earlier summaries.

- Do not allocate both a prefix and suffix array automatically if a running total is enough.

Practice questions

1. Build the leading-zero prefix array for [2, 5, 1, 4].

2. Use it to total [1, 4). Which actual elements are included?

3. What is the sum of [2, 2), and why is it not an error here?

4. Build suffix totals, including the final zero, for [2, 5, 1, 4].

5. For [5, -2, 4], find the sum of [1, 3) using prefixes.

6. Why must stored prefixes be reconsidered after changing an input value?

7. Find a balance position in [1, 2, 0, 3]. Does the chosen element belong to either side?

8. Why is O(1) per query not a complete description of preparing and answering many queries?

Answer: Check the practice answers

1. [0, 2, 7, 8, 12]. Each new entry adds the next original value.

2. P[4] - P[1] = 12 - 2 = 10. The selected values are 5, 1 and 4.

3. Zero. It is a valid empty range under this explicitly chosen half-open convention.

4. [12, 10, 5, 4, 0]. The final entry summarises no remaining elements.

5. Prefixes are [0, 5, 3, 7]. The answer is 7 - 5 = 2, matching -2 + 4.

6. They summarise old values. Any prefix including the changed entry may now give the wrong total unless it is updated or rebuilt.

7. Index 2 balances: left is 1 + 2 = 3 and right is 3. The zero at the chosen index belongs to neither side.

8. Building the summaries costs O(n) time and O(n) storage first. With q queries, this version costs O(n + q) time overall.

Interview questions

Answer: Why does the range formula work?

The prefix at end contains the selected range plus everything before begin. Subtracting the prefix at begin cancels that unwanted beginning and leaves the selected total.

Answer: Do prefix sums need non-negative inputs?

No. Addition and subtraction still work when values are negative, provided all required arithmetic fits the chosen type. Prefix totals do not have to be increasing.

Answer: When would you choose a direct scan instead?

For one short range, a scan may be simpler and avoid extra storage. Prefix preparation is useful when many queries reuse the same data. Frequent updates can make a plain prefix array a poor fit.

Answer: Can the balance-position problem use constant extra space?

Yes. Compute the total once, then keep the sum strictly to the left. The right sum is total minus the left sum minus the current value. No full prefix or suffix array is necessary for this one question.

Quick revision

- Prefixes describe beginning totals; suffixes describe ending totals.

- Our leading-zero convention answers [begin, end) using P[end] - P[begin].

- Preparation and query costs must both be reported.

- Negative values are allowed when arithmetic remains in range.

- Stored summaries must match the current input data.