

Level 0 Training

Session 3

Functions and Complexity Analysis

Organize your code. Understand its cost.

A step-by-step reference to read, trace,
practise, and return to after the session.

How to use this reference

This book is a reference for studying **after the session**. It explains what the code means, how to trace it, and why a solution works. You do not need to memorize the programs. You need to understand the decisions behind them.

A practical study routine

1. **Read one idea.** Explain the new term in your own words. A function is easier to understand when you can say exactly what job it performs.
2. **Trace a small example.** Record the arguments, the local variables, and the returned result. For a loop, count what actually repeats.
3. **Type and change the program.** Predict the output before running it. Change one input or one line and explain the difference.
4. **Solve independently.** Try the practice, then its hint, then the explained solution. Later, write the solution again without looking.

How the examples are labelled

A **complete program** includes headers and `main` and can be compiled on its own. A fragment marked **above main** defines a helper; a fragment marked **inside a function** belongs in a function body. Deliberately **wrong examples** are not templates to copy.

Pause and predict questions have answers nearby. Cover the answer first. Topics labelled **Optional extension** are a second-reading route, not extra requirements for the core practice.

The language and source package

All complete examples target **C++23**. Each listing is loaded directly from a separate `.cpp` file in the editable LaTeX package. Compile one program at a time: these files each have their own `main` and must not be linked together.

Read in stages, not in one sitting

Session 3 assumes you can use variables, conditions, loops, arrays, and `std::string`. Return to Sessions 1 and 2 whenever one of those building blocks feels unfamiliar.

Study ordinary functions before references, then learn to count loop work before using Big-O notation. The quick-reference pages support revision; they do not replace the worked explanations.

The Level 0 learning path

Session	Topics
1	Intro to problem solving: data types, conditions, and loops.
2	Arrays and strings.
3	Functions and complexity analysis.
4	STL, part 1: vector, pair, deque, queue, priority queue, and stack.
5	STL, part 2: set, multiset, ordered set, ordered map, and unordered map.
6	Static range queries: prefix sum, partial sum, and frequency array.

What is included in this edition

This booklet contains **Session 3: Functions and Complexity Analysis**. Its section numbers begin with 3 to match the collected book. The updated collected edition contains Sessions 1–3.

What Session 3 will help you do

Write and call helpers with clear inputs and results. Distinguish returning from printing. Choose value, reference, or read-only reference parameters. Pass arrays and strings safely. Estimate time and extra memory, including work hidden inside function calls.

What can wait

Overloading, default arguments, matrix parameters, and an introductory look at recursion are optional. Templates, lambdas, advanced pointer techniques, recurrence-solving methods, and later-session containers are not prerequisites.

Conventions

Input follows the stated constraints. Text tasks use ASCII characters, not arbitrary Unicode text. Unless a section says otherwise, complexity describes **worst-case work** under the fixed-width operation model introduced in this chapter. A bound is an estimate of growth, not a promise of an exact runtime.

Contents

How to use this reference	i
The Level 0 learning path	ii
3 Functions and Complexity Analysis	1
3.1 Why do we need functions?	2
3.2 Your first value-returning function	3
3.3 Parameters, arguments, and the flow of a call	5
3.4 Return values, printing, and early exits	7
3.5 Void functions and functions with no parameters	10
3.6 Scope, lifetime, and independent calls	12
3.7 Declarations, definitions, and program organization	14
3.8 Pass by value: work on a separate value	16
3.9 References: allow a helper to change the caller	18
3.10 Arrays as function inputs	21
3.11 Strings by value, by reference, and by const reference	24
3.12 Designing, testing, and combining small helpers	27
3.13 Optional extension: overloads and default arguments	29
3.14 From correct code to efficient code	31
3.15 Big-O: describe how the work grows	33
3.16 Single loops and sequential blocks	35
3.17 Nested loops and dependent bounds	37
3.18 Halving, doubling, and square-root work	39
3.19 Count total progress, not only the loop shape	42
3.20 Function calls and hidden string costs	44
3.21 Space complexity: count memory that is alive together	47
3.22 Constraints, time limits, and multiple test cases	49
3.23 Worked comparisons: improve the algorithm, not the formatting	51
3.24 Optional extension: recursion and the call stack	54
3.25 Optional extension: passing a two-dimensional array	57
3.26 Debugging functions and checking an analysis	59
3.27 Practice problems: write a helper and explain its cost	61
3.28 Complexity drills: count before naming the growth rate	67
3.29 Hints before solutions	72
3.30 Explained practice solutions	75
3.31 Explained complexity drills	94
3.32 Quick reference, glossary, and readiness checklist	100
3.33 Technical notes and references	104

SESSION 3

Functions and Complexity Analysis

Give each piece of work a name. Understand how much work it does.

You already know how to read input, make decisions, repeat work, and store collections. As your solutions grow, putting everything inside `main` becomes difficult to follow. A **function** lets you describe one part of the solution separately and use it when needed.

A correct solution must also finish within the problem's limits. **Complexity analysis** helps you estimate how the amount of work and memory grows when the input grows. It is not about counting how many lines you typed.

Two questions for every solution

Question	Tool	Example
What does this part do?	A function with a clear job.	<code>isPalindrome(text)</code> checks one string.
How much work does it do?	Time and space analysis.	A scan checks at most all the characters and uses a few counters.

Your route through the chapter

Stage A: organize a program. Learn definitions, calls, parameters, return values, scope, references, and helpers for arrays and strings.

Stage B: estimate the cost. Start by counting a loop on paper. Then learn growth rates, nested loops, logarithmic work, function-call costs, extra memory, and total work across test cases.

Stage C: practise and explain. Solve coding problems and short analysis drills. Use a hint before the full solution. Return to the optional extensions when the foundations feel comfortable.

You do not need advanced mathematics to begin

For the core material, you need addition, multiplication, and the ability to trace a loop. Logarithms are introduced as repeated halving. Recursion and its call stack have their own optional section; they are not prerequisites for the ordinary function examples.

3.1 Why do we need functions?

One named job instead of repeated instructions

Suppose a program checks whether several numbers are even. You can write `value % 2 == 0` each time. In a larger task, however, the repeated check might contain ten lines and several special cases. Copying it into three places means you must remember to fix all three when you discover a mistake.

A function gives those instructions a name. You write the job once, then **call** it whenever the solution needs that job. A useful name explains the intention: `countVowels`, `sumArray`, or `isPrime`.

You have already met functions

`main` is a function. A call such as `text.size()` asks a standard-library object for its length. Now you will write your own functions, using the same variables, conditions, and loops that you already know.

Think of a function as a small tool with a description: *what does it receive, what does it produce, and what is it allowed to change?* That description is its **contract**. For a helper that returns the largest array element, a necessary condition is that the array is not empty.

Definition and call are different actions

A **definition** describes the instructions. A **call** requests that they run. Merely putting a helper above `main` does not run the helper's body. In these ordinary programs, the helper runs when execution reaches a call to it.^{[3-1][3-2]}

A recipe written in a notebook is not a cooked meal. Similarly, the existence of `sumArray` does not calculate a sum until the program calls it with a particular array and length.

Functions improve organization, not automatically speed

Moving the same nested loops into a helper does not remove any of their repeated work. Functions help you organize and test an algorithm. Making the algorithm faster requires examining what those functions actually do.

Pause and predict – 1

A program defines `printWelcome` above `main`, but `main` never calls it. Will the helper print a welcome message?

Check your prediction. No. Defining a helper describes its body; a call is what requests that body to run.

3.2 Your first value-returning function

A complete example: add two numbers

E01 reads two integers in $[-1000, 1000]$, calls a helper, and prints their sum. The helper does the calculation; `main` handles the input and output.

Complete program E01 – one function, one result

```
#include <iostream>

int add(int a, int b) {
    return a + b;
}

int main() {
    int x = 0, y = 0;
    std::cin >> x >> y;
    int answer = add(x, y);
    std::cout << answer << '\n';
    return 0;
}
```

Sample input

4 7

Sample output

11

Read the definition from the outside in

In `int add(int a, int b)`, the first `int` is the **return type**: the kind of result the caller receives. `add` is the function's name. The parentheses contain its **parameters**: named inputs available inside the body. Each parameter has its own type.

The braces enclose the **body**. `return a + b;` calculates a result and sends it back to the place that called the function. The caller can store that result, compare it, or use it in another calculation.^{[3-1][3-3]}

Trace the call with the sample

Step	Where execution is	What happens
1	<code>main</code> reads input.	<code>x = 4, y = 7.</code>
2	<code>add(x, y)</code> is called.	The call supplies 4 and 7 to <code>a</code> and <code>b</code> .
3	Inside <code>add</code> .	<code>a + b</code> evaluates to 11.
4	<code>return</code> in <code>add</code> .	The call finishes with result 11.
5	Back in <code>main</code> .	<code>answer</code> receives 11; output prints it.

The caller pauses while the helper does its work, then continues with the returned value. It does not start `main` again. The names `a` and `b` belong to the helper; the caller does not have to use those same names.

Where to put the helper

For now, define ordinary helpers **above main**. Later in this chapter, a declaration will let you place the definition below it. Do not put a named function definition inside another function's body.^[3-4]

Compile one complete program

Save E01 as `add.cpp`. A GNU build command is:

```
g++ -std=c++23 -Wall -Wextra -Wpedantic add.cpp -o add
./add
```

The first command builds the program; the second runs it in a typical Linux terminal. Select C++23 in your editor or training platform when compiling there.^[3-18]

Pause and predict – 2

In E01, could the caller use `int result = add(10, -3);` even though there are no caller variables named `a` and `b`?

Check your prediction. Yes. The arguments are 10 and -3, and the returned result is 7. The parameter names are local names used by the helper.

3.3 Parameters, arguments, and the flow of a call

A parameter is a name; an argument supplies an input

In `int add(int a, int b)`, `a` and `b` are parameters. In `add(x, 5)`, the expressions `x` and `5` are arguments. They supply the inputs for this particular call. Argument expressions may be variables, literals, calculations, or other function calls.^[3-2]

For the ordinary functions in this chapter, match the number and order of the arguments to the parameter list. A parameter needs a type even when its neighbour has the same type: write `int a`, `int b`, not `int a, b` in the parameter list.

Fragment – define above main:

```
int subtract(int first, int second) {
    return first - second;
}
```

`subtract(9, 4)` returns 5. `subtract(4, 9)` returns -5. Changing the argument order changes which parameter receives each value.

Each call has its own ordinary local variables

E02 reads `x` in $[-1000, 1000]$. The first call applies the rule $2v + 1$ to `x`; the second applies that rule to the first result.

Complete program E02 – trace two separate calls

```
#include <iostream>

int twicePlusOne(int value) {
    int result = 2 * value + 1;
    return result;
}

int main() {
    int x = 0;
    std::cin >> x;
    int a = twicePlusOne(x);
    int b = twicePlusOne(a);
    std::cout << x << ' ' << a << ' ' << b << '\n';
    return 0;
}
```

Sample input	Sample output
3	3 7 15

Write a little record for each call

Call	Parameter	Result
<code>twicePlusOne(x)</code>	<code>value = 3</code>	<code>result = 7</code> , stored in <code>a</code> .
<code>twicePlusOne(a)</code>	<code>value = 7</code>	<code>result = 15</code> , stored in <code>b</code> .

The second call does not reuse the first call's old `result` variable. It creates its own ordinary local variable and initializes it again. Meanwhile, the caller's `x` remains 3.

Nested calls still have a clear dependency

`add(add(2, 3), 4)` must obtain the inner result 5 before the outer addition can use it. The outer call then receives 5 and 4 and returns 9.

Do not generalize this into a rule that all argument expressions run from left to right. C++ does not promise that order between separate arguments. When argument calculations change state or print messages, place them in separate statements to make the intended order explicit.^[3-2]

Types are part of the contract

A function taking `int` parameters is not the right interface for preserving fractional input. Converting 3.9 to an `int` loses the fraction. Choose parameter and return types for the values that the task actually permits, not merely for the sample.

Pause and predict – 3

For E02 with input 0, what are the three printed values? Does changing the helper's parameter name to `x` change the answer?

Check your prediction. The output is 0 1 3. Renaming the parameter consistently does not change the computation. It still represents a separate input to each call.

3.4 Return values, printing, and early exits

Returning and printing solve different problems

`return value;` gives a result to the **caller**. `std::cout << value;` writes characters to the **output stream**. Returning does not print automatically, and printing does not supply a numeric result for the caller to store.

If `add` returns 11, `int total = add(4, 7);` stores 11 without printing anything. `std::cout << add(4, 7);` uses the returned value as something to print.

Fragment – inside main, with E01’s helper available:

```
int total = add(4, 7);           // Stores 11; prints nothing.
int doubled = 2 * total;        // Uses the stored result.
std::cout << doubled << '\n';
```

The return type does not repair earlier arithmetic

E03 reads two integers in $[-10^9, 10^9]$ and prints their average to one decimal place. A wide intermediate sum is used before converting to floating point.

Complete program E03 – return a fractional result

```
#include <iomanip>
#include <iostream>

double average(int a, int b) {
    long long total = 1LL * a + b;
    return static_cast<double>(total) / 2.0;
}

int main() {
    int x = 0, y = 0;
    std::cin >> x >> y;
    std::cout << std::fixed << std::setprecision(1);
    std::cout << average(x, y) << '\n';
    return 0;
}
```

Sample input	Sample output
5 8	6.5

`return (a + b) / 2;` would perform integer arithmetic before conversion to the function’s `double` result. A return type describes the result type; it does not rewrite the expression’s arithmetic. Recall the integer-division and overflow rules from Session 1.

A Boolean helper answers a yes-or-no question

A helper whose name starts with `is`, `has`, or `can` often returns `bool`. The caller decides how to display or use that decision. E04 accepts an integer in $[-10^9, 10^9]$.

Complete program E04 – use a returned Boolean

```
#include <iostream>

bool isEven(int value) {
    return value % 2 == 0;
}

int main() {
    int n = 0;
    std::cin >> n;
    if (isEven(n)) {
        std::cout << "YES\n";
    } else {
        std::cout << "NO\n";
    }
    return 0;
}
```

Sample input	Sample output
-6	YES

`return value % 2 == 0;` returns the comparison's truth value. It is equivalent here to an `if` that returns `true` on a match and `false` otherwise. The name `isEven` makes the caller read almost like a sentence.

Return leaves the current function immediately

A function may have several **return** statements on different paths. Only the one reached in that call supplies its result. Instructions after an executed **return** in the same function are not executed.^[3-3]

Fragment – define above main:

```
int absoluteSmall(int value) { // Contract: -1000 <= value <= 1000.
    if (value < 0) {
        return -value;
    }
    return value;
}
```

For -8, the first return gives 8. For 5 or 0, the condition is false and the final return handles the remaining path. The small bound avoids the special problem of negating the most negative representable signed integer.

A **return** inside a loop leaves the *function*, not only the loop. A **break** leaves the innermost loop or switch; a **continue** skips to the next iteration of the current loop. Choose the statement that matches your intention.

Every result-producing path needs a result

For these ordinary non-void helpers, reaching the end without returning a value is an error in the program's logic and can cause undefined behavior. Do not assume the compiler inserts zero. **main** has a special rule; that exception is not a rule for your helpers.^[3-3]

Pause and predict – 4

A helper returns 12, and **main** executes only `calculate();`. Must 12 appear on the screen?

Check your prediction. No. The caller discarded the result. Output appears only if the helper or caller explicitly performs output.

3.5 Void functions and functions with no parameters

Some helpers perform an action instead of returning a value

`void` means that the function does not provide a result value. Such a helper can print, change an object through a suitable parameter, or perform another action. E05 reads $0 \leq n \leq 80$, prints a line of `n` hashes, then prints a line of three hashes.

Complete program E05 – a printing helper

```
#include <iostream>

void printLine(int length) {
    for (int i = 0; i < length; ++i) {
        std::cout << '#';
    }
    std::cout << '\n';
}

int main() {
    int n = 0;
    std::cin >> n;
    printLine(n);
    printLine(3);
    return 0;
}
```

Sample input

5

Sample output

#####

###

When `n = 0`, the loop prints no hashes, but the newline still creates an empty first line. After `printLine(n)` finishes, `main` continues to the next call.

Return without a value

A void helper can use `return;` to finish early. It can also simply reach its closing brace. Do not try to store the result of `println(5)` in an integer: this function has no value to store.^[3-3]

An empty parameter list is allowed

Fragment – define above `main`:

```
void printWelcome() {  
    std::cout << "Welcome to Level 0!\n";  
}
```

Call it with `printWelcome();`. The empty parentheses mean that this function needs no arguments. A no-parameter helper can also return a value; parameter count and return type are separate choices.

Pause and predict – 5

A void helper executes `return;` while called from `main`. Does that automatically end the whole program?

Check your prediction. No. It finishes that helper call. The caller continues after the call unless its own control flow ends the program.

3.6 Scope, lifetime, and independent calls

Where can a name be used?

Scope is the part of the program where a name is available. A variable declared inside a function's block is not a variable that unrelated functions can use by name. A parameter is available inside its function, not as a new name in the caller.^[3-5]

E06 has two different variables named `total`: one in `main`, and one in `localSum`. They are not the same object.

Complete program E06 – local state does not leak between calls

```
#include <iostream>

int localSum(int n) {
    int total = 0;
    for (int i = 1; i <= n; ++i) {
        total += i;
    }
    return total;
}

int main() {
    int total = 100;
    int first = localSum(3);
    int second = localSum(4);
    std::cout << first << ' ' << second << ' ';
    std::cout << total << '\n';
    return 0;
}
```

Sample input

No input.

Sample output

6 10 100

Trace the three printed values

`localSum(3)` starts a fresh `total` at zero and adds 1, 2, 3, returning 6. `localSum(4)` starts another fresh `total` at zero and returns 10. The caller's `total` was initialized to 100 and neither call changes it.

An ordinary local variable is not a memory of the previous call. Its initialization runs again each time execution reaches its declaration in a new call.^[3-6]

A loop's counter may belong only to the loop

Wrong fragment – inside a function:

```
for (int i = 0; i < 3; ++i) {  
    std::cout << i << '\n';  
}  
std::cout << i; // Wrong: this loop's i is out of scope.
```

Declare a variable in the surrounding block only when later code genuinely needs it there. Otherwise, a smaller scope makes accidental reuse less likely.

Scope is not the same as lifetime

Lifetime concerns when an object exists. The ordinary local objects in these helpers do not survive the end of the call. Returning a numeric or string value lets the caller receive a result; returning a reference to a destroyed local object does not keep that object alive.^{[3-6][3-7]}

You will learn references shortly. For now, use ordinary return-by-value results and avoid shared mutable global variables. A constant such as a fixed matrix width can be shared deliberately; a hidden running total makes separate calls harder to reason about.

Pause and predict – 6

If E06 called `localSum(3)` twice, would the second call return 12 because the first call already added 6?

Check your prediction. No. Each call initializes its own local total to zero, so each returns 6.

3.7 Declarations, definitions, and program organization

Tell the compiler the interface before the call

Defining helpers above `main` is the simplest arrangement. Another valid arrangement begins with a **declaration**, sometimes called a **prototype**. It states the name, return type, and parameter types, ending in a semicolon. The definition supplies the body later.^{[3-1][3-4]}

E07 reads two integers in $[-1000, 1000]$ and prints the larger one. It demonstrates this second layout.

Complete program E07 – declare first, define later

```
#include <iostream>

int larger(int a, int b);

int main() {
    int x = 0, y = 0;
    std::cin >> x >> y;
    std::cout << larger(x, y) << '\n';
    return 0;
}

int larger(int a, int b) {
    if (a > b) {
        return a;
    }
    return b;
}
```

Sample input	Sample output
9 4	9

Three lines can look similar but mean different things

Code shape	Meaning
<code>int larger(int a, int b);</code>	Declaration: the interface, without a body.
<code>int larger(int a, int b) { ... }</code>	Definition: the instructions implementing that interface.
<code>larger(x, y)</code>	Call: use the function with these arguments.

The definition itself does not need a semicolon after its closing brace. Statements such as the call in `main` do end with semicolons.

Keep the interface consistent

A declaration promising `int larger(int, int)` must match its corresponding definition. Parameter names in the declaration are optional and do not have to match the definition's local names; the types and function identity are what matter.

A declaration alone is not an implementation. If an ordinary helper is called but its required definition is missing from the built program, the build can fail at the linking stage. In this book, every complete program includes its own definitions so one file is enough.

A useful default layout

Put headers first, then needed constants, then helper definitions, then `main`. Use prototypes when a different order makes a particular program clearer. You do not need to split a small contest solution into many files.

Pause and predict – 7

Can moving a helper's definition below `main`, without adding a declaration before the call, always preserve a valid program?

Check your prediction. No. The name must be declared where the call is compiled. Keep the definition above the call or add a matching declaration first.

3.8 Pass by value: work on a separate value

The usual scalar parameter is independent of the caller

In `void incrementCopy(int value)`, the parameter is an ordinary `int` initialized from the argument. Changing it does not change the caller's integer.^[3-2]

E08 reads an integer in $[-1000, 1000]$. Output inside the helper shows its modified local value; output inside `main` shows the unchanged caller value.

Complete program E08 – a changed copy is not a changed caller

```
#include <iostream>

void incrementCopy(int value) {
    ++value;
    std::cout << value << '\n';
}

int main() {
    int x = 0;
    std::cin >> x;
    incrementCopy(x);
    std::cout << x << '\n';
    return 0;
}
```

Sample input

8

Sample output

9

8

A trace with two separate boxes

Moment	Caller x	Helper value
Before the call	8	Not created yet.
After the parameter is initialized	8	8.
After <code>++value</code>	8	9.
After the call finishes	8	Its ordinary local life-time has ended.

The word *copy* here describes independent value semantics. It is not a promise about exact machine instructions after compiler optimization.

Returning a changed value is often enough

A helper can return a new result rather than directly changing the caller.

Fragment – define above main:

```
int increased(int value) { // Contract: -1000 <= value <= 1000.  
    return value + 1;  
}
```

With this helper, `x = increased(x);` explicitly replaces `x`. Writing only `increased(x);` discards the returned result and leaves `x` unchanged.

This is a good default for small inputs such as `int`, `long long`, `double`, and `char`: receive a value, calculate an answer, and return it. The caller can see the assignment that changes its state.

Pause and predict – 8

A helper is defined as `void reset(int x) { x = 0; }`. After `int score = 90; reset(score);`, what is `score`?

Check your prediction. It is still 90. The assignment changes only the helper's value parameter.

3.9 References: allow a helper to change the caller

A reference is another name for an existing object

`int& value` declares a reference parameter. For a call such as `increment(x)`, the parameter refers to the caller's `x`; assigning through the parameter changes that same integer. The `&` is part of the parameter declaration. Do not add it to the call's argument.^{[3-7][3-8]}

E09 accepts the same input range as E08, $[-1000, 1000]$. Compare the two parameter declarations carefully.

Complete program E09 – modify the original integer

```
#include <iostream>

void increment(int& value) {
    ++value;
}

int main() {
    int x = 0;
    std::cin >> x;
    increment(x);
    std::cout << x << '\n';
    return 0;
}
```

Sample input

8

Sample output

9

Here there is one integer object with two usable names during the call. The helper does not need to return an integer because the intended effect is to modify that object.

A small helper with two mutable inputs

E10 reads two integers in $[-10^9, 10^9]$ and arranges the two caller variables so the smaller value comes first. The temporary variable preserves the value that would otherwise be overwritten.

Complete program E10 – put two caller values in order

```
#include <iostream>

void orderPair(int& a, int& b) {
    if (a > b) {
        int temporary = a;
        a = b;
        b = temporary;
    }
}

int main() {
    int x = 0, y = 0;
    std::cin >> x >> y;
    orderPair(x, y);
    std::cout << x << ' ' << y << '\n';
    return 0;
}
```

Sample input

9 2

Sample output

2 9

If $x \leq y$ already holds, the helper changes nothing. Equal inputs also remain unchanged. This is a deliberate side effect, clearly signalled by the reference parameters.

Not every expression is a modifiable object

An ordinary non-`const int&` parameter cannot bind to an integer literal such as 8 or a temporary result such as `x + 1`. There is no suitable caller integer variable there for this interface to modify. Pass an actual modifiable `int`, such as `x`.^[3-8]

Wrong calls – with E09’s helper available:

```
increment(8);           // Wrong for int&.
increment(x + 1);       // Wrong for int&.
```

Read-only references

`const int& value` provides a reference through which the helper cannot assign to that integer. For small integers, value parameters are usually simpler. For a larger object such as a string, `const std::string&` is especially useful: read the caller’s text without making a full parameter copy.

Do not change everything to a reference

Use a mutable reference when changing the caller is part of the job. Use a returned value when a calculation produces one answer. Use a read-only reference for a larger object that only needs inspection. Making all parameters mutable references hides which changes are intentional.

Pause and predict – 9

What changes between `void update(int x)` and `void update(int& x)` if the body assigns `x = 5`;

Check your prediction. The first changes a separate value parameter. The second changes the caller’s integer object. The call still looks like `update(number)` in both cases.

3.10 Arrays as function inputs

Send the array together with its active length

A helper often needs the elements of an array, not just one number. In this beginner interface, `const int a[]` gives read-only access to the caller's elements, and `int n` tells the helper how many active elements to inspect.

E11 reads $0 \leq n \leq 1000$ and `n` integers in $[-10^9, 10^9]$. Its result uses `long long` because the sum may exceed `int`.

Complete program E11 – receive elements and their length

```
#include <iostream>

long long sumArray(const int a[], int n) {
    long long total = 0;
    for (int i = 0; i < n; ++i) {
        total += a[i];
    }
    return total;
}

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::cin >> a[i];
    }
    std::cout << sumArray(a, n) << '\n';
    return 0;
}
```

Sample input

4
8 -2 5 9

Sample output

20

What exactly happens to an array parameter?

A built-in array parameter written `int a[]` is adjusted by the language to a pointer parameter. The caller supplies access to its original array, not a separate copy of all its elements. You can use the familiar `a[i]` syntax without studying pointer arithmetic yet.^{[3-1][3-9]}

This is **not the same declaration** as a C++ reference-to-array parameter. Saying loosely that an array is “passed by reference” hides an important distinction: the simple `a[]` form does not preserve its length as part of the parameter type.

Even spelling a bound as `int a[1000]` in this parameter list does not automatically check the caller's capacity or tell the helper the active length. Pass `n` explicitly and respect it. Using `sizeof(a)` inside this helper does not recover the caller's array length.^[3-1]

The helper's contract

The caller must supply valid elements at indices 0 through `n - 1`. The helper visits precisely that range and does not modify the elements. For `n = 0`, it does not access any element and returns zero. The sample's running total is 8, 6, 11, then 20.

`const` prevents writes *through this parameter*. It documents and checks the read-only role; it does not mean that every possible alias to the caller's array is frozen forever.

A helper may deliberately update the elements

E12 uses `int a[]` without `const`. It replaces every occurrence of one value with another, then `main` prints the modified array. Here $0 \leq n \leq 1000$, and all array, old, and new values lie in $[-10^9, 10^9]$.

Complete program E12 – update the caller's array

```
#include <iostream>

void replaceValue(int a[], int n, int oldValue, int newValue) {
    for (int i = 0; i < n; ++i) {
        if (a[i] == oldValue) { a[i] = newValue; }
    }
}

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0, oldValue = 0, newValue = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    std::cin >> oldValue >> newValue;
    replaceValue(a, n, oldValue, newValue);
    for (int i = 0; i < n; ++i) {
        if (i > 0) { std::cout << ' '; }
        std::cout << a[i];
    }
    std::cout << '\n';
    return 0;
}
```

Sample input

```
4
2 7 2 5
2 9
```

Sample output

```
9 7 9 5
```

The array's *elements* can change through `a`. The `n` parameter is still an ordinary integer value parameter. Assigning a different number to that local `n` would not change the caller's length variable.

Return a position with a clearly defined missing result

E13 reads $0 \leq n \leq 1000$, the array, and then a target. Values and target lie in $[-10^9, 10^9]$. It returns the first **zero-based index**, or -1 when absent.

Complete program E13 – a reusable linear search

```
#include <iostream>

int firstIndex(const int a[], int n, int target) {
    for (int i = 0; i < n; ++i) {
        if (a[i] == target) { return i; }
    }
    return -1;
}

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0, target = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    std::cin >> target;
    std::cout << firstIndex(a, n, target) << '\n';
    return 0;
}
```

Sample input

5
4 6 4 8 6
6

Sample output

1

`return i;` exits as soon as the first match is found. `return -1;` is outside the loop because failure is known only after every candidate was checked. A negative result is useful here because all valid indices are nonnegative. Never use `a[position]` until you know that `position` is valid.

Pause and predict – 10

A helper receives `const int a[]` and `n = 3`. May it inspect `a[3]` just because the caller allocated 1000 cells?

Check your prediction. Not as part of the promised three-element input. The active indices are 0, 1, and 2. Allocated capacity and active length have different roles.

3.11 Strings by value, by reference, and by const reference

Choose whether the caller's text should change

Unlike the built-in `a[]` parameter syntax, `std::string text` is an object passed by value. Calling it with an existing string normally creates a separate string value for the parameter. A `std::string&` parameter can modify the caller's text; a `const std::string&` parameter reads it without making that full copy.^{[3-2][3-8][3-10]}

Parameter	Meaning for an existing caller string
<code>std::string text</code>	Work with an independent value; useful when a changed copy is intended.
<code>std::string& text</code>	Refer to and permit changes to the caller's string.
<code>const std::string& text</code>	Refer to the caller's string for read-only work.

Pause and predict – 11

A palindrome helper only compares characters. Which parameter best expresses that job: `std::string`, `std::string&`, or `const std::string&`?

Check your prediction. `const std::string&`. It avoids a full parameter copy of an existing string and prevents writes through the parameter.

One helper can use another helper

E14 reads one ASCII line, possibly empty, of length at most 100000. It counts **lowercase** vowels only. The small helper answers whether one character is a vowel; the larger helper counts matching characters.

Complete program E14 – a read-only string helper

```

#include <iostream>
#include <string>

bool isLowerVowel(char c) {
    return c == 'a' || c == 'e' || c == 'i'
        || c == 'o' || c == 'u';
}

int countVowels(const std::string& text) {
    int count = 0;
    int n = static_cast<int>(text.size());
    for (int i = 0; i < n; ++i) {
        if (isLowerVowel(text[i])) { ++count; }
    }
    return count;
}

int main() {
    std::string line;
    std::getline(std::cin, line);
    std::cout << countVowels(line) << '\n';
    return 0;
}

```

Sample input

a quiet room

Sample output

6

`countVowels` calls `isLowerVowel` once for each visited character. There are six lowercase vowels in the sample: a, then u i e, then o o. Uppercase vowels do not count under this particular contract.

The length is converted to `int` only because the stated bound fits the training environment's integer range. `getline` is the first input operation, so there is no leftover numeric-input newline to discard. Keep the Session 2 input rules when adapting the helper to another program.

Returning a changed copy is a different job

E15 reads an ASCII line of length at most 100000, including an empty line. It prints the original and a reversed copy on separate lines. This example deliberately uses a value parameter because the helper is supposed to edit an independent copy.

Complete program E15 – return a string by value

```

#include <iostream>
#include <string>

std::string reversedCopy(std::string text) {
    int left = 0;
    int right = static_cast<int>(text.size()) - 1;
    while (left < right) {
        char temporary = text[left];
        text[left] = text[right];
        text[right] = temporary;
        ++left;
        --right;
    }
    return text;
}

int main() {
    std::string original;
    std::getline(std::cin, original);
    std::string result = reversedCopy(original);
    std::cout << original << '\n' << result << '\n';
    return 0;
}

```

Sample input

code now

Sample output

code now

won edoc

For empty text, `right` becomes -1 in a signed integer and the loop never runs. For a one-character string, `left == right` and no swap is needed. The caller's `original` remains unchanged in every case.

Returning a string *by value* is valid. It does not leave the caller referring to a destroyed local object. Compilers and the language's value-transfer rules can avoid unnecessary extra copies; you do not need to return a reference to a local string to make this interface usable.^{[3-3][3-10]}

3.12 Designing, testing, and combining small helpers

Write the contract before the implementation

For a proposed `indexOfMinimum` helper, write: “Receive a nonempty array and its length. Return the first zero-based position containing the minimum. Do not modify the input.” That one sentence resolves several choices before any loop is written.

The **preconditions** are what the caller must guarantee, such as `n >= 1` and valid array storage. The **postcondition** is what the helper guarantees on return, such as the earliest minimum index. The task’s input constraints help you decide whether a precondition is always satisfied.

Separate computation from display when practical

A helper returning `bool` can support many callers: one prints YES, one increments a counter, another uses it inside a larger condition. If the helper instead prints its answer, those callers are harder to combine without unwanted output.

Printing helpers are still useful when printing is their named job. `printLine` is clear. A helper named `isPrime` should generally return a decision rather than unexpectedly print messages.

Make a tiny test table

Helper	Important tests	Reason
<code>isEven</code>	Negative, zero, positive.	The mathematical property includes all three.
<code>firstIndex</code>	First, last, repeated, absent.	Position and early-exit errors appear here.
<code>sumArray</code>	Empty, one value, all negative, wide sum.	Covers neutral value, bounds, and result type.
<code>isPalindrome</code>	Empty, one letter, odd/even length, mismatch.	Covers both loop boundaries and decision paths.

Use `solve()` for one test case

E16 reads $1 \leq t \leq 1000$. Each case contains $0 \leq n \leq 100000$ scores in $[0, 100]$; the sum of all lengths is at most 100000. A score of at least 50 passes. Print the passing count for each case.

Complete program E16 – main repeats; solve handles one case

```
#include <iostream>

bool isPassing(int score) {
    return score >= 50;
}

void solve() {
    int n = 0, passed = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        int score = 0;
        std::cin >> score;
        if (isPassing(score)) { ++passed; }
    }
    std::cout << passed << '\n';
}

int main() {
    int t = 0;
    std::cin >> t;
    for (int test = 0; test < t; ++test) { solve(); }
    return 0;
}
```

Sample input**Sample output**

3	2
4	0
40 50 90 0	1
0	
2	
49 51	

main reads **t** only once. Each call to **solve** reads exactly one case. Its local **passed = 0** is initialized again for that call, so counts do not leak from an earlier case. No array is needed because the answer never revisits a score.

Prefer useful boundaries, not the largest number of functions

A helper should represent a meaningful task, a reusable rule, or a part worth testing independently. There is no prize for creating one helper for every arithmetic symbol. A short, clear program can use only one or two helpers.

Pause and predict – 12

E16 processes a case with ten passing scores and then a case with no scores. What should the second output be, and why?

Check your prediction. Zero. The second call initializes its own **passed** count to zero and its input loop runs zero times.

3.13 Optional extension: overloads and default arguments

Second-reading material

These tools make some interfaces convenient. You can solve all the core practice with distinct helper names and explicit arguments, so learn ordinary calls first.

Overloading: the same name for different parameter lists

Functions can share a name when their parameter lists distinguish them. In E17, one `larger` handles two integers and another handles two floating-point values. The argument types determine which matching helper is selected.^[3-11]

Complete program E17 – two interfaces with the same name

```
#include <iostream>

int larger(int a, int b) {
    if (a > b) { return a; }
    return b;
}

double larger(double a, double b) {
    if (a > b) { return a; }
    return b;
}

int main() {
    std::cout << larger(4, 9) << '\n';
    std::cout << larger(7.5, 2.5) << '\n';
    return 0;
}
```

Sample input

No input.

Sample output

9
7.5

Changing only the return type does not distinguish overloads that otherwise have the same parameter list. Also, `larger(4, 2.5)` is not a good call for this pair of overloads: each candidate would need a conversion for a different argument, and the call is ambiguous. Use two appropriately typed arguments or a clearer interface.

Default arguments: omit a trailing input when a default is suitable

A default argument supplies an input when the caller leaves it out. The defaulted parameters are trailing parameters in these ordinary declarations. E18 reads a non-whitespace ASCII character and $0 \leq n \leq 40$, then prints it using the default count and the explicit count.^[3-12]

Complete program E18 – use or override a default

```
#include <iostream>

void printRepeated(char symbol, int count = 3) {
    for (int i = 0; i < count; ++i) {
        std::cout << symbol;
    }
    std::cout << '\n';
}

int main() {
    char symbol = '#';
    int n = 0;
    std::cin >> symbol >> n;
    printRepeated(symbol);
    printRepeated(symbol, n);
    return 0;
}
```

Sample input

* 5

Sample output

`printRepeated(symbol)` uses 3. `printRepeated(symbol, n)` uses `n`. The default is not a minimum and does not override an explicitly supplied zero.

When using a separate declaration and definition, normally place the default only in the declaration visible to the caller. Do not repeat the same default argument again in the definition.^[3-12]

Pause and predict – 13

With E18's helper, what does `printRepeated('##', 0)` print?

Check your prediction. Only a newline. The explicit zero is used, so the character-printing loop has no iterations.

3.14 From correct code to efficient code

A small sample cannot reveal every performance problem

Imagine checking every pair among n students. With 5 students there are only 10 distinct unordered pairs. With 100000 students there are $100000 \times 99999 / 2 = 4,999,950,000$ pairs. The same idea can look harmless on a sample and require billions of checks at the maximum size.

Correctness asks whether the answer is right. Efficiency asks how much time and memory the method needs. An efficient wrong answer is still wrong; a correct method that exceeds the task's limits is not a usable submission.

Start by naming the input-size parameters

Situation	A useful size description
An array	n : its number of active elements.
A string	L : its number of stored characters in these ASCII tasks.
A rectangular matrix	r rows and c columns; rc cells in total.
Several test cases	t cases with lengths $n_1, n_2, \dots, n_t$.
Trial divisors for an integer	x : the numeric value being tested; state this explicitly.

Do not replace two independent sizes with one n unless the statement relates them. A scan of a 2-by-1000 matrix and a scan of a 1000-by-2 matrix both visit 2000 cells.

Pause and predict – 14

A program scans every cell of a matrix with r rows and c columns once. What size expression counts the visited cells?

Check your prediction. rc . Calling the work simply $O(n^2)$ would assume a square matrix or a particular relationship between the dimensions that has not been given.

Count simple work rather than seconds

Our basic model treats an operation on a fixed-width integer, an assignment, a comparison, and access to one valid array element as constant-cost steps. Different steps can take different real times; we are estimating growth, not claiming equal processor timings.

Large string operations are not automatically one step just because they occupy one line. Copying a long string handles many characters. Input and output also have a cost. We will add those costs when discussing whole programs.

This kind of operation model lets us compare algorithms without tying the explanation to one particular processor or stopwatch result.^[3-13]

The input's numeric value is not always its encoded length

A loop from 1 to the value x makes x iterations, but the decimal text used to write x has only about $\log_{10} x$ digits. In this chapter, “ $O(x)$ for summing up to x ” is explicitly a bound in the *numeric value*, under the fixed-width arithmetic model. It is not a claim of linear time in the number of input digits.

3.15 Big-O: describe how the work grows

From an operation count to a growth rate

Suppose a method needs $3n + 7$ simple operations. Doubling a large n roughly doubles the work. We call the growth **linear** and write $O(n)$. We ignore the constant multiplier and the smaller additive term when describing its asymptotic upper bound.

Big-O gives an upper bound on growth for sufficiently large inputs, allowing a fixed constant multiplier. It is not an exact number of operations or seconds.^[3-13]

For example, $3n + 7 \leq 10n$ whenever $n \geq 1$, so $10n$ is a suitable upper bound. For $2n^2 + 5n + 8$, the squared term eventually dominates, giving $O(n^2)$.

Common growth patterns

Notation	Name	Typical reason
$O(1)$	Constant	A fixed amount of work, independent of the size.
$O(\log n)$	Logarithmic	Repeatedly halve a positive size.
$O(\sqrt{n})$	Square-root	Test candidate values only up to $\sqrt{n}$.
$O(n)$	Linear	Visit each of n elements a constant number of times.
$O(n \log n)$	Linearithmic	n repetitions, each with logarithmic work.
$O(n^2)$	Quadratic	About n checks for each of n choices.
$O(n^3)$	Cubic	Three independent dimensions of size n .
$O(2^n)$	Exponential	Repeated branching can create exponentially many calls.

These are patterns to derive, not labels to attach by counting visible loops. Later examples show why two nested loops can still have only linear total work.

$O(1)$ does not mean exactly one operation

A calculation with 30 fixed arithmetic steps is still $O(1)$. An operation independent of n can also have a large constant cost. Conversely, $O(n)$ does not mean exactly n processor instructions.

An upper bound and a tight bound are different

A linear function is also bounded above by a suitable quadratic function, so saying it is $O(n^2)$ can be technically true but uninformative. When classifying code, aim for the **tightest simple bound you can justify**.

$\Theta(n)$, read “Theta of n ,” says that the growth is bounded both above and below by constant multiples of n . $\Omega(n)$ describes a lower bound. You do not need complicated formal proofs here; these symbols distinguish a tight description from just an upper limit.^[3-13]

Worst case is a choice of inputs, not the meaning of O

For linear search, a first-element match needs one comparison, while an absent target needs n comparisons. The helper has $O(1)$ **best-case** time and $O(n)$ **worst-case** time. Big- O itself does not mean “worst case”; we explicitly choose the worst case by default in this chapter.

An average-case claim needs assumptions about how inputs are distributed. Do not assume a target is equally likely to appear in every position unless that model has been stated.

Pause and predict – 15

A method does $7n + 20$ steps. Is $O(n)$ a reasonable tight growth description? Does it imply that the method takes exactly n milliseconds?

Check your prediction. Yes to the growth description, no to the time claim. The amount of work grows linearly, but milliseconds depend on the operations, implementation, and machine.

3.16 Single loops and sequential blocks

Count the body, then account for loop control

E19 reads $0 \leq n \leq 10^6$. It counts visits rather than doing a task with an array, so the relationship between the loop bound and the output is visible.

Complete program E19 – make the iteration count visible

```
#include <iostream>

long long countVisits(int n) {
    long long visits = 0;
    for (int i = 0; i < n; ++i) { ++visits; }
    return visits;
}

int main() {
    int n = 0;
    std::cin >> n;
    std::cout << countVisits(n) << '\n';
    return 0;
}
```

Sample input

5

Sample output

5

For $n = 5$, the body sees $i = 0, 1, 2, 3, 4$: five visits. The condition is checked six times, including the final false check. There are also five increments and one initialization. Adding these costs still gives linear growth.

The return and initialization take time even when $n = 0$. Big-O is an asymptotic description; writing $O(n)$ for this helper is not a statement that the empty case takes literally zero time.

A fixed step size does not change linear growth

Analysis fragment – inside a function; visits starts at zero:

```
for (int i = 0; i < n; i += 3) {
    ++visits;
}
```

For $n = 8$, the counter values are 0, 3, 6: three visits. For nonnegative n , the count is $\lceil n/3 \rceil$, where the ceiling means rounding upward. Dividing by a fixed 3 changes the constant factor, not the $O(n)$ growth.

Two loops after each other: add their costs

Analysis fragment – inside a function:

```
for (int i = 0; i < n; ++i) { ++visits; }  
for (int j = 0; j < n; ++j) { ++visits; }
```

The bodies run $n + n = 2n$ times. The result is $O(n)$, not $O(n^2)$. The second loop does not run inside each iteration of the first one.

If the first length is n and the second is m , write $O(n + m)$. Without a relationship between the sizes, dropping one of them would lose information.

Different sequential growth rates

A linear scan followed by a quadratic pair check costs $O(n + n^2) = O(n^2)$. Keep the dominant growth term, but first make sure the two blocks really use the same size parameter. $O(n + m^2)$ does not simplify to $O(n^2)$ when n and m are independent.

Conditionals: include the condition and the executed branch

For a worst-case analysis, bound the cost of the condition plus the most expensive branch that a valid input can execute. Do not add two mutually exclusive branch costs as though both must run. A condition can itself call a costly function, so it is not always constant-time.

Pause and predict – 16

One loop makes n visits and a later loop makes $5n$ visits. What is the total growth, and why?

Check your prediction. $O(n)$. The total is $6n$ visits plus loop overhead: the costs add because the loops are sequential.

3.17 Nested loops and dependent bounds

Independent loops: multiply repetitions

Analysis fragment – inside a function; body work is constant:

```
for (int row = 0; row < n; ++row) {
    for (int col = 0; col < m; ++col) {
        ++visits;
    }
}
```

The inner loop starts from zero for each row. Each of n rows contributes m visits, so the body count is nm . With positive m , the total time is $O(nm)$. Allowing $m = 0$, a precise all-boundaries expression includes the outer-loop cost: $O(1 + n + nm)$.

For a square n -by- n scan, this is $O(n^2)$. For an inner bound fixed at 5, it is only $5n$ body visits and therefore $O(n)$.

A triangular loop still has quadratic growth

E20 counts pairs of *different positions* (i, j) with $i < j$, for $0 \leq n \leq 1000$. This avoids counting both (i, j) and (j, i) , and never pairs a position with itself.

Complete program E20 – count each unordered pair once

```
#include <iostream>

long long countPairs(int n) {
    long long visits = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) { ++visits; }
    }
    return visits;
}

int main() {
    int n = 0;
    std::cin >> n;
    std::cout << countPairs(n) << '\n';
    return 0;
}
```

Sample input	Sample output
5	10

Trace the inner-loop lengths

For $n = 5$, the first position chooses among four later positions, then three, then two, then one, then zero:

$$4 + 3 + 2 + 1 + 0 = 10.$$

For general n , the count is

$$(n - 1) + (n - 2) + \cdots + 1 + 0 = \frac{n(n - 1)}{2}.$$

The factor $1/2$ does not remove the squared term. The total time is $O(n^2)$, and the pair counter needs only $O(1)$ extra space.

How to derive the formula rather than memorize it

Each position could be paired with $n - 1$ other positions. That gives $n(n - 1)$ *ordered* choices. Every unordered pair appears exactly twice in those choices, so divide by two.

The answer counts work; it does not require storing every pair. The counter and loop variables are reused, so quadratic time does not automatically imply quadratic memory.

Pause and predict – 17

If a triangular loop visits about half as many pairs as a full n -by- n loop, does its growth become $O(n)$?

Check your prediction. No. Roughly half of n^2 still grows quadratically. The constant factor improves, but the growth class remains $O(n^2)$.

3.18 Halving, doubling, and square-root work

Halving reaches a small value quickly

E21 reads $1 \leq n \leq 10^{18}$, repeatedly divides its local value by 2 using integer division, and counts the steps until the value is at most 1.

Complete program E21 – count repeated halving

```
#include <iostream>

int halvingSteps(long long value) {
    int steps = 0;
    while (value > 1) {
        value /= 2;
        ++steps;
    }
    return steps;
}

int main() {
    long long n = 0;
    std::cin >> n;
    std::cout << halvingSteps(n) << '\n';
    return 0;
}
```

Sample input

20

Sample output

4

Step	Value before division	Value afterwards
1	20	10
2	10	5
3	5	2
4	2	1

The count is $\lfloor \log_2 n \rfloor$. The floor means rounding down. You can understand this without a logarithm button: $\log_2 n$ asks how many factors of 2 are needed to reach n . For powers of two, the relationship is exact: $2^{10} = 1024$, so 1024 can be halved ten times to reach 1.

The base changes a factor, not the growth class

Dividing by any fixed factor greater than 1 gives logarithmic growth, although the exact number of steps changes. For asymptotic comparisons we usually write $O(\log n)$ without a base. Subtracting 2 each time is different: that removes only a fixed amount and takes linear work in n .

Doubling is the same growth pattern in the other direction

E22 reads $1 \leq n \leq 10^9$ and counts powers of 2 not exceeding n . Its sequence begins at 1, not zero.

Complete program E22 – powers of two up to n

```
#include <iostream>

int doublingVisits(int n) {
    int visits = 0;
    for (long long value = 1; value <= n; value *= 2) {
        ++visits;
    }
    return visits;
}

int main() {
    int n = 0;
    std::cin >> n;
    std::cout << doublingVisits(n) << '\n';
    return 0;
}
```

Sample input	Sample output
20	5

The visited values are 1, 2, 4, 8, 16, so the count is $\lfloor \log_2 n \rfloor + 1$. The extra 1 does not change $O(\log n)$. The `long long` counter and stated bound make the final doubling safe; unrestricted doubling can overflow, and doubling zero would never make progress.

N repetitions of logarithmic work

E23 accepts $1 \leq n \leq 10000$. Each outer iteration starts a *fresh* halving process at n . The output deliberately counts body visits, not elapsed seconds.

Complete program E23 – an $n \log n$ pattern

```

#include <iostream>

long long countWork(int n) {
    long long visits = 0;
    for (int i = 0; i < n; ++i) {
        int value = n;
        while (value > 1) {
            value /= 2;
            ++visits;
        }
    }
    return visits;
}

int main() {
    int n = 0;
    std::cin >> n;
    std::cout << countWork(n) << '\n';
    return 0;
}

```

Sample input

8

Sample output

24

For $n = 8$, there are eight outer iterations and three halvings in each. The total is 24. In general it is $n \lfloor \log_2 n \rfloor$ body visits, with total time $O(n \log n)$ for growing n . The $n = 1$ case still has ordinary loop overhead despite zero inner visits.

Square-root work comes from a different stopping condition

For positive x , there are about $\sqrt{x}$ positive candidate integers satisfying $d^2 \leq x$. A loop testing such candidates has $O(\sqrt{x})$ iterations, provided each iteration costs a constant amount.

Later, the prime-checking example uses $d \leq x / d$ with positive d , avoiding an overflowing product $d * d$. That stopping rule is about the square root of the numeric value x , not the number of digits in its input.

Pause and predict – 18

Which update is logarithmic in a positive n : repeatedly subtracting 2 or repeatedly dividing by 2 until the value is small?

Check your prediction. Dividing by 2. Subtraction needs about $n/2$ iterations, while division removes a constant fraction of the remaining value each time.

3.19 Count total progress, not only the loop shape

An inner counter that is not reset

E24 reads $0 \leq n \leq 10^6$. The variable `j` is created before the outer loop and can advance only from 0 to `n` across the whole function.

Complete program E24 – nested syntax, linear total work

```
#include <iostream>

long long countWork(int n) {
    int j = 0;
    long long visits = 0;
    for (int i = 0; i < n; ++i) {
        while (j < n) {
            ++j;
            ++visits;
        }
    }
    return visits;
}

int main() {
    int n = 0;
    std::cin >> n;
    std::cout << countWork(n) << '\n';
    return 0;
}
```

Sample input	Sample output
5	5

On the first outer iteration, `j` advances all the way to `n`. On later outer iterations, `j < n` is already false. There are n inner body visits *in total*, not n visits per outer iteration. Adding the n outer iterations gives $O(n)$ time.

This is a counting demonstration, not a claim that the outer loop is useful for this toy task. Real two-index algorithms often use the same principle: one index never moves backwards, so it can advance only a limited total distance.

Resetting the counter changes the analysis

Moving `int j = 0;` inside the outer loop makes a fresh full inner scan for every outer iteration. The work becomes $n \times n$, or $O(n^2)$. A one-line change in variable placement can change the growth rate.

Different inner lengths must be added

Suppose an outer value takes $1, 2, 4, 8, \dots$, and the inner loop makes that many visits. If the largest outer value is $2^k \leq n$, the total is

$$1 + 2 + 4 + \dots + 2^k = 2^{k+1} - 1 < 2n.$$

There are logarithmically many outer iterations, but the inner work is not n in all of them. The tight growth is $O(n)$, not $O(n \log n)$. Drill 5 asks you to trace this pattern yourself.

Early exits do not automatically make the worst case small

In `firstIndex`, an early match can be cheap. An absent target still forces the scan to examine every element. In a pair-search algorithm, finding a pair early does not rule out a valid input with no matching pair, where every candidate must be examined.

Pause and predict – 19

A pointer-like index advances at most n times over an entire outer loop and never resets. Should you charge n advances to every outer iteration?

Check your prediction. No. Charge the advances where they actually happen, then add their total. If the outer loop also has n constant-cost iterations, the combined work is linear.

3.20 Function calls and hidden string costs

A function call is not automatically $O(1)$

The call `sumArray(a, n)` occupies one source-code expression but performs n additions. Calling it n times with the same length means n^2 element visits. A function name hides implementation detail from the caller's reading, not from the program's cost.

E25 reads $0 \leq n \leq 1000$ and values in $[-10^6, 10^6]$. Both methods compute n times the array's sum. The first recomputes that sum repeatedly; the second stores it once.

Complete program E25 – recompute or reuse an unchanged result

```
#include <iostream>

long long sumArray(const int a[], int n) {
    long long sum = 0;
    for (int i = 0; i < n; ++i) { sum += a[i]; }
    return sum;
}

long long repeatedTotal(const int a[], int n) {
    long long total = 0;
    for (int i = 0; i < n; ++i) { total += sumArray(a, n); }
    return total;
}

long long cachedTotal(const int a[], int n) {
    long long sum = sumArray(a, n);
    long long total = 0;
    for (int i = 0; i < n; ++i) { total += sum; }
    return total;
}

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    std::cout << repeatedTotal(a, n) << '\n';
    std::cout << cachedTotal(a, n) << '\n';
    return 0;
}
```

Sample input

Sample output

3	15
2 -1 4	15

Analyze each helper separately

`repeatedTotal` makes n calls to an $O(n)$ helper, so its time is $O(n^2)$. `cachedTotal` does one $O(n)$ sum and then one $O(n)$ loop: $O(n + n) = O(n)$.

Reusing the sum is correct because neither the array nor its length changes during those repetitions. The final loop could even become `return n * sum;`; the initial array scan would still make the method $O(n)$.

The demonstration program calls both versions. Its overall runtime is therefore dominated by the slow version and remains $O(n^2)$. Do not label the whole demonstration $O(n)$ merely because one of its helpers improved.

Parameter preparation can do work before the body starts

Calling `inspect(std::string text)` with an existing string of length L creates a string value for the parameter. Budget linear copying work and $O(L)$ storage for the independent content. A body that looks at only the first character does not make that whole call constant-time. A read-only reference avoids that full parameter copy.^{[3-10][3-21]}

Calling such a value-taking helper q times can contribute $O(qL)$ time. If the calls are sequential and each copy is released when its call ends, the peak extra storage can remain $O(L)$, rather than $O(qL)$.

Pause and predict – 20

A helper scans n values. A caller invokes it n times without reusing the result. Does splitting the scan into another function make the total linear?

Check your prediction. No. The caller still requests n scans of n values, giving $O(n^2)$ total work.

A few library operations from Session 2

Operation	Cost to keep in mind
<code>text.size()</code> , <code>text.empty()</code>	Constant time; these do not scan all the characters. ^[3-14]
Copy an existing string	Linear in the copied length under the library's container requirements. ^{[3-10][3-21]}
<code>text.substr(pos, count)</code>	Creates a new string from the selected characters; budget work and storage proportional to the copied substring. ^{[3-15][3-10]}
Compare two strings	Can examine a long common prefix; do not assume constant time. ^[3-16]
<code>std::sort</code> on n integers	$O(n \log n)$ comparisons; an optional Session 2 tool, not a hidden constant-time rearrangement. ^[3-17]

For a substring made from an existing lvalue string, a copied length of zero still has constant object-handling overhead. For string comparison, identical length- L strings can require $O(L)$ work.

Input and output belong in a whole-program analysis

A helper that checks one already-stored character can take $O(1)$, while the program that first reads a line of L characters needs work to read that line. Similarly, printing n bounded-size integers cannot be free just because all output appears in one loop or helper.^[3-22]

3.21 Space complexity: count memory that is alive together

Time and memory measure different resources

Space complexity describes how required storage grows. A loop may run a million times while reusing the same two counters. It then has substantial time cost but only constant extra storage.

Auxiliary space means the extra working memory beyond the supplied input. **Total storage** also counts stored input; when relevant, say whether output storage is included. State the convention rather than silently switching between them.

For `sumArray(a, n)`, the input array already exists. The helper adds a sum and a counter, so its auxiliary space is $O(1)$. A program that stores n input integers uses linear input storage even though the helper itself needs only constant extra space.

Common memory patterns

What the algorithm keeps	Growth of that storage
A fixed number of scalar variables	$O(1)$.
An input sequence of n fixed-size values	$O(n)$ input storage.
An additional copy of that sequence	$O(n)$ auxiliary storage.
A table with r rows and c columns	$O(rc)$ cells.
Two sequential calls, each needing an n -cell temporary buffer	$O(n)$ peak extra space when the buffers do not coexist.
n pending recursive calls, each with constant-size local state	$O(n)$ call-stack space in the ordinary model.

The recursion row is explained in the optional recursion section. It is not the same as the `std::stack` container scheduled for Session 4.

Fixed capacity is not the same as active length

Our complete programs use declarations such as `int a[1000]{}`; to stay within the array material already taught. This allocates 1000 integers even when the input length is 3. The actual storage is $O(C)$ in the declared capacity C , and initialization with `{}` also touches the allocated elements.

For a literal capacity of 1000, that allocation is constant with respect to a varying active length n . When describing the general array algorithm with a capacity chosen proportional to the required n , we instead speak of $O(n)$ input storage. Neither description means an array automatically shrinks to its active length.

The practice solutions report **helper auxiliary space** separately from their input wrappers. This prevents a copied string, a fixed-capacity input array, and a small local counter from being mixed into an unexplained single label.

Convert a storage plan to approximate bytes

On a machine where `int` occupies 4 bytes, one million integers occupy 4,000,000 bytes for their elements alone. A 1000-by-1000 matrix of 8-byte `long long` elements occupies 8,000,000 bytes. These sizes are assumptions about a machine, not universal C++ requirements.

One decimal MB is 10^6 bytes; one MiB is 2^{20} bytes. Check the judge's stated unit, leave room for other storage, and remember that container bookkeeping and runtime overhead also consume memory.

Peak memory, not the sum of all allocations over time

If a function makes and discards a 1000-character copy in each of 100 calls, it has processed 100000 copied characters overall. That is a time-related count. It does not mean 100000 copied characters were all stored simultaneously. If those copies are kept in a collection instead of discarded, the space analysis changes.

Pause and predict – 21

An algorithm uses three loop counters and a running total while checking every pair of n elements. Must its extra space be $O(n^2)$?

Check your prediction. No. The pair checks can take $O(n^2)$ time while the reused counters and total use only $O(1)$ auxiliary space.

3.22 Constraints, time limits, and multiple test cases

Use the largest permitted input, not only the sample

Once you derive a growth rate, substitute the maximum input size to estimate the scale of the work. This is a plausibility check, not a guaranteed conversion from operations to seconds.

Growth term	$n = 10$	$n = 1000$	$n = 100000$
n	10	1000	100000
n^2	100	1000000	10000000000
n^3	1000	1000000000	10^{15}
$\lceil \log_2 n \rceil$	4	10	17
$n \lceil \log_2 n \rceil$	40	10000	1700000

These are values of the displayed growth terms, not exact instruction counts. A ten-billion-check pair scan should prompt a different plan or a much stronger justification; the fact that its sample finishes instantly is not useful evidence at that scale.

Do not memorize a universal operations-per-second rule

Integer addition, division, allocation, string comparison, and output have different costs. Optimization settings, hardware, input format, and the judge environment also matter. First reduce obviously excessive work. Then test a realistic release build with large valid inputs when performance is close to the limit.

The counter demonstrations in this chapter are for *tracing*, not benchmarking. An optimizing compiler may replace a simple counting loop with a direct expression. A stopwatch on that toy program does not refute the algorithmic source-level count.

Add the work across all test cases

For case lengths $n_1, n_2, \dots, n_t$, a linear scan per case costs

$$O(t + n_1 + n_2 + \dots + n_t).$$

The t term includes constant per-case work and output, which still exist for empty cases. Writing $O(tn)$ is appropriate as an upper bound when *every* case is bounded by the same n , but can be much less informative when the statement limits the sum of lengths.

For E16, the total length is at most 100000. There cannot be 1000 separate cases of length 100000 under that same constraint. Sum the actual or permitted total work rather than combining incompatible maxima.

A total-size limit does not make every algorithm linear

If each case uses a quadratic method, total work is proportional to

$$n_1^2 + n_2^2 + \cdots + n_t^2.$$

With total size $S = \sum n_i$, this can be as large as S^2 when one case holds the whole size. Splitting size across cases does not automatically make a quadratic algorithm safe. A separate bound $n_i \leq B$ gives $\sum n_i^2 \leq B \sum n_i = BS$, which can be a more useful estimate.

Input, processing, output, and initialization

A full runtime estimate includes reading input, preparing arrays, running helpers, and producing output. Resetting a capacity- C array for every case can contribute $O(tC)$, even if many active lengths are small. Streaming a running total avoids input storage when you do not need to revisit the values.

Pause and predict – 22

There are at most 1000 cases, but the sum of their lengths is at most 200000. A helper scans each case once. Is 200 million element visits forced by these bounds?

Check your prediction. No. The total scan visits are at most 200000. Add the per-case overhead, but do not assign the maximum total length independently to every case.

3.23 Worked comparisons: improve the algorithm, not the formatting

Problem A: sum the integers from 1 to n

For $0 \leq n \leq 10^6$, compute $1 + 2 + \dots + n$. One method adds each number. Another uses the sum formula. Both are valid under the stated bounds.

Complete program E26 – compare a loop with a formula

```
#include <iostream>

long long sumLoop(long long n) {
    long long total = 0;
    for (long long i = 1; i <= n; ++i) { total += i; }
    return total;
}

long long sumFormula(long long n) {
    return n * (n + 1) / 2;
}

int main() {
    long long n = 0;
    std::cin >> n;
    std::cout << sumLoop(n) << '\n';
    std::cout << sumFormula(n) << '\n';
    return 0;
}
```

Sample input

5

Sample output

15

15

Why the formula works. Write the sum forwards and backwards. Each aligned pair contributes $n + 1$, and there are n pairs. This counts the sum twice, giving $2S = n(n + 1)$, so $S = n(n + 1)/2$. At $n = 0$, the empty sum is zero, matching the formula.

Time and space. `sumLoop` takes $O(n)$ time; `sumFormula` takes $O(1)$ fixed-width arithmetic operations. Both use $O(1)$ auxiliary space. The *combined demonstration* still takes $O(n)$ because it runs the loop as well as the formula.

Safety. The largest intermediate product in this version is $10^6(10^6 + 1)$, which fits in the tested `long long`. A faster formula with overflowing intermediate arithmetic is not a correct replacement. Wider permitted inputs would require a renewed bound check.

Problem B: decide whether a number is prime

A prime is an integer greater than 1 whose only positive divisors are 1 and itself. A direct method can try every candidate divisor from 2 to $x - 1$. We can do less work without losing correctness.

If $x = ab$ is composite and both a and b were greater than $\sqrt{x}$, their product would exceed x . Therefore at least one divisor of a composite number lies at or below its square root. Test that smaller range.

E27 accepts $-10^9 \leq x \leq 10^9$. Values below 2 immediately return false. All divisions in the loop use a positive divisor.

Complete program E27 – stop at the square-root boundary

```
#include <iostream>

bool isPrime(int value) {
    if (value < 2) { return false; }
    for (int divisor = 2; divisor <= value / divisor; ++divisor) {
        if (value % divisor == 0) { return false; }
    }
    return true;
}

int main() {
    int n = 0;
    std::cin >> n;
    if (isPrime(n)) { std::cout << "YES\n"; }
    else { std::cout << "NO\n"; }
    return 0;
}
```

Sample input

49

Sample output

NO

Trace 49. Candidates 2 through 6 fail to divide it. Candidate 7 must still be checked because $7 \leq 49/7$. The remainder is zero, so the helper returns false. A strict $<$ bound would miss this square-root divisor.

Trace 2. The first candidate would be 2, but $2 \leq 2/2$ is false. There is no candidate divisor in the required range, so the final return is true.

Time and space. For $x \geq 2$, worst-case time is $O(\sqrt{x})$, compared with $O(x)$ candidate testing in the direct method. Extra space is $O(1)$. Some composite values return much earlier, but a worst-case bound must allow a prime input.

Problem C: check whether text is a palindrome

Session 2 compared characters at the left and right ends. Putting that code into `bool isPalindrome(const std::string& text)` preserves its idea while making it reusable.

Comparing inward uses at most $\lfloor L/2 \rfloor$ character pairs and only two indices: $O(L)$ worst-case time and $O(1)$ auxiliary space. Building a reversed copy and comparing it also uses linear work, but stores another length- L sequence. The time class may match while the memory costs differ.

The complete read-only solution appears in Practice 11. Use its `const` reference interface rather than paying for a copy merely to inspect the text.

Pause and predict – 23

Why does the prime test use `divisor <= value / divisor`, including equality, rather than a strict inequality?

Check your prediction. A perfect square can have its first relevant factor exactly at the square root. For 49, the divisor 7 must be tested. The division form also avoids forming a potentially overflowing squared product.

3.24 Optional extension: recursion and the call stack

A preview, not a prerequisite

Recursion means a function calls itself on a smaller case. The core practice uses ordinary helpers and loops; return to this section after those are comfortable. Advanced recursion techniques and recurrence-solving methods are outside this session.

Two ingredients: a stopping case and a smaller problem

For nonnegative n , $n! = n \times (n - 1)!$, with $0! = 1$. This gives a recursive definition. The **base case** answers directly without another recursive call. The **recursive case** reduces the argument so execution moves toward that base case.

E28 permits $0 \leq n \leq 20$, so the result fits in the tested 64-bit `long long`. These limits also keep the recursion shallow.

Complete program E28 – one smaller call per level

```
#include <iostream>

long long factorial(int n) {
    if (n == 0) { return 1; }
    long long smaller = factorial(n - 1);
    return n * smaller;
}

int main() {
    int n = 0;
    std::cin >> n;
    std::cout << factorial(n) << '\n';
    return 0;
}
```

Sample input

4

Sample output

24

Calls go down; results come back up

For `factorial(3)`, the pending work is:

```
factorial(3) waits for 3 * factorial(2)
factorial(2) waits for 2 * factorial(1)
factorial(1) waits for 1 * factorial(0)
factorial(0) returns 1
factorial(1) returns 1
factorial(2) returns 2
factorial(3) returns 6
```

Each call has its own n and local state. The **call stack** is a useful model of this pending work: the newest unfinished call must complete before its caller can use the result. C++ supports recursive calls; the exact machine implementation is not a language-mandated stack layout.^{[3-2][3-6]}

Time. There are $n + 1$ calls including the base case, each with constant extra work, so the algorithm has $O(n)$ time in n .

Space. Up to $n + 1$ calls are pending together, giving $O(n)$ auxiliary call-stack space in the ordinary implementation model. An iterative factorial can reuse a running product and needs only $O(1)$ auxiliary space.

Before and after a recursive call are different moments

E29 reads $0 \leq n \leq 20$. It prints a number before the smaller call and again after that smaller call returns.

Complete program E29 – see the return journey

```
#include <iostream>

void downAndUp(int n) {
    if (n == 0) { return; }
    std::cout << n << '\n';
    downAndUp(n - 1);
    std::cout << n << '\n';
}

int main() {
    int n = 0;
    std::cin >> n;
    downAndUp(n);
    return 0;
}
```

Sample input

3

Sample output

3
2
1
1
2
3

The second `std::cout` is not skipped forever. It waits until the smaller call has finished. With $n = 0$, the base case returns immediately and the program produces no output at all.

There are $n + 1$ calls and $2n$ printed lines, so the time and ordinary call-stack space are both $O(n)$ in this bounded-size integer output model.

Two recursive calls can repeat the same subproblem

E30 implements the definition $F(0) = 0$, $F(1) = 1$, and $F(n) = F(n - 1) + F(n - 2)$. Its input is intentionally restricted to $0 \leq n \leq 25$. It demonstrates repeated work, not an efficient general Fibonacci method.

Complete program E30 – a deliberately inefficient valid method

```
#include <iostream>

int fibonacci(int n) {
    if (n <= 1) { return n; }
    int left = fibonacci(n - 1);
    int right = fibonacci(n - 2);
    return left + right;
}

int main() {
    int n = 0;
    std::cin >> n;
    std::cout << fibonacci(n) << '\n';
    return 0;
}
```

Sample input

6

Sample output

8

For `fibonacci(5)`, one branch computes `fibonacci(4)`, which itself computes `fibonacci(3)`. The other top-level branch computes `fibonacci(3)` again. That overlapping work expands rapidly.

A simple time upper bound is $O(2^n)$: each level can branch into at most two children and there are at most n reductions along a path. This is a valid *upper bound*, not a claim that the exact call count is 2^n or that $\Theta(2^n)$ is tight for this implementation.

Only one branch executes at a time. The maximum chain of pending calls has linear depth, so the ordinary auxiliary call-stack space is $O(n)$, not $O(2^n)$. A loop that remembers the two most recent Fibonacci values avoids this repeated branching; more advanced reuse techniques come later.

Common recursion failures

A missing base case, a recursive argument that does not get closer to the base case, or an input outside the stated domain can prevent termination. Even a mathematically terminating recursion can exhaust available call-stack resources when the depth is large. Do not assume that an input of one million is safe just because one call works.

Pause and predict – 24

In E28, would `factorial(n)` instead of `factorial(n - 1)` make progress toward the base case for positive n ?

Check your prediction. No. The same positive argument would be called again and again. The smaller argument is what makes the recursion reach zero.

3.25 Optional extension: passing a two-dimensional array

Keep the row width in the parameter type

The simple one-dimensional array interface used `a[]` and a length. For a two-dimensional built-in array, the inner width is needed to interpret the rows. E31 uses a fixed width of three columns in both the caller's array and the helper's parameter.^{[3-1][3-9]}

Read $0 \leq r \leq 100$, then r rows of exactly three integers each. Each value is in $[-10^9, 10^9]$. Print the sum of all active cells.

Complete program E31 – receive rows of a known width

```
#include <iostream>

const int COLUMNS = 3;

long long sumMatrix(const int a[][COLUMNS], int rows) {
    long long total = 0;
    for (int row = 0; row < rows; ++row) {
        for (int col = 0; col < COLUMNS; ++col) {
            total += a[row][col];
        }
    }
    return total;
}

int main() {
    const int MAX_ROWS = 100;
    int a[MAX_ROWS][COLUMNS]{};
    int rows = 0;
    std::cin >> rows;
    for (int row = 0; row < rows; ++row) {
        for (int col = 0; col < COLUMNS; ++col) {
            std::cin >> a[row][col];
        }
    }
    std::cout << sumMatrix(a, rows) << '\n';
    return 0;
}
```

Sample input

```
2
1 2 3
4 5 6
```

Sample output

```
21
```

What the helper knows and what the caller promises

`COLUMNS` is a shared constant describing the actual row width. `rows` is the active row count. The caller promises that the requested rows exist. The helper visits $3 * \text{rows}$ cells and does not modify them.

For this fixed-width interface, time is $O(r)$ and helper auxiliary space is $O(1)$. More generally, scanning r rows and c active columns takes $O(rc)$ time. The wrapper's fixed allocation is `MAX_ROWS * COLUMNS` integers, not merely the number of active cells.

Do not write an unspecified inner dimension as `int a[] []` in this ordinary parameter declaration, and do not substitute a runtime width into a built-in array type as though variable-length arrays were standard C++23. Later containers offer more flexible storage interfaces.

Pause and predict – 25

A matrix helper expects rows with exactly three stored columns. Can changing only a runtime `columns` variable turn an actual five-column-row array into the same parameter type?

Check your prediction. No. The stored row width is part of the relevant type. The caller's array layout and the helper's interface must agree.

3.26 Debugging functions and checking an analysis

Find which contract was broken

When a helper fails, separate the questions: Did the caller provide valid inputs? Did the helper calculate the right result? Did the caller use that result correctly? A correct search helper can still cause trouble if the caller treats `-1` as a valid array index.

Symptom	First thing to inspect
“Not declared” at a call	Is a matching declaration or definition visible before the call?
A result-producing path has no value	Does every permitted path return a value of the intended kind?
The caller’s value never changes	Was the parameter by value? Was the returned result discarded?
The caller changes unexpectedly	Is there a mutable reference, array-element update, or shared global state?
The first candidate controls every answer	Is a return incorrectly placed inside a loop?
Correct output, but too slow	Count calls and their costs, including string copies and repeated scans.
Unexpected memory growth	Count stored copies and pending recursive calls, not only named variables.

Do not return success before checking all candidates

Wrong helper – for teaching only:

```
bool wrongPrime(int value) {
    if (value < 2) { return false; }
    for (int d = 2; d <= value / d; ++d) {
        if (value % d == 0) { return false; }
        return true; // Wrong: one failed divisor is not enough.
    }
    return true;
}
```

For 9, the first candidate 2 does not divide it, so this wrong version returns true before checking 3. The corrected E27 returns true only after the loop finishes without finding any divisor.

A small analysis checklist

Define the size parameters. Count the repeated body work. Add sequential costs. Multiply only when repetitions genuinely repeat the full inner work. Expand helper calls, include expensive conditions and copies, and check whether counters reset. Finally, count peak extra storage and whole-input/output costs separately.

Use warnings and test boundaries

Compiler warnings can reveal missing returns, suspicious conversions, and shadowed variables. Runtime checking can catch some invalid operations on tested paths, but neither tool replaces a correct contract or proves every input safe.^{[3-19][3-20]}

Try empty and one-element inputs where allowed, equal values, negative values, the first and last matching positions, and cases with no match. For recursion, test the base case first. For time analysis, use maximum permitted sizes on paper before running an obviously excessive loop.

Pause and predict – 26

An absent-target test fails because the caller executes `a[firstIndex(a, n, target)]`. Is the helper necessarily incorrect?

Check your prediction. No. The helper may correctly return -1. The caller must check the result before using it as an index. A contract describes both the helper's promise and the caller's responsibility.

3.27 Practice problems: write a helper and explain its cost

Try these without the solutions open. For each problem, write the helper's contract, its complete program, and a short time/space explanation. Every required interface below is a definition header: add its body, not a semicolon-only declaration.

Problems 1–17 form the core practice. Problem 18 is an optional challenge. All input follows the stated format and bounds. Array examples use a declared capacity; helper-space answers exclude the array already supplied by the caller.

1. Largest of three

Read three integers and print the largest. Write a helper for the larger of two values, then reuse it inside `maximumOfThree`. Do not print inside either helper.

Input and output. Three integers `a b c`, each in $[-10^9, 10^9]$. Output one integer. Equal values are allowed.

Required helper: `int maximumOfThree(int a, int b, int c)`

Sample input	Sample output
--------------	---------------

-8 -2 -5	-2
----------	----

Explained solution: page 76.

2. The last digit

Read a nonnegative integer and print its last decimal digit. Put the calculation in `lastDigit`; let `main` read and print.

Input and output. One integer $0 \leq n \leq 10^{18}$. Output a digit from 0 to 9. The last digit of zero is zero.

Required helper: `int lastDigit(long long n)`

Sample input	Sample output
--------------	---------------

907	7
-----	---

Explained solution: page 77.

3. Sum of decimal digits

Read a nonnegative integer and print the sum of its decimal digits. The helper receives its number by value; its repeated divisions must not change a caller variable.

Input and output. One integer $0 \leq n \leq 10^{18}$. Output one integer. The digit sum of zero is zero.

Required helper: `int digitSum(long long n)`

Sample input	Sample output
--------------	---------------

5074	16
------	----

Explained solution: page 78.

4. Keep a value inside a range

Return `low` when the value is below the range, `high` when it is above the range, and the value itself otherwise. The interval includes both endpoints.

Input and output. Three integers `value` `low` `high`, all in $[-10^9, 10^9]$, with `low` $\leq$ `high`. Output the returned integer.

Required helper: `int clampToRange(int value, int low, int high)`

Sample input	Sample output
--------------	---------------

17 3 10	10
---------	----

Explained solution: [page 79](#).

5. Scale an array in place

Multiply every active array element by `factor`. The helper must change the caller's array, not print a separate answer.

Input and output. Read $0 \leq n \leq 1000$, then n integers, then `factor`. Each value and the factor lie in $[-1000, 1000]$. Print the changed values separated by spaces, or an empty line when $n = 0$.

Required helper: `void scaleArray(int a[], int n, int factor)`

Sample input	Sample output
--------------	---------------

4	-4 6 0 -10
2 -3 0 5	
-2	

Explained solution: [page 80](#).

6. Count matching values

Return how many active array elements equal the target. The helper must not modify the array. Every matching position counts, including duplicates.

Input and output. Read $0 \leq n \leq 1000$, then the n values, then the target. Values and target lie in $[-10^9, 10^9]$. Output one integer.

Required helper: `int countOccurrences(const int a[], int n, int target)`

Sample input	Sample output
--------------	---------------

5	3
4 2 4 4 7	
4	

Explained solution: [page 81](#).

7. First position of the minimum

Return the zero-based index of the smallest array value. If the minimum appears several times, return its first index. Return an index, not the value stored there.

Input and output. Read $1 \leq n \leq 1000$, then n values in $[-10^9, 10^9]$. Output one zero-based index. The input is guaranteed nonempty.

Required helper: `int indexOfMinimum(const int a[], int n)`

Sample input	Sample output
4	1
7 -2 -2 9	

Explained solution: page 82.

8. Reverse an array through a helper

Reverse the active array elements in place. Use a helper that changes the caller's array and only a constant number of extra variables.

Input and output. Read $0 \leq n \leq 1000$, then n integers in $[-10^9, 10^9]$. Output the reversed values separated by spaces; an empty array produces an empty line.

Required helper: `void reverseArray(int a[], int n)`

Sample input	Sample output
5	5 4 9 2 1
1 2 9 4 5	

Explained solution: page 83.

9. Is the sequence nondecreasing?

Decide whether each value is at least the value immediately before it. Equal neighbours are allowed. Return a Boolean and let `main` print YES or NO.

Input and output. Read $0 \leq n \leq 1000$, then n integers in $[-10^9, 10^9]$. Empty and one-element arrays are considered nondecreasing.

Required helper: `bool isNondecreasing(const int a[], int n)`

Sample input	Sample output
4	YES
-2 0 0 8	

Explained solution: page 84.

10. Count digit characters

Read one full line and count the characters from '0' through '9'. Count characters, not numeric values: the text 42 contains two digits.

Input and output. One ASCII line of length $0 \leq L \leq 100000$, excluding its line terminator. Spaces and punctuation may appear. Output one count.

Required helper: `int countDigits(const std::string& text)`

Sample input	Sample output
--------------	---------------

room 42, floor 3	3
------------------	---

Explained solution: page 85.

11. A reusable palindrome check

Return whether a full line reads identically forward and backward. The check is case-sensitive: spaces and punctuation count as ordinary characters. Do not build a reversed copy.

Input and output. One ASCII line with $0 \leq L \leq 100000$. Output YES or NO. Empty and one-character strings are palindromes.

Required helper: `bool isPalindrome(const std::string& text)`

Sample input	Sample output
--------------	---------------

ab ba	YES
-------	-----

Explained solution: page 86.

12. Return an uppercase copy

Return a copy of the input with ASCII lowercase letters changed to uppercase. Preserve all other characters. The caller's original string must remain unchanged.

Input and output. One ASCII line with $0 \leq L \leq 100000$. Output the transformed line, including its spaces. An empty input line produces an empty output line.

Required helper: `std::string uppercaseCopy(std::string text)`

Sample input	Sample output
--------------	---------------

Hello, level 0!	HELLO, LEVEL 0!
-----------------	-----------------

Explained solution: page 87.

13. Return two results through references

Find both the smallest and largest array values. Keep the array read-only and store the two results through reference parameters. In `main`, pass two distinct variables that are not array elements.

Input and output. Read $1 \leq n \leq 1000$, then n values in $[-10^9, 10^9]$. Output the minimum, then the maximum, separated by a space.

Required helper: `void minMax(const int a[], int n, int& minimum, int& maximum)`

Sample input	Sample output
--------------	---------------

4 -6 -1 -9 -3	-9 -1
------------------	-------

Explained solution: page 88.

14. Prime checks across test cases

For each input value, decide whether it is prime. Put the mathematical decision in `isPrime` and one test case's input/output in a separate `void solve()` helper.

Input and output. Read $1 \leq t \leq 100$, then t integers in $[-10^6, 10^6]$. Print YES or NO on a separate line for each value. A prime is an integer at least 2 with exactly two positive divisors.

Required helper: `bool isPrime(int value)`

Sample input	Sample output
--------------	---------------

5	NO
1 2 49 97 -7	YES
	NO
	YES
	NO

Explained solution: page 89.

15. Count index pairs with a given sum

Count pairs of indices (i, j) with $0 \leq i < j < n$ whose values add to the target. Each pair of positions counts separately, even when its values equal another pair's values.

Input and output. Read $0 \leq n \leq 2000$, then n values in $[-10^9, 10^9]$, then a target in $[-2 \times 10^9, 2 \times 10^9]$. Output the pair count.

Required helper: `long long countPairsWithSum(const int a[], int n, long long target)`

Sample input	Sample output
--------------	---------------

4 1 3 2 2 4	2
-------------------	---

Explained solution: page 90.

16. Count words using two helpers

A word is a maximal nonempty run of characters other than the ordinary space character. Count the words in a full line. Write a small `bool isSeparator(char c)` helper and reuse it.

Input and output. One ASCII line with $0 \leq L \leq 100000$. Only ordinary spaces separate words in this task; the line terminator is not part of the input string. Print one integer.

Required helper: `int countWords(const std::string& text)`

Sample input	Sample output
--------------	---------------

<code> read trace test</code>	3
-------------------------------	---

The visible marks represent spaces; type ordinary spaces in the actual input. Explained solution: page 91.

17. Repeated threshold queries

For each query threshold, print how many array values are at least that threshold. Reuse a read-only counting helper. Analyze the whole batch, not only a single helper call.

Input and output. Read $0 \leq n \leq 1000$ and $1 \leq q \leq 1000$, then n values and q thresholds. Values and thresholds lie in $[-10^9, 10^9]$. Output one count per query.

Required helper: `int countAtLeast(const int a[], int n, int threshold)`

Sample input	Sample output
--------------	---------------

4 3	2
2 7 7 -1	3
7 0 8	0

Explained solution: page 92.

18. Optional challenge: keep the order

Decide whether `small` can be obtained from `large` by deleting zero or more characters without reordering those kept. This is a subsequence test, not a substring test: the kept characters need not be adjacent.

Input and output. Two full ASCII lines, first `small`, then `large`; each has length from 0 to 100000. Output YES or NO. Spaces and letter case matter. The empty string is a subsequence of any string.

Required helper: `bool isSubsequence(const std::string& small, const std::string& large)`

Sample input	Sample output
--------------	---------------

ace	YES
abcde	

Explained solution: page 93.

3.28 Complexity drills: count before naming the growth rate

These questions do not require a complete program. Assume fixed-width arithmetic and array access cost $O(1)$, the supplied input is already available, and counters do not overflow. Use $0 \leq n, m \leq 10^6$ unless a drill says otherwise. Give a tight growth rate when possible, plus a short reason. Θ means a tight bound; a matching Big-O bound is acceptable for this beginner exercise.

Drill 1 – A step of three

How many times is `visits` increased? Give the time and auxiliary space.

```
long long visits = 0;
for (int i = 0; i < n; i += 3) { ++visits; }
```

Drill 2 – Two inputs, three consecutive loops

Do not replace both independent lengths by one symbol.

```
long long visits = 0;
for (int i = 0; i < n; ++i) { ++visits; }
for (int j = 0; j < m; ++j) { ++visits; }
for (int k = 0; k < 10; ++k) { ++visits; }
```

Drill 3 – Including the diagonal

Find an exact count for ++visits, then simplify the growth rate.

```
long long visits = 0;
for (int i = 0; i < n; ++i) {
    for (int j = 0; j <= i; ++j) { ++visits; }
}
```

Drill 4 – Two loops, but only five inner visits

Is this quadratic? Explain using the bounds rather than the number of loops.

```
long long visits = 0;
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < 5; ++j) { ++visits; }
}
```

Drill 5 – Growing chunks

Here $1 \leq n \leq 10^6$, so the long long counter stays safe. Find the exact visits for $n = 10$, and the growth rate for general n .

```
long long visits = 0;
for (long long width = 1; width <= n; width *= 2) {
    for (long long j = 0; j < width; ++j) { ++visits; }
}
```

Drill 6 – Is the inner counter reset?

Compare the two fragments. They look similar but do not do the same total work.

```
// Version A
int j = 0;
for (int i = 0; i < n; ++i) {
    while (j < n) { ++j; }
}

// Version B
for (int i = 0; i < n; ++i) {
    int j = 0;
    while (j < n) { ++j; }
}
```

Drill 7 – A helper called many times

An already-read string has length L , where $0 \leq L \leq 100000$; $1 \leq q \leq 1000$. The helper is the read-only two-ended palindrome check from this chapter. Analyze the loop's worst-case time and peak auxiliary space. Assume the source-level calls are actually performed; this is algorithm analysis, not a compiler benchmark.

```
int successful = 0;
for (int query = 0; query < q; ++query) {
    if (isPalindrome(text)) { ++successful; }
}
```

Drill 8 – An almost-empty body hides a copy

An existing string has length $n \geq 1$. Analyze n calls to this helper with that existing string as the argument. Include parameter copying and peak extra storage; do not assume those calls have been optimized away.

```
bool emptyCopy(std::string text) {  
    return text.empty();  
}
```

Drill 9 – Repeated halving

For $1 \leq n \leq 10^{18}$, how many increments occur for $n = 20$? Give the general count and complexity.

```
long long value = n;  
int steps = 0;  
while (value > 1) {  
    value /= 2;  
    ++steps;  
}
```

Drill 10 – Total constraints across test cases

Test case i has length n_i , and its chosen algorithm performs n_i^2 counted visits. The total length is at most $S = 200000$. What is an upper bound on all visits with no other length restriction? How does the bound improve when every $n_i \leq 1000$? Are either of these numbers a promise about seconds?

Drill 11 – Peak memory is not cumulative allocation

Assume an input array stores n integers and a helper creates one separate copy of those n integers, uses a few counters, returns one integer, then releases its copy. The helper is called 20 times sequentially; copies do not survive their calls.

What are the helper's auxiliary-space and total-live-space growth rates? With $n = 10^6$ and 4 bytes per integer, approximately how many bytes are occupied by the two arrays at peak? Ignore allocator and object overhead for this estimate. Do not write a variable-length built-in array as an implementation: this is a memory-accounting scenario.

Drill 12 – Optional: calls over time versus calls still alive

Compare the ordinary recursive factorial helper and the naive recursive Fibonacci helper shown earlier. Factorial calls only the problem of size $n - 1$; Fibonacci calls both $n - 1$ and $n - 2$. Both have constant-size base cases.

Explain why factorial has linear time and linear call depth. Give a valid exponential upper time bound for Fibonacci and explain why its peak live call depth is not exponential. Use the ordinary call-stack model, without assuming a compiler removes recursive calls.

A useful format for every analysis answer

Name the varying input sizes. Count total repetitions or character visits. State what each repetition costs. Finally, count the extra storage that must exist *at the same time*. If you use a worst-case bound, say so.

3.29 Hints before solutions

Use the smallest hint that helps you move forward. After reading a hint, close this page and try the program again before opening its complete solution.

Coding problems 1–9

- 1. Largest of three.** First find the larger of a and b ; compare that result with c .
- 2. The last digit.** Dividing by 10 leaves the last digit as the remainder.
- 3. Sum of decimal digits.** Add the last digit to an accumulator, then remove that digit by dividing by 10.
- 4. Keep a value inside a range.** There are three disjoint cases. Handle the two outside cases with early returns.
- 5. Scale an array in place.** Visit each valid index once and replace $a[i]$ by its product with the factor.
- 6. Count matching values.** Start a counter at zero; increase it exactly when the current element equals the target.
- 7. First position of the minimum.** Start with index zero. Replace the best index only for a strictly smaller value.
- 8. Reverse an array through a helper.** Swap the first and last active elements, then move both ends inward.
- 9. Is the sequence nondecreasing?.** A single neighbour pair with $a[i] < a[i - 1]$ is enough to return false.

Coding problems 10–18

- 10. Count digit characters.** Test both endpoints of the digit range for each character; a read-only string reference avoids a copy.
- 11. A reusable palindrome check.** Compare the two ends, then move both indices inward. One mismatch settles the answer.
- 12. Return an uppercase copy.** Here a by-value string is deliberate: modify the local copy and return it by value.
- 13. Return two results through references.** Initialize both results from the first element, then update each independently.
- 14. Prime checks across test cases.** Reject values below 2. A composite positive integer has a divisor no larger than its square root.
- 15. Count index pairs with a given sum.** For each first index, begin the second index at $i + 1$. Use a wide intermediate sum.
- 16. Count words using two helpers.** Count a word only on the transition from being outside a word to reading a non-space.
- 17. Repeated threshold queries.** One query scans n elements. There are q separate queries, so account for every scan.
- 18. Optional challenge: keep the order.** Scan the larger string once while remembering the next character still needed from the smaller string.

Complexity drills

1. Write the visited indices for $n = 8$: 0, 3, and 6. How many groups of three are needed to cover n positions?
2. Consecutive loops contribute separate costs that add. The fixed loop contributes ten, not another factor of ten.
3. For successive i , the inner lengths are 1, 2, 3, and so on.
4. Five does not grow with n .
5. Add the actual chunk sizes: $1 + 2 + 4 + 8 + \dots$. Do not charge the largest chunk to every iteration when seeking a tight bound.
6. In A, j never returns to zero. In B, it restarts from zero on every outer iteration.
7. A palindrome forces all mirrored pairs to be checked. Multiply that per-call cost by the number of calls, not by the number of parameters.
8. The string is passed by value from an existing lvalue. A tiny body does not make copying all its characters free.
9. For 20, record $20 \rightarrow 10 \rightarrow 5 \rightarrow 2 \rightarrow 1$.
10. Use $\sum n_i^2 \leq (\sum n_i)^2$. Under the extra cap B , also use $n_i^2 \leq Bn_i$.
11. Draw only the arrays that are alive during one call. Calls that have finished do not still own their released copies.
12. Draw a chain for factorial and a branching tree for Fibonacci. Then mark a single root-to-leaf path: that path describes simultaneously unfinished calls.

3.30 Explained practice solutions

The following solutions are complete programs. Their listings are loaded from the same files that were compiled and tested. Attempt a problem first; then compare the idea, contract, and edge cases, not only the spelling of the code.

How to read the cost statements

Unless a paragraph explicitly says *whole program*, its time and auxiliary-space analysis refers to the named helper. The supplied input storage is excluded from auxiliary space. Fixed-capacity wrappers store C elements and zero-initialize them; that adds $O(C)$ storage and initialization work. Reading and printing sequences adds their input/output cost.

Writing $O(n + 1)$ instead of $O(n)$ in an exercise that permits $n = 0$ makes the constant overhead visible. For large n , both describe linear growth. The plus one is not a new complexity class.

A check to make before reading a solution

Can you explain the helper's input, its result or permitted changes, and the smallest valid input? For an array, can you say exactly which indices it may touch? For a string, can you justify whether the parameter is a copy or a read-only reference?

Writing a correct helper is only half the solution

The wrapper must still read the specified input format, pass the correct active length, and print the required output. A correct algorithm with the wrong input order or an unexpected explanation in its output can still fail a contest problem.

Solution 1 – Largest of three

Idea and why it works. The inner call returns a value at least as large as both **a** and **b**. Comparing that result with **c** therefore considers all three inputs. Equality does not require a special case because the value to return is the same.

Complete solution – C++23

```
#include <iostream>

int larger(int a, int b) {
    if (a > b) { return a; }
    return b;
}

int maximumOfThree(int a, int b, int c) {
    return larger(larger(a, b), c);
}

int main() {
    int a = 0, b = 0, c = 0;
    std::cin >> a >> b >> c;
    std::cout << maximumOfThree(a, b, c) << '\n';
    return 0;
}
```

Trace and boundary checks. For **-8 -2 -5**, the inner call gives **-2**, then the outer call keeps it. Test all equal values and a largest value in each of the three positions.

Cost. Two constant-size comparisons give $O(1)$ time and $O(1)$ auxiliary space.

Solution 2 – The last digit

Idea and why it works. For a nonnegative integer, $n \% 10$ is the remainder after removing whole groups of ten. That remainder is exactly its last decimal digit. It is in $[0, 9]$, so converting it to `int` is safe.

Complete solution – C++23

```
#include <iostream>

int lastDigit(long long n) {
    return static_cast<int>(n % 10);
}

int main() {
    long long n = 0;
    std::cin >> n;
    std::cout << lastDigit(n) << '\n';
    return 0;
}
```

Trace and boundary checks. $907 = 90 \times 10 + 7$. Test 0, a one-digit input, a number ending in zero, and 10^{18} .

Cost. One fixed-width remainder calculation uses $O(1)$ time and $O(1)$ auxiliary space under our arithmetic model.

Solution 3 – Sum of decimal digits

Idea and why it works. At each iteration, `sum` contains the total of the digits already removed, and `n` contains the digits not yet processed. Adding `n % 10` and dividing by 10 preserves this description. When no digits remain, the accumulator is the answer.

Complete solution – C++23

```
#include <iostream>

int digitSum(long long n) {
    int sum = 0;
    while (n > 0) {
        sum += static_cast<int>(n % 10);
        n /= 10;
    }
    return sum;
}

int main() {
    long long n = 0;
    std::cin >> n;
    std::cout << digitSum(n) << '\n';
    return 0;
}
```

Trace and boundary checks. For 5074, the removed digits are 4, 7, 0, 5; the partial sums are 4, 11, 11, 16. Zero needs no iterations and correctly returns the initial zero.

Cost. For d decimal digits, the helper takes $O(d)$ time and $O(1)$ auxiliary space. For positive n , $d = \lfloor \log_{10} n \rfloor + 1$.

Solution 4 – Keep a value inside a range

Idea and why it works. A value below the lower endpoint must become `low`; a value above the upper endpoint must become `high`. If neither condition holds, it already belongs to the interval. These cases cover every allowed input.

Complete solution – C++23

```
#include <iostream>

int clampToRange(int value, int low, int high) {
    if (value < low) { return low; }
    if (value > high) { return high; }
    return value;
}

int main() {
    int value = 0, low = 0, high = 0;
    std::cin >> value >> low >> high;
    std::cout << clampToRange(value, low, high) << '\n';
    return 0;
}
```

Trace and boundary checks. For 17 3 10, only the upper-bound case applies. Test equality with each endpoint, negative ranges, and `low == high`.

Cost. At most two comparisons give $O(1)$ time and $O(1)$ auxiliary space.

Solution 5 – Scale an array in place

Idea and why it works. The array parameter accesses the caller's elements. The loop changes each index from 0 through $n - 1$ exactly once; after it finishes, every active value has the required multiplier. The product magnitude is at most 10^6 , which fits the training environment's `int`.

Complete solution – C++23

```
#include <iostream>

void scaleArray(int a[], int n, int factor) {
    for (int i = 0; i < n; ++i) { a[i] *= factor; }
}

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0, factor = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    std::cin >> factor;
    scaleArray(a, n, factor);
    for (int i = 0; i < n; ++i) {
        if (i > 0) { std::cout << ' '; }
        std::cout << a[i];
    }
    std::cout << '\n';
    return 0;
}
```

Trace and boundary checks. Zero values stay zero. A negative factor changes signs. For an empty active range, the loop does nothing and `main` prints only its newline.

Cost. The helper uses $O(n + 1)$ time and $O(1)$ auxiliary space. The wrapper stores a fixed capacity $C = 1000$, using $O(C)$ array storage.

Solution 6 – Count matching values

Idea and why it works. After processing the first i elements, `count` is the number of matches among those elements. The next comparison either adds one matching position or leaves the count unchanged. At the end all active positions have been considered.

Complete solution – C++23

```
#include <iostream>

int countOccurrences(const int a[], int n, int target) {
    int count = 0;
    for (int i = 0; i < n; ++i) {
        if (a[i] == target) { ++count; }
    }
    return count;
}

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0, target = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    std::cin >> target;
    std::cout << countOccurrences(a, n, target) << '\n';
    return 0;
}
```

Trace and boundary checks. The sample matches indices 0, 2, and 3. Test a missing target, an all-matching array, and an empty array; an empty array returns zero.

Cost. The helper performs n comparisons: $O(n + 1)$ time and $O(1)$ auxiliary space. Wrapper array storage is $O(C)$ for its declared capacity.

Solution 7 – First position of the minimum

Idea and why it works. The first element is a valid starting candidate because the contract requires $n \geq 1$. After each step, `best` points to the first minimum in the processed prefix. Updating only on $<$, not $\leq$, preserves the earlier position on a tie.

Complete solution – C++23

```
#include <iostream>

int indexOfMinimum(const int a[], int n) {
    int best = 0;
    for (int i = 1; i < n; ++i) {
        if (a[i] < a[best]) { best = i; }
    }
    return best;
}

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    std::cout << indexOfMinimum(a, n) << '\n';
    return 0;
}
```

Trace and boundary checks. Index 1 becomes `best` when its value is -2 . The equal value at index 2 does not replace it. A one-element array returns zero.

Cost. The helper scans once, taking $O(n)$ time and $O(1)$ auxiliary space. The input array is not copied.

Solution 8 – Reverse an array through a helper

Idea and why it works. Each swap places two values in their final reversed positions. The remaining work is the smaller interval between the two indices. Stop when they meet or cross; an odd-length middle element already belongs in the right place.

Complete solution – C++23

```
#include <iostream>

void reverseArray(int a[], int n) {
    int left = 0, right = n - 1;
    while (left < right) {
        int temporary = a[left];
        a[left] = a[right];
        a[right] = temporary;
        ++left;
        --right;
    }
}

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    reverseArray(a, n);
    for (int i = 0; i < n; ++i) {
        if (i > 0) { std::cout << ' '; }
        std::cout << a[i];
    }
    std::cout << '\n';
    return 0;
}
```

Trace and boundary checks. For an empty array, the signed right index is -1 , so no access occurs. A one-element array also performs no swap. A temporary variable preserves the first value during each swap.

Cost. There are $\lfloor n/2 \rfloor$ swaps: $O(n + 1)$ time and $O(1)$ auxiliary space.

Solution 9 – Is the sequence nondecreasing?

Idea and why it works. A decrease between neighbours violates the required order immediately, so it is safe to return false at the first one. If all neighbour pairs pass, the whole sequence is nondecreasing. Starting at index 1 makes $i - 1$ valid.

Complete solution – C++23

```
#include <iostream>

bool isNondecreasing(const int a[], int n) {
    for (int i = 1; i < n; ++i) {
        if (a[i] < a[i - 1]) { return false; }
    }
    return true;
}

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    if (isNondecreasing(a, n)) { std::cout << "YES\n"; }
    else { std::cout << "NO\n"; }
    return 0;
}
```

Trace and boundary checks. An empty or one-element input has no offending pair, so the final return is true. Test equality, a decrease at the beginning, and a decrease only at the end.

Cost. The helper's worst-case time is $O(n + 1)$, its best case can be $O(1)$, and auxiliary space is $O(1)$. The wrapper still reads every supplied element.

Solution 10 – Count digit characters

Idea and why it works. Each character contributes one exactly when it is a decimal digit. The counter therefore accumulates the number of digit positions, independently of whether adjacent digits form one number. `getline` keeps the spaces in the supplied line.

Complete solution – C++23

```
#include <iostream>
#include <string>

int countDigits(const std::string& text) {
    int count = 0;
    int n = static_cast<int>(text.size());
    for (int i = 0; i < n; ++i) {
        if ('0' <= text[i] && text[i] <= '9') { ++count; }
    }
    return count;
}

int main() {
    std::string text;
    std::getline(std::cin, text);
    std::cout << countDigits(text) << '\n';
    return 0;
}
```

Trace and boundary checks. The sample contains 4, 2, and 3. An empty line or a line containing only letters returns zero. The length bound makes the conversion to `int` safe here.

Cost. The helper takes $O(L + 1)$ time and $O(1)$ auxiliary space. Storing the input string itself needs $O(L + 1)$ space.

Solution 11 – A reusable palindrome check

Idea and why it works. A palindrome requires every mirrored pair to match. A mismatch is sufficient for false. When the indices meet or cross, every required pair has matched, so the answer is true. The original string is shared read-only rather than copied.

Complete solution – C++23

```
#include <iostream>
#include <string>

bool isPalindrome(const std::string& text) {
    int left = 0;
    int right = static_cast<int>(text.size()) - 1;
    while (left < right) {
        if (text[left] != text[right]) { return false; }
        ++left;
        --right;
    }
    return true;
}

int main() {
    std::string text;
    std::getline(std::cin, text);
    if (isPalindrome(text)) { std::cout << "YES\n"; }
    else { std::cout << "NO\n"; }
    return 0;
}
```

Trace and boundary checks. The sample's middle space needs no comparison. **Aa** is not a palindrome under the case-sensitive rule. The empty string has right index -1 and performs no access.

Cost. Worst-case helper time is $O(L + 1)$, with at most $\lfloor L/2 \rfloor$ comparisons and $O(1)$ auxiliary space. Reading the full input remains linear even when the helper exits early.

Solution 12 – Return an uppercase copy

Idea and why it works. The parameter is a separate string initialized from `original`. For each lowercase ASCII letter, subtracting `'a'` gives its alphabet position; adding `'A'` gives the matching uppercase code. Characters outside that range are not changed.

Complete solution – C++23

```
#include <iostream>
#include <string>

std::string uppercaseCopy(std::string text) {
    int n = static_cast<int>(text.size());
    for (int i = 0; i < n; ++i) {
        if ('a' <= text[i] && text[i] <= 'z') {
            text[i] = static_cast<char>(text[i] - 'a' + 'A');
        }
    }
    return text;
}

int main() {
    std::string original;
    std::getline(std::cin, original);
    std::string result = uppercaseCopy(original);
    std::cout << result << '\n';
    return 0;
}
```

Trace and boundary checks. The original and transformed values are separate logical strings. Returning a string by value is safe; returning a reference to the local parameter would not be. Test mixed case, punctuation, and an empty line.

Cost. Copying and scanning take $O(L + 1)$ time. Including the returned text, additional storage is $O(L + 1)$. The scan itself uses only $O(1)$ counters; it is not the only storage cost.

Solution 13 – Return two results through references

Idea and why it works. After a prefix has been processed, the two output variables describe that prefix's minimum and maximum. Checking the next value against each one extends the description. Starting from a real element handles all-negative inputs correctly.

Complete solution – C++23

```
#include <iostream>

void minMax(const int a[], int n, int& minimum, int& maximum) {
    minimum = a[0];
    maximum = a[0];
    for (int i = 1; i < n; ++i) {
        if (a[i] < minimum) { minimum = a[i]; }
        if (a[i] > maximum) { maximum = a[i]; }
    }
}

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0, low = 0, high = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    minMax(a, n, low, high);
    std::cout << low << ' ' << high << '\n';
    return 0;
}
```

Trace and boundary checks. The references name the caller's `low` and `high`. They must refer to separate variables outside the input array, as required by the helper's contract. A one-element array has identical minimum and maximum.

Cost. One scan gives $O(n)$ time and $O(1)$ auxiliary space. Reference parameters do not require copying the array.

Solution 14 – Prime checks across test cases

Idea and why it works. If $x = ab$ with $a, b > 1$, at least one factor is at most $\sqrt{x}$; otherwise their product would exceed x . Testing every integer divisor from 2 through that boundary therefore finds every composite input. The condition `d <= value / d` avoids multiplying `d * d`.

Complete solution – C++23

```
#include <iostream>

bool isPrime(int value) {
    if (value < 2) { return false; }
    for (int d = 2; d <= value / d; ++d) {
        if (value % d == 0) { return false; }
    }
    return true;
}

void solve() {
    int value = 0;
    std::cin >> value;
    if (isPrime(value)) { std::cout << "YES\n"; }
    else { std::cout << "NO\n"; }
}

int main() {
    int t = 0;
    std::cin >> t;
    for (int test = 0; test < t; ++test) { solve(); }
    return 0;
}
```

Trace and boundary checks. Check 1, 2, negative values, a square such as 49, and a prime such as 97. Returning from `isPrime` finishes that check, not the remaining test cases.

Cost. Each $x \geq 2$ takes $O(\sqrt{x})$ worst-case time; smaller inputs take $O(1)$. With positive upper bound $M \geq 2$, total time is $O(t\sqrt{M})$, auxiliary space $O(1)$.

Solution 15 – Count index pairs with a given sum

Idea and why it works. Every allowed pair has exactly one smaller index i and one larger index j . The nested loops visit that combination once, never visit its reversed duplicate, and never pair an index with itself. Increase the count only when the wide sum matches.

Complete solution – C++23

```
#include <iostream>

long long countPairsWithSum(const int a[], int n, long long target) {
    long long count = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if (1LL * a[i] + a[j] == target) { ++count; }
        }
    }
    return count;
}

int main() {
    const int CAPACITY = 2000;
    int a[CAPACITY]{};
    int n = 0;
    long long target = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    std::cin >> target;
    std::cout << countPairsWithSum(a, n, target) << '\n';
    return 0;
}
```

Trace and boundary checks. The sample matches index pairs (0, 1) and (2, 3). Four copies of 2 with target 4 produce six pairs, not one. Empty and one-element arrays produce zero.

Cost. There are exactly $n(n-1)/2$ comparisons: $O(n^2 + 1)$ time and $O(1)$ auxiliary space. This quadratic method is chosen for the stated small bound, not for arbitrarily large arrays.

Solution 16 – Count words using two helpers

Idea and why it works. `insideWord` records whether the previous character belongs to the current word. A space resets it. A non-space starts a new word only when it was false; later characters in that same run must not add more words.

Complete solution – C++23

```
#include <iostream>
#include <string>

bool isSeparator(char c) {
    return c == ' ';
}

int countWords(const std::string& text) {
    int words = 0;
    bool insideWord = false;
    int n = static_cast<int>(text.size());
    for (int i = 0; i < n; ++i) {
        if (isSeparator(text[i])) { insideWord = false; }
        else {
            if (!insideWord) { ++words; }
            insideWord = true;
        }
    }
    return words;
}

int main() {
    std::string text;
    std::getline(std::cin, text);
    std::cout << countWords(text) << '\n';
    return 0;
}
```

Trace and boundary checks. Leading, repeated, and trailing spaces create no words by themselves. An empty or all-space line returns zero. The helper uses the exact separator rule in the statement, not a broader language-dependent idea of a word.

Cost. There is one constant-cost separator call per character: $O(L + 1)$ time and $O(1)$ auxiliary space. More helper calls do not automatically change the growth rate.

Solution 17 – Repeated threshold queries

Idea and why it works. For one threshold, the helper increases its counter once for each qualifying array position. Each call has its own local counter initialized to zero, so answers do not leak across queries. The array remains available for every query.

Complete solution – C++23

```
#include <iostream>

int countAtLeast(const int a[], int n, int threshold) {
    int count = 0;
    for (int i = 0; i < n; ++i) {
        if (a[i] >= threshold) { ++count; }
    }
    return count;
}

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0, q = 0;
    std::cin >> n >> q;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    for (int query = 0; query < q; ++query) {
        int threshold = 0;
        std::cin >> threshold;
        std::cout << countAtLeast(a, n, threshold) << '\n';
    }
    return 0;
}
```

Trace and boundary checks. With thresholds 7, 0, and 8, the sample counts are 2, 3, and 0. If the array is empty, every answer is zero, but reading and printing the q queries still costs work.

Cost. Whole-program time is $O(C + n + q + nq)$, including fixed-capacity initialization. Helper auxiliary space is $O(1)$, stored-array space $O(C)$; thresholds are processed one at a time.

Solution 18 – Optional challenge: keep the order

Idea and why it works. Match the next needed character at its earliest available position. Taking that position leaves at least as much remaining text as taking a later matching position, so it cannot prevent a possible completion. Increase **needed** only on a match; success means every required character was matched in order.

Complete solution – C++23

```
#include <iostream>
#include <string>

bool isSubsequence(const std::string& small, const std::string& large) {
    int needed = 0;
    int n = static_cast<int>(small.size());
    int m = static_cast<int>(large.size());
    for (int i = 0; i < m && needed < n; ++i) {
        if (large[i] == small[needed]) { ++needed; }
    }
    return needed == n;
}

int main() {
    std::string small, large;
    std::getline(std::cin, small);
    std::getline(std::cin, large);
    if (isSubsequence(small, large)) { std::cout << "YES\n"; }
    else { std::cout << "NO\n"; }
    return 0;
}
```

Trace and boundary checks. For **ace** in **abcde**, the needed index advances on **a**, **c**, and **e**. For **aec**, the order cannot be completed. The loop condition prevents indexing an already fully matched or empty smaller string.

Cost. Let the lengths be n and m . The helper visits at most m characters: $O(m + 1)$ time, $O(1)$ auxiliary space. Reading both lines makes whole-program time $O(n + m + 1)$.

3.31 Explained complexity drills

Use these explanations to check the reasoning, not only the final Big-O expression. Each answer counts source-level work under the stated model.

Drill 1 – A step of three

The visited indices are $0, 3, 6, \dots$, strictly smaller than n . The increment count is $\lceil n/3 \rceil$, including zero visits when $n = 0$. For $n = 8$, the three indices are 0, 3, and 6.

The step is a constant, so the work still grows linearly: $\Theta(n + 1)$ time including loop setup and the final failed comparison. Only a counter and index are stored, so auxiliary space is $O(1)$.

Drill 2 – Two inputs, three consecutive loops

The first loop contributes n increments, the second contributes m , and the last contributes 10. Total increments are $n + m + 10$, giving $\Theta(n + m + 1)$ time and $O(1)$ auxiliary space.

The loops are consecutive, not nested. Multiplying nm would charge work that is never performed. Neither independent input length can be dropped without a stated relation between them.

Drill 3 – Including the diagonal

For $i = 0$, the inner loop runs once; for $i = 1$, it runs twice. The complete count is

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

The diagonal pairs (i, i) are included because the condition is $j \leq i$. The total time is $\Theta(n^2 + 1)$, and auxiliary space is $O(1)$. Dividing by two does not change the quadratic growth rate.

Drill 4 – Two loops, but only five inner visits

Every one of the n outer iterations performs exactly five inner increments. The total is $5n$, so time is $\Theta(n + 1)$ and auxiliary space is $O(1)$.

There are two syntactic loops, but only one input-dependent repetition count. “Two loops means quadratic” is not a valid rule.

Drill 5 – Growing chunks

For $n = 10$, the widths are 1, 2, 4, and 8. Total visits are $1 + 2 + 4 + 8 = 15$. In general, let $k = \lfloor \log_2 n \rfloor$. Then

$$1 + 2 + 4 + \dots + 2^k = 2^{k+1} - 1.$$

Since $2^k \leq n < 2^{k+1}$, this sum lies between n and $2n$, with the upper endpoint excluded. Therefore time is $\Theta(n)$, not the looser upper bound $O(n \log n)$; auxiliary space is $O(1)$.

The outer-loop overhead is logarithmic and does not dominate the linear sum of inner visits. A logarithmic number of different-size chunks does not necessarily total $n \log n$.

Drill 6 – Is the inner counter reset?

In A, j increases from 0 to n only once. After the first outer iteration, every later inner check immediately fails. There are n increments of j in total and n outer iterations: $\Theta(n + 1)$ time.

In B, each of the n outer iterations gets a fresh $j = 0$ and performs n increments. Total inner increments are n^2 , so time is $\Theta(n^2 + 1)$. Both versions use $O(1)$ auxiliary space.

Drill 7 – A helper called many times

On a palindrome, one check examines $\lfloor L/2 \rfloor$ pairs before returning true. The helper takes $\Theta(L + 1)$ worst-case time, including constant work for an empty string. Performing q calls gives $\Theta(q(L + 1))$ source-level work.

The reference parameter does not copy the string. Each completed call releases its few local variables before the next call begins, so peak auxiliary space is $O(1)$, not $O(q)$. The existing string's storage is outside that auxiliary-space measurement.

Because the input does not change, a real solution that needed this same decision repeatedly could calculate it once. The drill analyzes the stated repeated-call version, not that improvement.

Drill 8 – An almost-empty body hides a copy

The string parameter is passed by value. Initializing it from the existing string copies n characters, even though calling `empty()` inside the body is constant-time. One such call has a linear copy cost; n calls therefore have $\Theta(n^2)$ modeled copying work.

Only one parameter copy exists at a time in these sequential calls. The helper's peak auxiliary space is $O(n)$, not $O(n^2)$. Changing the interface to `const std::string&` would remove the unnecessary input copy for this read-only test.

Drill 9 – Repeated halving

For 20, the values after each division are 10, 5, 2, and 1, giving four increments. For any positive integer n , the exact count is $\lfloor \log_2 n \rfloor$; for $n = 1$, it is zero.

After k divisions, the remaining value is $\lfloor n/2^k \rfloor$. The loop stops when it reaches 1. Time is $\Theta(1 + \log n)$, and auxiliary space is $O(1)$. The added one covers the immediate-stop case without pretending that setup is free.

Drill 10 – Total constraints across test cases

Without a per-case cap,

$$\sum_i n_i^2 \leq \left(\sum_i n_i \right)^2 \leq S^2 = 200000^2 = 40\,000\,000\,000.$$

This can be reached by one case of length S . With every $n_i \leq 1000$, use $n_i^2 \leq 1000n_i$, giving

$$\sum_i n_i^2 \leq 1000S = 200\,000\,000.$$

That bound is achievable by 200 cases of length 1000 if no other restriction prevents that many cases. These are counted visits, not seconds. Per-case setup and input/output are additional costs when analyzing the entire program.

Drill 11 – Peak memory is not cumulative allocation

The helper's extra copy contains n integers, so its auxiliary space is $\Theta(n)$. At peak, both the input and copy are alive, so total live array storage is $2n$ integers: still $\Theta(n)$, not a different growth class.

For $n = 10^6$ and 4-byte integers, each array occupies 4 000 000 bytes; together they occupy 8 000 000 bytes, about 8 decimal MB or 7.63 MiB, before the excluded overhead. Only one helper copy is live at once, so 20 sequential calls do not make the peak 20 times larger. They do multiply cumulative copying work.

Drill 12 – Optional: calls over time versus calls still alive

Factorial has the chain $n, n-1, \dots, 1, 0$. There are $n+1$ calls, each doing constant local work: $\Theta(n+1)$ time. The unfinished calls can all exist at once, so ordinary stack space is also $\Theta(n+1)$.

Naive Fibonacci branches into up to two calls per level, with depth at most proportional to n . A binary tree with that depth has at most an exponential number of nodes, giving a valid $O(2^n)$ time upper bound. This is an upper bound, not a claim that the tight bound is $\Theta(2^n)$.

At any instant, execution has only one active chain of unfinished calls, plus constant-size local data per call. Its maximum depth is $\Theta(n+1)$, so peak auxiliary stack space is linear. Total calls performed over time and calls alive simultaneously answer different questions.

3.32 Quick reference, glossary, and readiness checklist

These pages are for revision after the explanations. When a rule feels unfamiliar, return to its worked example rather than memorizing the table in isolation.

Choose the interface that matches the job

Interface pattern	Meaning
<code>int f(int x)</code>	Receive a separate integer value and return one integer. Assigning to <code>x</code> does not assign to the caller's variable.
<code>void f(int& x)</code>	Access the caller's integer through another name. The helper can change it.
<code>int f(const int a[], int n)</code>	Read the caller's first <code>n</code> array elements without copying them. The length is an explicit part of the contract.
<code>void f(int a[], int n)</code>	Access and potentially modify the caller's active array elements.
<code>int f(const std::string& s)</code>	Read the existing string without making a by-value parameter copy.
<code>std::string f(std::string s)</code>	Work on a local string value and return text by value. Copying the supplied string may be intentional.
<code>void solve()</code>	Perform one test case's actions without receiving explicit parameters or returning a result.

Function checklist

A declaration must be visible before a call. A definition supplies the body. Each parameter has a type. A call supplies arguments in corresponding positions. The return type describes the value the caller receives, not how earlier arithmetic is evaluated.

Use `return value;` for a result, `return;` to leave a `void` helper early, and output statements only when the task needs output. For ordinary non-void helpers, every reachable result-producing path must return a value.

Two rules that prevent many bugs

An array parameter written with `[]` does not carry its active length; pass that length explicitly. A string copied by value is different: it is a separate string object with its own length and characters. Do not apply one of these behaviors to the other.

A compact complexity guide

Assume the repeated body has constant cost, the bounds below are the varying input sizes, and no overflow changes control flow. Empty inputs add constant setup work where permitted.

Work pattern	Typical bound and reason
A fixed number of arithmetic operations	$O(1)$: the count does not grow with the input size.
A complete scan of n elements	$O(n)$: constant work per element.
One scan of n , followed by one of m	$O(n + m)$: consecutive costs add.
Every pair from n items	$O(n^2)$: a quadratic number of pairs, even when each is visited only once.
Every cell in an r -by- c table	$O(rc)$ for positive dimensions: one visit per cell.
Repeatedly halve a positive integer n	$O(1 + \log n)$: each step removes roughly half of what remains.
n independent halving loops of size n	$O(n(1 + \log n))$: logarithmic work repeated n times.
Chunks of sizes $1, 2, 4, \dots$, up to n	$O(n)$: the sum of the chunk sizes is below $2n$.
q calls to a linear helper on n values	$O(q(n + 1))$: include the work inside every call.
Recursive factorial using one smaller call	$O(n + 1)$ time and stack space in the ordinary recursive model.
Naive recursive Fibonacci	$O(2^n)$ valid time upper bound; linear peak stack depth.

Analyze in this order

Name the sizes. Count repetitions. Include hidden work inside calls and copies. Add consecutive costs; multiply only when each repeated job really occurs for every outer repetition. Keep the dominant growth terms, then state whether the result is worst-case, best-case, or another specified measure.

For space, count input storage, result storage, and temporary storage separately. When reporting auxiliary space, state which of these is excluded. Count maximum simultaneous use, not the sum of all allocations across sequential calls.

Glossary: the words used in this session

Term	Meaning in this chapter
Function	A named unit of work that can be called.
Definition	The function's declaration together with its body.
Declaration / proto-type	Information about a function's name and type, allowing calls before its later definition.
Parameter	A named input declared in the function's parameter list.
Argument	An expression supplied at a particular call.
Caller	The part of the program that requests the function call.
Return value	A result sent back to the caller, not necessarily printed.
Contract	The helper's valid inputs, promised result, and permitted changes.
Scope	The region where a particular declaration's name can be used.
Lifetime	The period during execution when an object exists.
Reference / alias	Another way to refer to the same object.
Read-only access	Access that cannot modify an object through that particular interface.
Predicate	A function that answers a yes-or-no question.
Time complexity	How modeled work grows with named input sizes.
Auxiliary space	Extra storage used beyond whatever input/output storage the stated convention excludes.
Big-O	An asymptotic upper bound, not a synonym for worst case.
Theta	An asymptotically tight growth bound.
Worst case	The greatest cost among inputs of the specified size.
Recursion	A function calling itself, directly or indirectly.
Base case	A case solved without another recursive call.
Call depth	How many unfinished calls are simultaneously on one chain.

Before moving on, explain these without copying

1. Define a small helper, call it from `main`, and trace the parameter values and returned result.
2. Explain why returning and printing are different. Show what happens after an early return.
3. Predict whether an assignment changes the caller when using `int`, `int&`, `int a[]`, or `std::string` as a parameter.
4. Write a read-only helper for an array with an explicit active length, and one for a string without an unnecessary copy.
5. State the contract for an empty input and for an index-returning search. Test both a missing value and repeated values.
6. Count a single scan, consecutive scans, a rectangular nested loop, and a triangular nested loop.
7. Explain a halving loop using a small trace. Distinguish it from a loop that subtracts a fixed amount.
8. Include helper calls, string copying, input/output, and every test case in your time estimate.
9. Distinguish input storage from auxiliary space, and total calls over time from peak live calls.
10. Choose a plausible approach from the constraints, then explain why its counted work is acceptable enough to test rather than claiming a universal operations-per-second rule.

A small self-test

Without running code, explain what `incrementCopy(x)` and `increment(x)` do differently in E08 and E09. Then explain why E24's nested syntax still performs only a linear number of total increments. Finally, explain why calling a linear helper from a loop can make the caller quadratic.

Your goal is explanation, not vocabulary

You are ready for the next session when you can write a small correct helper, test its contract, and explain its work and memory in your own words. You do not need to master optional recursion, overload resolution, or every unusual loop pattern before learning the next container.

Where the training goes next

Session 4 introduces `vector`, `pair`, `deque`, `queue`, `priority_queue`, and `stack`. The interface and complexity habits from this session will help you ask what each operation does, what it changes, and how much work it requires. Those container lessons are not included in this edition.

3.33 Technical notes and references

These pages document the assumptions behind the beginner-facing explanations. They are for checking a detail or a second reading, not extra prerequisites for the core practice.

Language version and source boundaries

The examples target **C++23**. Language references point to the fixed **N4950 C++23 working draft**, using a public HTML rendering of the committee wording, rather than a moving draft that may include later language changes. The wording is a technical reference, not a beginner tutorial. Compiler references are GCC's own documentation; the asymptotic-analysis reading is MIT course material.

Ordinary named functions, built-in arrays, and `std::string` are the foundation here. Templates, lambdas, function pointers, general pointer arithmetic, dynamic allocation, STL container internals, and recurrence-solving techniques are not required. Overloading, default arguments, recursion, and matrix parameters are explicitly marked as optional.

Every complete program uses standard headers and its own `main`. Compile one program at a time. No variable-length built-in arrays or `bits/stdc++.h` are required. The training compiler accepts the C++23 language mode using `-std=c++23`.^[3-18]

Integers, lengths, characters, and helper contracts

The tested environment has a 32-bit `int` and a 64-bit `long long`. C++ does not universally require those exact widths. Input bounds such as 10^9 and the fixed capacity examples assume a sufficiently wide `int`, as in that training environment. Wide totals use `long long`; multiplication must be widened before it overflows, not after.

String exercises use the stated ASCII characters, not arbitrary Unicode text. Their length bounds justify converting `size()` to `int` in these examples. A character count or character-by-character reversal is not presented as a general solution for human-language text in every encoding.

Array helpers receive an explicit active length. The caller guarantees that many accessible elements, and helpers that inspect `a[0]` require a nonempty input. A `const` parameter restricts mutation through that access path; it does not imply that no other alias can modify the object. Multiple output references have explicit non-aliasing conditions where those are necessary for the supplied algorithm.

Cost model and notation

The chapter uses a fixed-width unit-cost model: ordinary integer arithmetic, comparisons, and valid array indexing are counted as constant-cost operations. This is a useful model for these bounded C++ problems, not a statement that arbitrary-precision arithmetic or arbitrarily large input tokens always cost constant time.

A bound in numeric value n differs from a bound in the number of digits needed to represent n . State the variable being measured. Big-O is an upper-bound notation; worst-case and average-case describe which inputs or probability model supply the cost being bounded. A tight bound is stated with Θ when that distinction matters.^[3-13]

Theoretical growth describes a family of problem sizes; a printed capacity exists so a small teaching program is well-defined for its stated bounds. The actual array allocation has C elements even when only $n < C$ are active. Describing the algorithm's scalable input storage as $O(n)$ assumes capacity scales with the problem size; it does not mean a particular built-in array changes length at runtime.

Analysis of helpers excludes already-supplied input storage unless stated otherwise. Returning or building another string can add storage proportional to its length. Recursion uses the ordinary model of constant-size bookkeeping per unfinished call. The C++ language does not require a particular physical stack representation or promise that an implementation eliminates recursive calls.

Algorithm work is not a stopwatch guarantee

The counters in E19–E26 illustrate counts on paper. Optimizing compilers may simplify or eliminate computations with no observable effect; this does not turn a counted-loop lesson into a reliable microbenchmark. No table here promises a universal number of operations per second.

Bounds on `std::sort` refer to comparisons, whose own cost depends on the compared objects. The standard comparison bound does not by itself specify an auxiliary-space guarantee for every implementation. Similarly, an ordinary function call's short spelling does not hide the cost of its body or value parameters.^[3-17]

Original exercises and reference markers

Examples, traces, practice problems, and explanations are written for this training. A superscript such as ^[3-2] links to the corresponding source entry below. The first number identifies Session 3, so references do not collide when chapters are collected.

What was checked for this edition

The source package contains **31 complete Session 3 examples** and **18 complete practice solutions**. All **49 programs** compiled successfully with **GCC 14.2.0** in C++23 mode, with strict diagnostics.

They passed **6911 input/output checks**: **113 fixed sample/boundary checks** and **6798 generated or exhaustive-small-case checks**. The test runner compares exact standard output with independently constructed expected results and rejects compiler diagnostics, sanitizer diagnostics, unexpected standard error, and timeouts.

Coverage includes zero and negative numeric inputs where allowed, wide sums, all-negative arrays, repeated minima, missing targets, empty sequences, one-element inputs, capacity boundaries, full-length lines, leading/trailing spaces, repeated test cases, powers-of-two boundaries, square composites, prime inputs, and the stated optional-recursion bounds.

The verification build used:

```
-std=c++23 -pedantic-errors -Wall -Wextra  
-Wconversion -Wshadow -Werror -O1 -g  
-D_GLIBCXX_ASSERTIONS -fsanitize=undefined  
-fno-sanitize-recover=all
```

These flags record the checking configuration, not a requirement that beginners memorize them. Undefined-behavior instrumentation and library assertions check the paths actually executed; passing tests is not a proof for every possible input. Deliberately wrong teaching fragments are labeled and are not treated as compilable solutions.^{[3-19][3-20]}

Reproduce the checks or rebuild the PDFs

Run `python3 tests/verify_session_03.py` from the project root with Python 3.10 or later and a suitable GNU C++ compiler installed. The fixed manifest and machine-readable verification report are stored beside the test runner. The README explains normal compilation and the optional strict verification setup.

Build `session_03.tex` for the standalone Session 3 PDF. Build `book.tex` for the collected edition containing Sessions 1–3. Both load this chapter and the same external C++ source files, so the printed listings remain synchronized with the tested programs. Sessions 4–6 remain roadmap entries, not completed chapters in this edition.

Language and library sources

[3-1] C++23 N4950: function declarators and parameter adjustments

<https://timsong-cpp.github.io/cppwp/n4950/dcl.fct>

[3-2] C++23 N4950: function calls and parameter initialization

<https://timsong-cpp.github.io/cppwp/n4950/expr.call>

[3-3] C++23 N4950: return statements

<https://timsong-cpp.github.io/cppwp/n4950/stmt.return>

[3-4] C++23 N4950: general function-definition rules

<https://timsong-cpp.github.io/cppwp/n4950/dcl.fct.def.general>

[3-5] C++23 N4950: block scope

<https://timsong-cpp.github.io/cppwp/n4950/basic.scope.block>

[3-6] C++23 N4950: automatic storage duration

<https://timsong-cpp.github.io/cppwp/n4950/basic.stc.auto>

[3-7] C++23 N4950: reference declarations

<https://timsong-cpp.github.io/cppwp/n4950/dcl.ref>

[3-8] C++23 N4950: reference initialization

<https://timsong-cpp.github.io/cppwp/n4950/dcl.init.ref>

[3-9] C++23 N4950: array-to-pointer conversion

<https://timsong-cpp.github.io/cppwp/n4950/conv.array>

[3-10] C++23 N4950: string construction and copy behavior

<https://timsong-cpp.github.io/cppwp/n4950/string.cons>

[3-11] C++23 N4950: overloading

<https://timsong-cpp.github.io/cppwp/n4950/over>

[3-12] C++23 N4950: default arguments

<https://timsong-cpp.github.io/cppwp/n4950/dcl.fct.default>

Analysis, library operations, and compiler documentation

[3-13] MIT 6.006, Spring 2020: Recitation 1, asymptotic notation

https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/c6d8f06c6f11e3342633dec85498f551_MIT6_006S20_r01.pdf

[3-14] C++23 N4950: string capacity, size, and empty

<https://timsong-cpp.github.io/cppwp/n4950/string.capacity>

[3-15] C++23 N4950: string substr

<https://timsong-cpp.github.io/cppwp/n4950/string.substr>

[3-16] C++23 N4950: string comparison

<https://timsong-cpp.github.io/cppwp/n4950/string.compare>

[3-17] C++23 N4950: sort and its comparison bound

<https://timsong-cpp.github.io/cppwp/n4950/sort>

[3-18] GCC 14.2: language-standard options

<https://gcc.gnu.org/onlinedocs/gcc-14.2.0/gcc/C-Dialect-Options.html>

[3-19] GCC 14.2: warnings and strict-conformance diagnostics

<https://gcc.gnu.org/onlinedocs/gcc-14.2.0/gcc/Warning-Options.html>

[3-20] GCC 14.2: instrumentation and sanitizer options

<https://gcc.gnu.org/onlinedocs/gcc-14.2.0/gcc/Instrumentation-Options.html>

[3-21] C++23 N4950: general container requirements and copy costs

<https://timsong-cpp.github.io/cppwp/n4950/container.reqmts>

[3-22] C++23 N4950: string input and output

<https://timsong-cpp.github.io/cppwp/n4950/string.io>